PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Testy prvočíselnosti

2016 Vít Schnaubelt

Anotace

Student zpracuje podrobné porovnání testů prvočíselnosti (počínaje základními, jako je např. Fermatův test, konče algoritmem AKS). Všechny studované algoritmy budou implementovány a otestována jejich efektivita. Diplomant dále ukáže použití testů prvočíselnosti v kryptografii a dalších disciplínách.

Děkuji vedoucímu práce RNDr. Eduardu Bartlovi, Ph.D., za vedení a veškerou pomoc při řešení problému, při zpracovávání této diplomové práce. Dále bych chtěl poděkaovat rodině za podporu v celém průběhu vytváření práce.

Obsah

Seznam obrázků

Seznam tabulek

1. Úvod

Cílem práce je popsat, naimplementovat a otestovat různé metody detekce prvočíslenosti od nejjednoduších, přes různé pravděpodobnostní po deterministické jako AKS. Práce obsahuje vzájemné porovnání složitosti těchto metod. Nakonec práce obsahuje souhrn využití prvočísel v kryptografii a jiných disciplínách.

První zmínky o prvočíslech pochází ze starověkého Egypta. Dochované záznamy ukazují, že Egypťané měli jiné formy pro prvočísla a složená čísla. První záznamy o studiu prvočísel pochází z antického Řecka, kde přibližně 300 let př.n.l. Euklidés sepsal věty o prvočíslech včetně věty o nekonečném množství prvočísel a základní věty aritmetiky. Eratosthenes ve stejném období ukázal jednoduchou metodu výpočtu prvočísel tzv. Eratosthenovo síto.

Další významnou osobou v oblasti prvočísel se stal v 17. století Pierre de Fermat se svojí malou větou (Fermatova malá věta). Fermat také prohlásil, že všechna čísla ve formátu $2^{2^n} + 1$ (Fermatova čísla) jsou prvočísla. Toto tvrzení zkontroloval pro čísla do $n = 4$ $(2^{16} + 1)$. Nicméně, jak později sám zjistil, jeho tvrzení neplatilo pro $2^{32} + 1$ a ani pro další (není známo žádně větší Fermatovo číslo). Další Francouz, mnich Marin Mersenne, se zabýval prvočísly ve tvaru $2^p - 1$ kde p je prvočíslo. Tyto prvočísla jsou nazývána Mersenneova prvočísla. Euler v 18. století ukázal, že nekonečná řada $\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \dots$ je divergentní a že dokonalá čísla(číslo, které se rovná součtu svých dělitelů bez sebe samotného) jsou čísla ve tvaru $2^{p-1}(2^p - 1)$. Na konci 19. století nezávisle na sobě vědci Jacques Hadamard a Charles Jean de la Vallée-Poussin dokázali prvočíselnou větu.

Prvočísla se vyskytují i v přírodě. Existuje hypotéza, že cikády využívají prvočísel pro zvýšení své šance na přežití druhu. Cikády žijí většinu svého života pod zemí, ale jednou za 7, 13 nebo 17 let se zakuklí a vylétají na povrch. Cikády pravděpodobně volí prvočísla, protože pokud existuje predátor, který by se objevoval v pravidelných intervalech jiných než (7, 13, 17), mají větší šanci, že se ve stejném roce nepotkají. Pokud by se predátor objevoval každých 2, 3, 4, 6 nebo 12 let mají cikády při cyklu 13 a 17 let přibližně o 2% menší šanci na potkání predátorů než při cyklu 14, 15 let. Tato malá výhoda je pro cikády pravděpodobně natolik dostatečná, že volí prvočíselné roky.

V dnešní době se prvočísla využívají pro šifrování komunikace. Cílem metod je ověřit zdali je náhodné číslo prvočíslem a následné využití prvočísla pro vytvoření privátních a veřejných klíčů.

2. Prvočíslo

Prvočíslo je přirozené číslo větší než 1, které lze dělit pouze jedničkou a samo sebou. Přirozené číslo větší než 1, které není prvočíslo, se nazývá složené. Například číslo 7 je prvočíslo (lze dělit pouze jedničkou a sedmičkou) a číslo 8 je složené (lze dělit čísly 1, 2, 4, 8). Základní věta aritmetiky říká, že každé přirozené číslo

větší než 1 lze rozložit na součin prvočísel.

Vlastnost býti prvočíslem se nazývá prvočíselnost a zjišťuje se testováním prvočíselnosti. Pro testování prvočíselnosti existují různé metody od pomalých, jednoduchých (brute-force) až po rychlé, pravděpodobnostní.

Matematik Euclid přibližně 300 let př.n.l. dokázal existenci nekonečně mnoha prvočísel, ale neexistuje žádná formule na rozdělení prvočísel od čísel složených. Nicméně lze předpokládat že číslo n je prvočíslo s pravděpodobností nepřímo úměrnou počtu číslic, nebo logaritmu n.

2.1. Základní věta aritmetiky

Základní věta aritmetiky tvrdí, že každé přirozené číslo větší než 1 lze jednoznačně rozložit na součin prvočísel. Tedy, že každé číslo větší než 1 lze rozložit na posloupnost prvočísel, jejichž součin bude roven původnímu číslu. Jednoznačností je myšleno, že pokud nebude brán ohled na uspořádání, bude rozklad právě jeden. Například číslo 30 lze rozložit na součin $2 \cdot 3 \cdot 5$ neboli $2^1 \cdot 3^1 \cdot 5^1$.

Věta 1 (Základní věta aritmetiky)**.** *Každé přirozené číslo větší než 1 lze rozložit na právě jeden součin prvočísel (pokud neuvažujeme pořadí).*

Důkaz. Důkaz matematickou indukcí

- Pro prvočísla platí triviálně(tedy i pro číslo 2). Pro prvočíslo p bude posloupnost p.
- Pokud platí pro všechny $i \leq n$, tak $n+1$ je prvočíslo, nebo lze n rozdělit na součin dvou menších čísel a spojením jejich rozkladů získáme nějaký rozklad výsledného čísla.
- Stačí dokázat že výsledný rozklad bude právě jeden. Sporem pokud pro $n+1$ existují dva různé rozklady tak musely existovat i dva různé rozklady pro číslo ze kterého je $n+1$ složené.

□

2.2. Existence nekonečně mnoha prvočísel

Euclidův důkaz sporem počítá s konečnou množinou prvočísel. Pokud mezi sebou vynásobíme všechna čísla z množiny a přičteme jedničku. Výsledné číslo zcela určitě nebude dělitelné žádným číslem z množiny. Výsledek tedy bude další prvočíslo, nebo bude dělitelný prvočíslem které není v množině. Pro množinu čísel $(2, 5, 11)$ získáme $(2 \cdot 5 \cdot 11) + 1 = 111$. Číslo 111 není prvočíslo, protože je dělitelné číslem 3 tedy dalším prvočíslem.

Věta 2 (Euclidův důkaz nekonečného množství prvočísel)**.** *Předpokládejme konečnou množinu prvočísel.*

Důkaz. Zvažme $N = 1 + \prod_{p \in S} p$.

- N je jako každé přirozené číslo dělitelné minimálně jedním prvočíslem (N může být prvočíslo samo).
- N není dělitelné žádným číslem z S (pokud vynásobíme čísla větší než 1 a přičteme k nim právě jedničku. Tak výsledek nemůže být dělitelný žádným z původních čísel).
- Musí tedy existovat nějaké prvočíslo $x \leq N$ takové, že $x \notin S$, což je ve sporu s předpokladem.

□

2.3. Distribuce prvočísel (Prvočíselná věta)

Prvočíselná věta hrubě popisuje distribuci prvočísel mezi přirozenými čísly. Prvočíselná věta říká, že pokud vezmeme nějaké velké číslo N, šance, že N je prvočíslo je $\frac{1}{\ln N}$. Tedy průměrná vzdálenost mezi dvěma prvočísly poblíž N je $\ln N$.

Věta 3 (Prvočíselná věta)**.** *Nechť $\pi(x)$ je prvočíselná funkce udávající počet prvočísel $p \leq x$ např. pro $x = 10$ je $\pi(x) = 4$ (prvočísla 2, 3, 5, 7). Věta potom říká, že limita podílu funkcí $\pi(x)$ a $\frac{x}{\ln(x)}$ pro x jdoucí k nekonečnu je rovna nule. Vzorcem*

$$\lim_{x \to \infty} \frac{\pi(x)}{x \ln(x)} = 1.$$

Pomocí asymptotické notace

$$\pi(x) \approx \frac{x}{\ln(x)}.$$

Vzorec nic neříká o rozdílu těchto funkcí pro x jdoucí k nekonečnu. Tento rozdíl je komplikovaný a je spojen s nevyřešenou Riemannovou domněnkou. Věta vyjadřuje, že výraz $\frac{x}{\ln}(x)$ aproximuje $\pi(x)$ tak, že chyba aproximace se blíží k nule pro x jdoucí k nekonečnu.

2.4. Prvočíselnost jedničky

Většina prvních Řeků nepovažovala jedničku za číslo. Takže nebyla považována ani za prvočíslo. Tento přístup se změnil v 19. století kdy většina matematiků začala jedničku za prvočíslo považovat. Většina pravidel která platí pro prvočísla

by platila i pro jedničku, ale existují i pravidla která by neplatila. Jedním z pravidel které by nefungovalo je například základní věta aritmetiky, kde bychom pro číslo 10 mohli dostali rozklady $1 \cdot 2 \cdot 5$ a $2 \cdot 5$. Dále by nefungovalo Erasetonovo síto a další.

2.5. Největší nalezené prvočíslo

Největší dosavadní známé prvočíslo (nalezeno 2016) je $2^{74207281} - 1$ a má 22338618 číslic. Toto číslo bylo nalezeno Great Internet Mersenne Prime Search institutem a jedná se o Mersenneovo prvočíslo. Posledních 16 největších prvočísel bylo nalezeno právě tímto institutem a všechna byla ve formátu $2^p - 1$. Nadace Electronic Frontier Foundation motivuje k hledání vyšších prvočísel peněžně. Za nalezení prvočísla se sto miliony číslic nabízí 150 000 dolarů, za prvočíslo s miliardou číslic pak 250 000 dolarů.

2.6. Základy modulární aritmetiky

Čísla a a b jsou kongruentní modulo n značeno $a \equiv b \mod n$. Jestliže rozdíl $a - b$ je dělitelný číslem n, tedy n dělí $a - b$. Lze zapsat i jako $a \mod n = b \mod n$. Například pokud máme $a = 25$, $b = 10$ jsou kongruentní modulo $n = 5$ protože $a - b = 25 - 10 = 15$ a 15 je dělitelné číslem 5. V textu se dále využívá značení $gcd(a, b)$, které označuje největšího společného dělitele čísel a a b.

3. Testování prvočíselnosti

Moderní metody testování prvočíselnosti lze rozdělit do dvou hlavních skupin. První skupinou jsou deterministické algoritmy, které vždy správně rozhodnou, zdali je dané číslo prvočíslo nebo číslo složené. Mezi tyto algoritmy patří jednoduché triviální dělení, ale i složitější jako AKS. Druhá skupina obsahuje pravděpodobnostní (Monte-Carlo) algoritmy. Tato skupina obsahuje rychlejší algoritmy s různě velkou pravděpodobností správného označení. Patří sem Fermatův test prvočíselnosti, Solovay-Strassen, Miller-Rabin a další.

3.1. Triviální dělení (Brut force)

Triviální dělení je nejjednodušší metoda kontroly prvočíselnosti daného čísla n. Číslo n postupně dělíme všemi čísly většími než 1 a menšími nebo rovnými $\sqrt{n}$ (pokud bude číslo složené tedy $n = ab$ kde $a, b > 1$ není možné, aby se skládalo z čísel větších než $\sqrt{n}$). Pro $n = 37$ bychom kontrolovali dělitele $2, 3, 4, 5, 6$, žádný z nich nedělí n bezezbytku, takže n je prvočíslo. Metodu lze vylepšit testováním pouze lichými čísly(kontrola prvočísla 2) nebo prvočísly $1 < p \leq \sqrt{n}$. Z algoritmu vyplývá, že pro rostoucí n se stává nepoužitelný.

Algoritmus

Vstupen algoritmu je testované číslo $n > 2$. Výstupem je informace o prvočíselnosti.

$n \leftarrow$ *testované číslo*
for $i = 2$ **to** $i \leq \sqrt{n}$ **do**
  if $n \mod i = 0$ **then**
    return *composite*
  else
    return *prime*
  end if
  $i = i + 1$
end for

Kvůli složitosti $\mathcal{O}(2^{\frac{n}{2}})$ je algoritmu vhodný pouze pro malá n. Pro větší čísla se stává nepoužitelný.

3.2. Eratosthenovo síto

Eratosthenovo síto je speciální druh algoritmu který najde všechna prvočísla do velikosti n. Algoritmus na začátku vytvoří pole s hodnotami $2 \ldots n$, poté vezme první číslo seznamu, prohlásí je prvočíslem a vymaže/vynuluje z pole všechna čísla která dělí. Následně se opakuje stejný proces pro druhé nevymazané/nenulové číslo. Proces se opakuje dokud je kontrolované číslo menší jak $\sqrt{n}$. Algoritmus má složitost $\mathcal{O}(n \log(\log n))$ a vyplatí se, pokud potřebujeme vypočítat a uchovat větší množství menších prvočísel. Vzhledem k tomu, že vypočítáváme všechna prvočísla je algoritmus náročný na paměť. Algoritmus je pojmenován po řeckém Matematikovi Eratosthenovi, který ho popsal ve 3. století př.n.l.

Algoritmus

Vstupem algoritmu je velikost pole $n > 2$. Výstupem je pole s hodnotami true na pozicích s prvočísly.

$n \leftarrow$ *velikost pole*
$siev[n] \leftarrow$ *pole hodnot true*
for $i = 2$ **to** $i \leq \sqrt{n}$ **do**
  if $siev[i] = true$ **then**
    for $k = i \cdot 2$ **to** $k \leq n$ **do**
      $siev[k] = false$
    end for
    $k = k + i$
  end if$i = i + 1$
end for

Po projití pole algoritmem v poli zůstanou pouze prvočísla. Algoritmus je možné upravit na pole hodnot typu bool a nastavování false podle dělitelnosti indexů.

3.3. Fermatův test prvočíselnosti

Fermatův test prvočíselnosti je rychlý pravděpodobnostní test, který vychází z Fermatovy malé věty. Algoritmus má složitost $\mathcal{O}(k \log^{2+\epsilon} n)$, kde k je množství provedených testů a ϵ je malé přirozené číslo.

Princip

Věta 4 (Fermatova malá věta)**.** *Jestliže n je prvočíslo a $1 \leq a < n$ pak:*

$$a^{n-1} \equiv 1 \pmod{n}$$

Při testování čísla n vybereme náhodné a z intervalu a zkontrolujeme zdali rovnost platí. Pokud neplatí je číslo n složené. Jestliže rovnost platí pro různé hodnoty a je číslo n pravděpodobně prvočíslo. Během testování se může stát, že n zkontrolujeme pouze pro taková a, pro která rovnost platí ikdyž je n složené. Dále zde existuje nekonečně mnoho složených čísel zvaných Carmichaelova čísla. Pro tato čísla platí $gcd(a, n) = 1$ pro jakékoliv a. I přesto že jsou Carmichaelova čísla podstatně vzácnější než prvočísla Fermatův test se v jeho základní podobě nepoužívá. Z Fermatova testu jsou odvozeny další jako Solovay-Strassenův test nebo Miller-Rabinův.

Algoritmus

Vstupem algoritmu je testované číslo $n > 2$ a počet provedených testů k. Výstupem je informace zdali je číslo složené, nebo prvočíslem s danou pravděpodobností.

$n \leftarrow$ *testované číslo*
$k \leftarrow$ *množství testů*
for $i = 0$ **to** $i \leq k$ **do**
  $a = rand(2, n)$
  if $a^{n-1} \mod n \neq 1$ **then**
    return *composite*
  end if
  $i = i + 1$
end for
return *probably prime*

Příklad

Mějme $n = 55$ a zvolíme $a = 21$. Rovnice vyjde $21^{54} \pmod{55} = 1$, tedy číslo je možná prvočíslo. Pokud zvolíme nové $a = 15$, vyjde rovnice $15^{54} \pmod{55} = 25$ a můžeme prohlásit, že n je číslo složené.

3.4. Solovay–Strassenův test prvočíselnosti

Solovay–Strassenův test prvočíselnosti je jeden z prvních rozšířených pravděpodobnostních algoritmů, díky kterému bylo poprvé možné použít kryptosystém RSA. V dnešní době je již z velké části nahrazen Miller–Rabinovým testem a testem Baillie–PSW. Časová složitost algoritmu je $\mathcal{O}(k \log^3 n)$ s pravděpodobností chyby 2^{-k}.

Princip

Metoda je založena na Eulerově kritériu se zobecněným Legendreovým symbolem.

Věta 5 (Eulerovo kritérium)**.** *Pro každé prvočíslo p a celé číslo a platí:*

$$\tfrac{a^{p-1}}{2} \equiv (\tfrac{a}{p}) \pmod{p}$$

kde $(\frac{a}{p})$ je Legendreův symbol.

Zobecněním Legendreova symbolu je Jacobiho Symbol,

$$(\tfrac{a}{p}) = \begin{cases} 0 & \text{if a} \equiv 0 \pmod{p} \\ +1 & \text{if a} \not\equiv 0 \pmod{p}, \text{ a pro celá čísla } x, \text{ kde } a \equiv x^2 \pmod{p} \\ -1 & \text{v ostatních případech} \end{cases}$$

který je možné spočítat pro jakékoliv liché číslo v čase $\mathcal{O}((\log n)^2)$. Po nahrazení Legendreova symbolu Jacobiho symbolem vznikne rovnice

$$\tfrac{a^{n-1}}{2} \equiv (\tfrac{a}{n}) \pmod{n}$$

kde n je číslo, u kterého kontrolujeme prvočíselnost. Pokud je n prvočíslo pak rovnost platí pro libovolné hodnoty a. Pokud rovnost platí pro a a n je složené nazývá se a Eulerův lhář pro n. V opačném případě (rovnice neplatí, n je složené) se a nazývá Eulerův svědek pro n. Pokud rovnice platí pro různé hodnoty a je n pravděpodobně prvočíslo.

Algoritmus

Vstupem algoritmu je testované číslo $n > 2$ a počet provedených testů k. Výstupem je informace zdali je číslo složené, nebo prvočíslem s danou pravděpodobností.

$n \leftarrow$ *testované číslo*
$k \leftarrow$ *množství testů*
for $i = 0$ **to** $i \leq k$ **do**
  $a = rand(2, n)$
  $x = Jacobi(\frac{1}{n})$
  if $x = 0$ **or** $a^{\frac{n-1}{2}} \mod n \neq x$ **then**
    return *composite*
  end if
  $i = i + 1$
end for
return *probably prime*

Příklad

Mějme rozhodnout zdali je číslo $n = 221$ prvočíslo. Náhodně zvolíme $a = 47$ a spočteme: $a^{\frac{(n-1)}{2}} \pmod{n} = -1 \pmod{221}$ a pravou stranu $(\frac{a}{n}) \pmod{n} = -1 \pmod{221}$. Obě strany se rovnají, takže číslo n je prvočíslo, nebo číslo a je Eulerův lhář pro n. Pro další náhodné $a = 2$ vyjde levá strana $30 \pmod{221}$ a pravá $-1 \pmod{221}$, takže a je Eulerův svědek a číslo n je složené.

3.5. Miller–Rabinův test prvočíselnosti

Miller–Rabinův test prvočíselnosti je další pravděpodobnostní algoritmus podobný Fermatovu. Původní verze Garry L. Millera byla deterministická ale spoléhala na nedokázané zobecnění Riemannovi hypotézy. Michael O. Rabin algoritmus upravil na nepodmíněný pravděpodobnostní. Složitost algoritmu je $\mathcal{O}(k \log^{2+\epsilon}(n))$ s pravděpodobností chyby 4^{-k}.

Princip

Stejně jako Fermatův a Solovay–Strassenův spoléhá i tento test na množinu rovností, které platí pro prvočísla. Z malé fermatovy věty

$$a^{p-1} - 1 \equiv 0 \pmod{p}$$

kde p a a jsou nesoudělná. Pro $p - 1$ musí platit

$$p - 1 = 2^s \cdot d$$

s a d jsou celá kladná čísla a d je liché. Proto

$$a^{2^s \cdot m} - 1 = 0 \pmod{p}$$

Tento vztah rozepíšeme pomocí vzorce $(A^2 - B^2) = (A - B) \cdot (A + B)$ a vznikne

$$a^{2^s \cdot m} - 1 = (a^{2^{s-1} \cdot m} - 1) \cdot (a^{2^{s-1} \cdot m} + 1) = (a^{2^{s-2} \cdot m} - 1) \cdot (a^{2^{s-2} \cdot m} + 1) \cdot (a^{2^{s-1} \cdot m} + 1)$$
$$= (a^m - 1) \cdot (a^m + 1) \cdot (a^{2m} + 1) \cdots (a^{2^{s-3} \cdot m} + 1) \cdot (a^{2^{s-2} \cdot m} + 1) \cdot (a^{2^{s-1} \cdot m} + 1)$$

Pokud je a prvočíslo tak p musí dělit $a^{p-1} - 1$ a musí tedy dělit jeden z rozepsaný výraz. Tedy zkontrolujeme a pro následující rovnosti.

$$\begin{gathered} a^m \equiv 1 \pmod{p} \\ a^m \equiv -1 \pmod{p} \\ a^{2m} \equiv -1 \pmod{p} \\ \vdots \\ a^{\frac{(p-1)}{p}} \equiv -1 \pmod{p} \end{gathered}$$

Pokud platí alespoň jedna rovnice, p je prvočíslo s pravděpodobností 75%, pokud neplatí žádný výraz, p je složené.

Algoritmus

Vstupem algoritmu je testované číslo $n > 2$ a počet provedených testů k. Výstupem je informace zdali je číslo složené, nebo prvočíslem s danou pravděpodobností.

$n \leftarrow$ *testované číslo*
$k \leftarrow$ *množství testů*
$s, m \leftarrow n - 1 = 2^{s-m}$
for $i = 0$ **to** $i \leq k$ **do**
  $a = rand(2, n)$
  $x = a^s \mod n$
  if $x \neq 1$ **and** $x \neq n - 1$ **then**
    for $j = 0$ **to** $j < s$ **do**
      $x = x^2 \mod n$
      if $x = 1$ **then**
        return *composite*
        if $x = n - 1$ **then**
          $j = s$
        end if
      end for
    end if
    $i = i + 1$
  end for
  return *probably prime*

Příklad

Mějme $n = 21$ z rovnice $n - 1 = 2^s \cdot d$ vypočteme $s = 2$ a $d = 5$ a náhodně vybereme $a = 10$(z intervalu $0 < a < n$). Dále zkontrolujeme rovnice

$$a^{2^0 \cdot d} \pmod n = 10^5 \pmod n = 21 \neq 1$$
$$a^{2^1 \cdot d} \pmod n = 10^{10} \pmod n = 4 \neq n - 1$$

tedy číslo 21 je složené.

3.6. Baillie-PSW test prvočíselnosti

Baillie-PSW pravděpodobnostní test prvočíselnosti je kombinací Miller-Rabinova testu s bází 2 a silného Lucasova pravděpodobnostního testu. Oba zmíněné testy mají svoje vlastní pseudoprvočísla (složená čísla, kterí projdou testem). Baillie-PSW test spoléhá na to, že Fermatova pseudoprvočísla a Lucasova pseudoprvočísla nemají žádné známé překrytí. Test neselže pro žádné složené číslo menší než 2^{64}(test je pro tyto čísla deterministický) a v současné době není známé žádné větší složené číslo, které by bylo testem označeno za prvočíslo. Složitost algoritmu je $\mathcal{O}(\log^3 n)$.

Princip

Na začátku je vhodné zkontrolvoat, zdali není kontrolované p dělitelné prvočísly pod danou malou mez. Dále provedeme kontrolu Miller-Rabbinovým testem s $a = 2$ tedy jestli platí

$$2^d \equiv 1 \pmod n$$
nebo
$$2^{2^r \cdot d} \equiv -1 \pmod n$$

kde $n = 2^s \cdot d + 1$ a $0 \leq r < s$. Dále nalezneme parametry pro silný Lucasův test, D ze sekvence $5, -7, 9, -11, 13, -15, \cdots, (P^2 + 4)$, pro které je Jacobiho smybol $(\frac{D}{n})$ roven -1 a dále nastavíme $P = \sqrt{D - 4}$ a $Q = -1$. Silný Lucasův test musí splňovat jednu z následujících podmínek.

$$U_d \equiv 0 \pmod n$$
nebo
$$V_{2^r \cdot d} \equiv 0 \pmod n$$

kde $2^s \cdot d = n - Jacobi(\frac{D}{n})$ (d musí být liché) a $0 \leq r < s$. Sekvence $U(P, Q)$ pro $Q = -1$ je Fibbonaciho polynom s definicí.

$$U_0(P, Q) = 1$$
$$U_1(P, Q) = 1$$
$$U_n(P, Q) = P \cdot U_{n-1}(P, Q) - Q.U_{n-2}(P, Q) \qquad \text{pro } n > 1$$

Pokud je i $P = 1$ jedná se o Fibbonaciho číslo. Druhá sekvence $V(P, Q)$ se pro $Q = -1$ nazývá Lucasev polynom z následující definice.

$$V_0(P, Q) = 2$$
$$V_1(P, Q) = P$$
$$V_n(P, Q) = P \cdot V_{n-1}(P, Q) - Q.V_{n-2}(P, Q) \qquad \text{pro } n > 1$$

Pro urychlení výpočtu U a V je možné využít následujících rovnic

$$U_{2k} = U_k \cdot V_k$$
$$V_{2k} = V_k^2 - 2Q^k$$
$$U_{2k+1} = \frac{P \cdot U_{2k} + V_{2k}}{2}$$
$$V_{2k+1} = \frac{D \cdot U_{2k} + P \cdot V_{2k}}{2}$$

Algoritmus

Vstupem algoritmu je testované číslo $n > 2$. Výstupem je informace zdali je číslo složené, nebo prvočíslem s danou pravděpodobností.

$n \leftarrow$ *testované číslo*
if $millerRabin(n, 2) = composite$ **then**
  return *composite*
end if
$S = computeD(n)$
$P = 1$
$Q = \frac{1-D}{4}$
$s = computeS(n)$
$d = \frac{n+1}{2^s}$
$U = computeU(d, D, P, Q)$
$V = computeV(d \cdot 2^r, D, P, Q)$
if $U \mod n = 0$ **then**
  return *probably prime*
end if
if $V \mod n = 0$ **then**
  return *probably prime*
end if
return *composite*

Příklad

Příklad silného Lucasova testu pro číslo $n = 23$. Nejdříve zkontrolujeme Miller-Rabinuv test pro číslo $n = 23$ s číslem $a = 2$. Z rovnice $n-1 = 2^s \cdot d$ vypočteme $s = 1$ a $d = 11$. Zkontrolujeme rovnice

$$a^{2^0 \cdot d} \pmod n = 2^{11} \pmod n = 1$$

takže n je pravděpodobně prvočíslo. Dalším krokem je provedení silného Lucasova testu. Vypočteme $D = 5$ a odtud $P = 1$ a $Q = -1$. Dále spočítáme d z rovnice $2^s \cdot d = n - Jacobi(\frac{D}{n})$, odkud $d = 3$, $s = 3$. Zbývá zkontrolovat rovnice Lucasva testu

$$U_d \pmod n = U_3 \pmod n = 2 \neq 0$$
$$V_{2^r \cdot d} \pmod n = V_2^2 \cdot 3 \pmod n = 322 \pmod n = 0$$

pro $0 < r < s$.Číslo 23 je tedy prvočíslem.

3.7. AKS test prvočíselnosti

AKS (Agrawal–Kayal–Saxena) [1] test je deterministický algoritmus dokazující prvočíselnost v polynomickém čase. Algoritmus byl vytvořem v roce 2002 Manindra Agrawalem, Neeraj Kayalem, a Nitin Saxenem. Autoři za svůj výtvor dostali Gödelovu a Fulkersonovu cenu. AKS je první algoritmus dokazující prvočíselnost, který je zaráz obecný, polynomiální, deterministický a nepodmíněný.

- Algoritmus funguje správně pro libovolné celé číslo. Některé algoritmy nefungují správně pro některá čísla (Fermatuv test pro Carmichaleova čísla ...).
- Maximální doba běhu algoritmu je omezena polynomem na počet číslic kontrolovaného čísla. U ECPP nevíme, zdali je běh algoritmu omezen polynomem pro všechyn vstupy.
- Algoritmus rozhodne o daném čísle deterministicky, zda je prvočíslo nebo číslo složené. Na rozdíl od pravděpodobnostních.
- Správnost AKS algoritmu není podmíněna žádnou nedokázanou hypotézou.

Složitost základního algortimu je $\mathcal{O}(\log(n)^{10.5})$ a existují vylepšení až na $\mathcal{O}(\log(n)^6)$

Princip

Číslo $n \geq 2$ je prvočíslo jen když

$$(x-a)^n \equiv (x^n - a) \pmod{n}$$

platí pro všechna a nesoudělná s n (nebo jen pro některá a, zejména pro $a = 1$). x je volná proměnná a nikdy nebude nahrazena číslem. Pro ověření prvočíselnosti AKS pužívá následující rovnost.

$$(x-a)^n \equiv (x^n - a) \pmod{(n, x^r - 1)}$$

která je stejná jako

$$(x-a)^n - (x^n - a) = nf + (x^r - 1)g$$

pro nějaké polynomy f a g. Tato shoda může být zkontrolována v čase polynomicky omezeném na počet číslic n. Tuto rovnici splňují všechna prvočísla (pokud zvolíme $g = 0$ dostaneme první rovnici). Nicméně některá složená čísla tuto rovnici splní také. Důkaz AKS ukazuje, že zde existuje malé r a přijatelně malá množina A taková, že když rovnost platí pro všechna $a \in A$, tak je n prvočíslem.

Algoritmus

Algoritmus testuje prvočíselnsot čísla $n > 1$

1. if $n = a^b$ for $a > 0$, $b > 0$ return composite
2. $r =$ nejmenší r takové že $O_r(n) > (logn)^2$
3. if $1 < gcd(a, n) < n$ for $a \leq r$ return composite
4. if $n \leq r$ return prime
5. for $a = 1$ to $\left\lfloor \sqrt{\varphi(r)} log(n) \right\rfloor$
   if $(x + a)^n \neq X^n + 1 \pmod{X^r - 1, n}$ return composite
6. return prime

4. Porovnání složitostí

Tato kapitola popisuje teoretickou složitost algoritmu a reálnou časovou složitost pro dané vstupy. Program je napsán v jazyce Java a využívá knihovnu BigInteger. Tato knihovna umožňuje teoreticky práci s čísly až do velikosti RAM počítače, nicméně žádná z metody by nespočítala tak velké prvočíslo v přijatelném čase. Knihovná dále obsahuje sadu funkci pro práci s těmito čísly. Knihovna je zmíněna z toho důvodu, že také ovlivňuje rychlost algoritmu. Např. u metody Baillie-PSW při výpočtu sekvencí U a V dochází k umocňování velkých čísel typu BigInteger číslem Integer což má velký dopad na rychlost algoritmu pro větší čísla.

4.1. Teoretická složitost algoritmů

Teoretická složitost algoritus vychází z popisu originálního algoritmu popsaného daným tvůrcem. Některé pravděpodobnostní metody (Monte Carlo) jsou rychlejší na úkor přesnosti odpovědi. Nicméně jejich opakováním je možné dosáhnout slušných výsledků a v praxi se běžně používají.
Číslo k udává počet testů. Dále metoda Eratosthenovo síto vypočítá všechna prvočísla až do velikosti n. Fermatův test selže pro Carmichaleova čísla a u Baillie-PSW není jisté jak funguje pro čísla větši než 2^{64}.

4.2. Reálné složitosti implementovaných algoritmů.

Všechyn metody byly spouštěny na stejném stroji s 4GHz procesorem a 16GB pamětí RAM na operačním systému Windows 10. Při testování jsem záměrně vyhledával pouze prvočísla, protože nalezení složených čísel u většiny metod

Metoda	Typ 2	Složitost
Triviální dělení	deterministická	$\mathcal{O}(2^{\frac{n}{2}})$
Eratosthenovo síto	deterministická	$\mathcal{O}(n \log(\log n))$
Fermatův test	pravděpodobnsotní	$\mathcal{O}(k \log^{2+\epsilon} n)$
Solovay–Strassenův	pravděpodobnostní	$\mathcal{O}(k \log^3 n)$
Miller–Rabinův	pravděpodobnostní	$\mathcal{O}(k \log^{2+\epsilon}(n))$
Baillie-PSW	pravděpodobnostní	$\mathcal{O}(\log^3 n)$
AKS	deterministická	$\mathcal{O}(\log(n)^{10.5})$

Tabulka 1. Teoretická složitost algoritmů.

trvá kratší dobu. Dále může být výkon metod ovlivněn plánovačem, instrukční sadou... Všechny testy ihned kontrolují dělitelnost dvojkou, takže z časového hlediska nemá smysl kontrolovat sudá čísla. Pro měření času byla využita prvočísla blízko čísel $1 \cdot 10^x$.

4.2.1. Triviální dělení

Triviální dělení, jak již název napovídá, je nejjednoduší metoda detekce prvočíselnosti, která používá pouze operaci cyklu, modula a odmocniny (1x). Navzdory předpokladům dokáže i čísla okolo miliardy řešit v přijatelném čase menším jak 10ms.

číslo	potřebný čas
10^6	2ms
10^7	3ms
10^8	5ms
10^9	10ms
10^{10}	21ms
10^{11}	43ms
10^{12}	81ms
10^{13}	220ms
10^{14}	651ms
10^{15}	1.800s
10^{18}	55s

Tabulka 2. Časová složitost triviálního děleni.

Z tabulky lze vypozorovat, že triviální dělení je schopné řešit čísla do 12 míst v přijatelném čase. Pravidelné kontrolování prvočíselčísel s 15 a více místy by se stalo časově velmi náročným.

4.2.2. Eratosthenovo síto

Eratosthenovo síto je rozdílné v tom, že po dané n vypočítá všechna prvočísla. Tím je tedy mnohonásobně náročnejší na paměť a v implementované podobě vypočítá prvočísla do velikosti maximální kladné hodnoty int(2147483647). Vypočítání všech prvočísel do miliardy trvá přibližně 14s, do dvou miliard je to 30s a na větší čísla již nestačí 16GB operační paměti. Pro větší n může dojít k nedostatku přiřazené paměti Jave. Návod na zvětšení HEAP paměti je popsán v sekci Apliakce. Síto slouží pro jednorázové vypočítání a uchování prvočísel takže nemá smysl ho spouštět opakovaně.

4.2.3. Fermatův test

Fermatův test je nejjednoduší pravděpodobnostní metoda detekce prvočíselnosti. V dnešní době je již plně nahrazena.

číslo	**potřebný čas k=1**	potřebný čas k=1000	potřebný čas k=[2,...,n-2]
10^6	1ms	10ms	670ms
10^7	1ms	10ms	6.730s
10^8	1ms	10ms	73s
10^9	1ms	10ms	810s
10^{10}	1ms	14ms	
10^{11}	1ms	14ms	
10^{12}	1ms	15ms	
10^{13}	1ms	16ms	
10^{14}	1ms	16ms	
10^{15}	1ms	17ms	
10^{18}	1ms	17ms	
10^{30}	1ms	23ms	
10^{200}	5ms	608ms	
10^{400}	10ms	4s	

Tabulka 3. Časová složitost Fermatova testu.

Z tabulky plyne, že jeden test je proveden téměř instantně i pro velká n. Při provádění 1000 testů se čas v závislosti na n zvětšuje. Tato změna je způsobena výpočetní silou stroje. Provést test pro všechna k je z časovéhop hlediska nepoužitelné už pro čísla přesahující milion. Nicméně test selže pro Carmichaleova čísla a pokud neprovedeme test pro všechna k nemůžeme si být jisti i u ostatních.

4.2.4. Solovay–Strassen

Metoda vytvořená Robertem M. Solovayem a Volkerem Strassenem je první

použitelnou metodou detekce prvočíselnosti, dnes již nahrazenou metodou Miller–Rabin a Baillie-PSW.

číslo	**potřebný čas k=1**	potřebný čas k=1000	potřebný čas k=[2,...,n-2]
10^6	1ms	19ms	2.369s
10^7	1ms	20ms	25s
10^8	1ms	20ms	299s
10^9	1ms	21ms	1037s
10^{10}	1ms	27ms	
10^{11}	1ms	30ms	
10^{12}	1ms	31ms	
10^{13}	1ms	33ms	
10^{14}	1ms	33ms	
10^{15}	1ms	36ms	
10^{18}	1ms	39ms	
10^{30}	1ms	65ms	
10^{200}	7ms	950ms	
10^{400}	15ms	5.1s	

Tabulka 4. Časová složitost Solovay–Strassenova testu.

Tabulka časové složitosti Solovay–Strassenova testu je podobná časové složitosti testu Fermatova. Potřebný čas pro 1000 testů je přibližně 2x větší což stále činí test použitelným. časová složitost pro všechny testy je přibližně 5x větší, takže testovat všechny hodnoty k je nevhodné už pro čísla okolo milionu. Chybovost metody je 2^{-k}.

4.2.5. Miller–Rabin

Miller-Rabinův test byl prvním testem využívaným k šifrování RSA.
Časová složitost metody je podobná Fermatově a Solovay–Strassenově, nicméně chybovost je menší a to 4^{-k}. Tedy pro $k = 3$ je šance na špatný výsledek 1,5625%. Podobně jako u předešlých metod je test pro všechny k již brzy nepoužitelný.

4.2.6. Baillie-PSW

Metoda Baillie-PSW je složitější než předešlé a nelze u ní ovlivňovat počet testů.
Časová složitost metody Ballie-PSW je závislá na nutnosti vypočítat sekvenci U a V. Při výpočtu těchto sekvencí dochází k umocněňování velkých čísel číslem Integer, což má zásadní vliv na výkonnost metody. Metoda dosahuje špatných časových výsledků už pro čísla okolo 10^7.

číslo	**potřebný čas k=1**	potřebný čas k=1000	potřebný čas k=[2,...,n-2]
10^6	1ms	15ms	932ms
10^7	1ms	15ms	9.2s
10^8	1ms	15ms	100s
10^9	1ms	15ms	1067s
10^{10}	1ms	18ms	
10^{11}	1ms	19ms	
10^{12}	1ms	20ms	
10^{13}	1ms	21ms	
10^{14}	1ms	21ms	
10^{15}	1ms	21ms	
10^{18}	1ms	21ms	
10^{30}	1ms	33ms	
10^{200}	4ms	616ms	
10^{400}	10ms	3.9s	

Tabulka 5. Časová složitost Miller–Rabinova testu.

číslo	**potřebný čas**
10^6	40ms
10^7	420ms
10^8	9s
10^9	252s

Tabulka 6. Časová složitost metody Baillie-PSW.

4.2.7. AKS

AKS je první dokázanou deterministickou metodou.
Z tabulky vyplývá že časová složitost metody AKS je příjemná i pro velmi vysoká n.

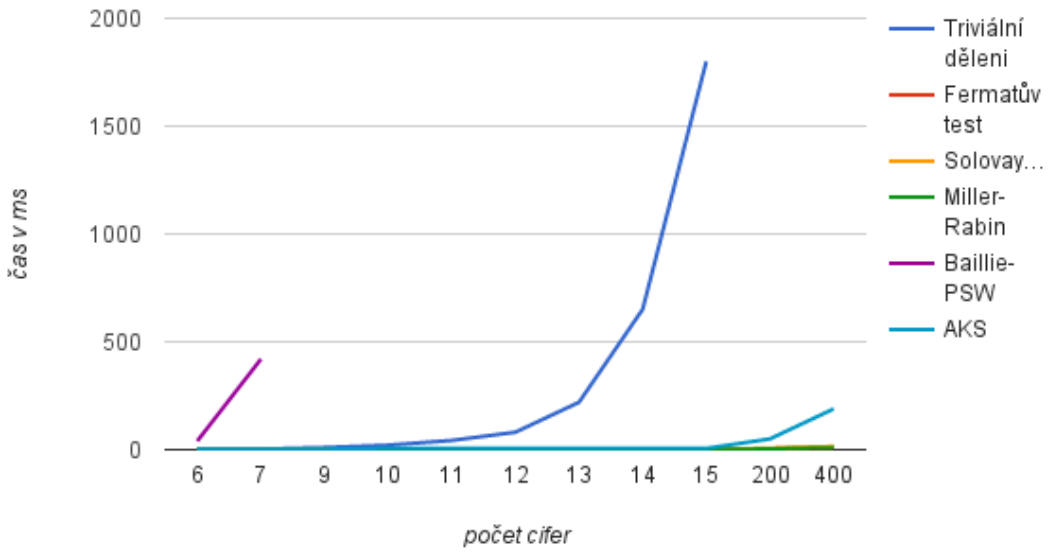
4.3. Porovnání složitostí

Z tabulek lze vypozorovat, že všechny pravděpodobnostní metody kromě Baillie-PSW, která trpí v důsledku složitosti operace umocnování velkých čísel, lze využít i pro čísla s 200 a více číslicemi. Vzhledem k chybovosti metod se nejlépe jeví metoda Miller-Rabin, kde již při 10 testech je výsledek téměř určitě správný (šance na špatný výsledek je $1 \cdot 10^{-6}\%$) a jeden test čísla s 400 číslicemi trval 10ms. Z deterministických metod je nejvhodnější metoda AKS. Volba použití metody by mohla být omezena pouze na deterministickou AKS a pravděpodobnostní Miller-Rabinovou v závislosti na požadované přesnosti, nicméně pár testů Miller-Rabinovou metodou bude rychlejší než metoda AKS

číslo	potřebný čas
10^6	3ms
10^7	3ms
10^8	3ms
10^9	3ms
10^{10}	6ms
10^{11}	6ms
10^{12}	6ms
10^{13}	6ms
10^{14}	6ms
10^{15}	6ms
10^{18}	7ms
10^{30}	10ms
10^{200}	50ms
10^{400}	189ms

Tabulka 7. Časová složitost metody AKS.

a požadovaná přesnost bude dostačující. V grafu je znázorněn potřebný čas pro test prvočíselnosti v závislosti na velikosti vstupu. U pravděpodobnostních metod se počítá s jedním testem, tedy $k = 1$.



Obrázek 1. Graf závislosti vstupu na čase.

5. Program

Aplikaci jsem se rozhodl napsat v programovacím jazyce Java, protože je multiplatformní a má lehce dostupné vývojové prostředí. Stejně jako C# i Java obsahuje využívanou knihovnu BigIteger pro práci s libovolně velkými celými čísly. Mimo klasických funkcí jako sčítání, odčítání, násobení, dělení knihovna umožňuje i další funkce jako modulo, modpow (modulo mocniny), umocňování a další. Jistou nevýhodou knihovny je, že porovnávání dvou čísel se musí provádět funkcí, což činí kód mírně nepřehledným. Výstupem je Java aplikace ve formátu .jar, kterou lze spouštět z konzole.
Program má pro každou metodu svou vlastní třídu a dále jedni třídu řídící. Každá metoda kromě Eratosthenova síta nejdříve zkontroluje, zda je číslo kladné, rovno dvěma, a dále je-li liché.

5.1. Seznam velkých prvočísel

počet míst	**prvočíslo**
10^6	1 000 003
10^7	10 000 019
10^8	100 000 007
10^9	1 000 000 007
10^{10}	10 000 000 019
10^{11}	100 000 000 003
10^{12}	1 000 000 000 039
10^{13}	10 000 000 000 037
10^{14}	100 000 000 000 031
10^{15}	1 000 000 000 000 037
10^{18}	1 000 000 000 000 000 003
10^{30}	1 000 000 000 000 000 000 000 000 000 057
10^{200}	$10^{200} + 357$
10^{400}	$10^{400} + 69$

Tabulka 8. Seznam vybraných prvočísel.

5.2. Ovládání aplikace

Program je v balíčku .jar a v konzoli operačního systému Windows se spouští příkazem **java -jar "soubor s příponou .jar"**. Pro správnou funkčnost je nutné mít nainstalovanou a aktualizovanou javu. Program se spouští s jedním až třemi parametry dle následující tabulky (**java -jar "soubor s příponou .jar"parametr1 parametr2 parametr3**).

metoda	parametr1	parametr2	parametr3
Triviální dělení	t	testované číslo	
Eratosthenovo síto	e	testované číslo	
Fermatův test	f	testované číslo	počet testů
Solovay–Strassen	s	testované číslo	počet testů
Miller-Rabin	m	testované číslo	počet testů
Baillie-PSW	b	testované číslo	
AKS	a	testované číslo	

Tabulka 9. Seznam parametrů programu.

Např. **java -jar DPP.jar m 12345 5** Pokud není zadán parametr3 u metod, které ho umožňují provede se test pro všechny možné hodnoty, stejně jako pokud je parametr3 větší než testované číslo. jestliže je parametr3 záporný, vezme se jeho absolutní hodnota. Dalším omezením je testování pomocí Eratosthenova síta. Maximální možná testovaná hodnota je 2147483646 (Integer.MAX_VALUE -1). Dále je možné, že test vyvolá chybu nedostatku heap paměti. Paměť jě možné navýšit parametrem javy **-Xmx** tedy **java -jar -Xmx2056m "soubor s příponou .jar"parametr1 parametr2 parametr3**.

5.3. Triviální dělení

Jednou z nejjednoduších metod na programování je triviální dělení. Jedinou nutnou věcí bylo doprogramovat odmocninu čísla BigInteger.

```
public boolean test(BigInteger n) {
    BigInteger sqrt = bigIntegerSqrt(n);
    for (BigInteger i = BigInteger.valueOf(2);
         i.compareTo(sqrt) == -1;
         i = i.add(BigInteger.valueOf(1))) {

        if (n.mod(i).compareTo(BigInteger.valueOf(0))==0){
            return false;
        }
    }
    return true;
}

public BigInteger bigIntegerSqrt(BigInteger a) {
    if (a.compareTo(BigInteger.valueOf(0)) == 0) {
        return BigInteger.valueOf(0);
    }
    BigInteger n1 = (a.shiftRight(1)).add(BigInteger.valueOf(1));
    BigInteger n2 = (n1.add(a.divide(n1))).shiftRight(1);

    while (n2.compareTo(n1) == -1) {
        n1 = n2;
        n2 = (n1.add(a.divide(n1))).shiftRight(1);
    }

    if (a == a.pow(2)) { return n1;
    } else { return n1.add(BigInteger.valueOf(1));}
}
```

5.4. Eratosthenovo síto

Podobně jako triviální dělení je Eratosthenovo síto velmi jednoduchou metodou detekce prvočíselnosti. Nevýhodou je nutnost uchovávát všechna prvočísla v paměti, takže může dojít k přetečení maximální povolené paměti. Při testování prvočísla pomocí síta se vypočítají všechna prvočísla do n a následně se zkontroluje poslední. Tato metoda není vhodná pro kontrolování jednoho čísla. Je zde spíše z důvodu kontroly rychlosti.

```
public boolean[] test(int n) {
    boolean[] siev = new boolean[n];
    Arrays.fill(siev, Boolean.TRUE);

    siev[0] = false;
    siev[1] = false;

    for (int i = 2; i < n; i++) {
        if (siev[i] != false) {
            siev[i] = true;
            int x = i + i;
            while (x < n && x > 0) {
                siev[x] = false;
                x += i;
            }
        }
    }
    return siev;
}
```

5.5. Fermatův test

U Fermatova testu je podobně jako u některých ostatních pravděpodobnostních metod možné ovlivňovat počet testů. V této metodě se náhodně generuje a takové, že $1 < a < n$ viz 3.3.. Nevýhodou může být, že se a generuje náhodně, tak, že se teoreticky může vícekrát vygenerovat to samé. Nicméně pokud máme číslo s 400 číslicemi a náhodně vygenerujeme 10 čísel pro deset testů, šance, že se vygenerovaná a překryjí je minimální. Generátor náhodných čísel o velikosti n (BigInteger) je zde jedinou složitější věcí.

```
private boolean test(BigInteger n, BigInteger a) {
    if (a.modPow(n.subtract(BigInteger.valueOf(1)), n).compareTo(BigInteger.valueOf(1)) != 0) {
        return false;
    }
    return true;
}
private BigInteger randomBigInteger(BigInteger n) {
    Random rnd = new Random();
    int maxNumBitLength = n.bitLength();
    BigInteger aRandomBigInt;
    do {
        aRandomBigInt = new BigInteger(maxNumBitLength, rnd);
    } while (aRandomBigInt.compareTo(n) > 0);
    return aRandomBigInt;
}
```

5.6. Solovay–Strassen

Solovay–Strassen je další pravděpodobnostní metoda s možností volby počtu testů, a stejně jako u předchozí se vypočítává náhodně a. Dále je u tohoto testu nutné pro dané n a a vypočítat Jacobiho symbol.

```
private boolean test(BigInteger n, BigInteger a) {
    BigInteger t1 = a.modPow(n.subtract(BigInteger.valueOf(1)).divide(BigInteger.valueOf(2)), n);
    int t2 = bigIntegerJacobi(n, a);

    if (t1.compareTo(n.subtract(BigInteger.valueOf(1)))==0){
        t1 = BigInteger.valueOf(-1);
    }
    if (t2 == 0 || t1.compareTo(BigInteger.valueOf(t2)) != 0){
        return false;
    }
    return true;
}
private int bigIntegerJacobi(BigInteger n, BigInteger a) {
    int j = 1;
    while (a.compareTo(BigInteger.valueOf(0)) != 0) {
        while (a.mod(BigInteger.valueOf(2)).compareTo(BigInteger.valueOf(0)) == 0) {
            a = a.divide(BigInteger.valueOf(2));

            if (n.mod(BigInteger.valueOf(8)).compareTo(BigInteger.valueOf(3)) == 0
                    || n.mod(BigInteger.valueOf(8)).compareTo(BigInteger.valueOf(5)) == 0) {
                j = -j;
            }
        }
        BigInteger c = n;
        n = a;
        a = c;
        if (a.mod(BigInteger.valueOf(4)).compareTo(BigInteger.valueOf(3)) == 0
                && n.mod(BigInteger.valueOf(4)).compareTo(BigInteger.valueOf(3)) == 0) {
            j = -j;
        }
        a = a.mod(n);
    }
    if (n.compareTo(BigInteger.valueOf(1)) == 0) {
        return j;
    } else {
        return 0;
    }
}
```

5.7. Miller-Rabin

Miller-Rabin je, stejně jako dvě předešlé, pravděpodobnostní metoda, která umožňuje měnit počet testů. Jedinou další částí výpočtu je výpočet s z 3.5..

```
private boolean realTest(BigInteger n, BigInteger x) {

    int s = computeS(n);
    BigInteger d = n.subtract(BigInteger.valueOf(1)).divide(BigInteger.valueOf(2).pow(s));
    int jump = 0;

    BigInteger t1 = x.modPow(d, n);
    if (t1.compareTo(BigInteger.valueOf(1)) != 0 && t1.compareTo(n.subtract(BigInteger.valueOf(1))) != 0) {
        for (int i = 0; i < s; i++) {
            jump = 0;
            t1 = t1.modPow(BigInteger.valueOf(2), n);
            if (t1.compareTo(BigInteger.valueOf(1)) == 0) {
                return false;
            }
            if (t1.compareTo(n.subtract(BigInteger.valueOf(1))) == 0) {
                jump = 1;
                break;
            }
        }
        if (jump != 1) {
            return false;
        }
    }
    return true;
}

private int computeS(BigInteger n) {
    int s = 0;
    while (n.subtract(BigInteger.valueOf(1)).mod(BigInteger.valueOf(2).pow(s)).compareTo(BigInteger.valueOf(0)) == 0)
        {
        s++;
    }
    return s - 1;
}
```

5.8. Baillie-PSW

Z důvodu nutnosti výpočtu sekvencí U a V (3.6.) je Bailliw-PSW test nejsložitější se všech pravděpodobnostních.
Výpočet D ze sekvence $5, -7, 9, -11 \ldots$, pro které je Jacobiho p, D symbol roven -1:

```
private int computeD(BigInteger p) {
    int D = 5;
    while (true) {
        if (bigIntegerJacobi(BigInteger.valueOf(D), p) == -1) {
            break;
        }
        if (D < 0) {
            D = (D - 2) * -1;           //sekvence 5,-7,9,-11,\dots
        } else {
            D = (D + 2) * -1;
        }
    }
    return D;
}
```

Výpočet s pro $p + 1 = 2^s \cdot d$:

```
public int computeS(BigInteger p) {
    BigInteger factor = p.add(BigInteger.valueOf(1));
    int s = 0;
    int exp = 0;
    do {
        s++;
        exp = (int) Math.pow(2, s);
    } while (factor.mod(BigInteger.valueOf(exp)).compareTo(BigInteger.ZERO) == 0);
    return s - 1;
}
```

Další částí výpočtu je výpočet sekvencí U a V. Sekvence jsem zkoušel počítat různými způsoby a následující se jevila nejrychlejší. Tato část výpočtu zabere největší množství času.

```
public ArrayList<BigInteger> computeUV(BigInteger k, int P, int Q, int D) {
    ArrayList<BigInteger> list = new ArrayList<BigInteger>();
    ArrayList<BigInteger> temp = new ArrayList<BigInteger>();
    //n=0
    if(k.compareTo(BigInteger.valueOf(0))==0){
        list.add(BigInteger.valueOf(0));
        list.add(BigInteger.valueOf(2));
        return list;
    }//n=1
    if(k.compareTo(BigInteger.valueOf(1))==0){
        list.add(BigInteger.valueOf(1));
        list.add(BigInteger.valueOf(P));
        return list;
    }
    if (k.mod(BigInteger.valueOf(2)).compareTo(BigInteger.valueOf(0)) == 0) {

        BigInteger newk = k.divide(BigInteger.valueOf(2));
        temp = computeUV(newk,P, Q, D);

        BigInteger U = temp.get(0).multiply(temp.get(1));
        BigInteger V =
                temp.get(1).pow(2).subtract(BigInteger.valueOf(Q).pow(newk.intValue()).multiply(BigInteger.valueOf(2)));

        list.add(U);
        list.add(V);
        return list;
    } else {
        temp = computeUV(k.subtract(BigInteger.valueOf(1)),P, Q, D);

        BigInteger U = ((temp.get(0).multiply(BigInteger.valueOf(P))).add(temp.get(1))).divide(BigInteger.valueOf(2));
        BigInteger V =
                ((temp.get(0).multiply(BigInteger.valueOf(D))).add(temp.get(1).multiply(BigInteger.valueOf(P)))).divide(BigInteger.valueOf(2));

        list.add(U);
        list.add(V);
        return list;
    }
}
```

Sloučením předchozích částí vznikne celý test:

```
public boolean test(BigInteger n) {

    MillerRabin mr = new MillerRabin();
    if (!mr.testForTwo(n)) {
        return false;
    }
    int D = computeD(n);
    int P = 1;
    int Q = (1 - D) / 4;
    int s = computeS(n);
    BigInteger d = n.add(BigInteger.valueOf(1)).divide(BigInteger.valueOf((int)Math.pow(2, s)));

    ArrayList<BigInteger> list = computeUV(d, P, Q, D);
    BigInteger Ud = list.get(0);
    Ud = Ud.mod(n);
    if (Ud.compareTo(BigInteger.valueOf(0)) == 0) {
        return true;
    }
    BigInteger newd = d.multiply(BigInteger.valueOf((int)Math.pow(2, s-1)));
    list = computeUV(newd, P, Q, D);
    BigInteger Vd = list.get(1);
    Vd = Vd.mod(n);

    if (Vd.compareTo(BigInteger.valueOf(0)) == 0) {
        return true;
    }
    return false;
}
```

5.9. AKS

Poslední a nejsložitější implementovanou metodou je metoda AKS 3.7.. Prvním krokem výpočtu je bod 1. z 3.7.:

```
private boolean isPower(BigInteger n) {
    int log = (int) bigIntegerLog(n);
    for (int i = 2; i < log; i++) {
        if (isPowerOf(n, i)) {
            return true;
        }
    }
    return false;
}

private boolean isPowerOf(BigInteger n, int i) {
    int len = (int) Math.ceil(bigIntegerDigits(n) / i);
    BigInteger mid, res;

    BigInteger low = BigInteger.valueOf(10).pow(len - 1);
    BigInteger high = BigInteger.valueOf(10).pow(len).subtract(BigInteger.valueOf(1));

    while (low.compareTo(high) != 1) {
        mid = low.add(high);
        mid = mid.divide(BigInteger.valueOf(2));
        res = mid.pow(i);

        if (res.compareTo(n) < 0) {
            low = mid.add(BigInteger.valueOf(1));
        } else if (res.compareTo(n) > 0) {
            high = mid.subtract(BigInteger.valueOf(1));
        } else {
            return true;
        }
    }
    return false;
}
```

Následuje krok 2. výpočet r. Pří výpočtu r je nutné mít vypočteno menší prvočíslo. Třída si při svém vytvoření pomocí Eratosthenova síta uloží prvních 100 000 prvočísel. Pokud je hledané prvočíslo větší, bude se rekurzivně hledat pomocí AKS.:

```
private BigInteger computeR(BigInteger n) {
    int q, o, tm;
```

```
    BigInteger t;
    BigInteger r = BigInteger.valueOf(2);
    int tr = r.intValue();

    while (r.compareTo(n) < 0) {
        if (r.gcd(n).compareTo(BigInteger.valueOf(1)) != 0) {
            return BigInteger.valueOf(0);
        }
        tr = r.intValue();

        boolean te = false;
        if(tr>size){
            te = test(BigInteger.valueOf(tr));
        } else {
            te = siev[tr];
        }

        if (te == true) {
            q = largestFactor(tr - 1);
            o = (int) (tr - 1) / q;
            tm = (int) (4 * (Math.sqrt(tr)) * 1);
            t = mPower(n, BigInteger.valueOf(o), r);
            if ((q >= tm) && (t.compareTo(BigInteger.valueOf(1)) != 0)) {
                break;
            }
        }
        r = r.add(BigInteger.valueOf(1));
    }
    return r;
}
```

Krok 3.

```
private BigInteger mPower(BigInteger a, BigInteger b, BigInteger n) {
    BigInteger res = BigInteger.valueOf(1);
    while (b.compareTo(BigInteger.valueOf(0)) > 0) {
        while (b.mod(BigInteger.valueOf(2)).compareTo(BigInteger.valueOf(0)) == 0) {
            b = b.divide(BigInteger.valueOf(2));
            a = a.multiply(a).mod(n);
        }
        b = b.subtract(BigInteger.valueOf(1));
        res = res.multiply(a).mod(n);
    }
    return res;
}
```

Celý test:

```
public boolean test(BigInteger n) {
    int log = (int) bigIntegerLog(n);
    if (isPower(n)) {
        return false;
    }
    BigInteger r = computeR(n);
    int tr = r.intValue();
    if(tr == 0){
        return false;
    }
    BigInteger fm = mPower(BigInteger.valueOf(2), r, n).subtract(BigInteger.valueOf(1));
    int up = (int) (2* Math.sqrt(tr)*1);

    for(int i=1; i<up; i++){
        BigInteger bi = BigInteger.valueOf(i);
        BigInteger le = mPower(BigInteger.valueOf(2).subtract(bi), n, n).mod(n);
        BigInteger ri = mPower(BigInteger.valueOf(2), n, n).subtract(bi).mod(n);

        if(le.compareTo(ri)!=0) return false;
    }
    return true;
}
```

6. Užití prvočísel

Jedním z největších přínosů velkých prvočísel je jejich využití v asynchronním šifrování. Např. metody RSA a Diffie–Hellman jsou založeny na problému rozkladu čísla v prvočísla. Pokud máme dvě velká prvočísla je jednoduché je

vynásobit, ale v současnosti je prakticky nemožné tento součin rozložit zpět na součin prvočísel. Nevýhodou těchto metod by mohlo být využití tzv. kvantových počítačů (v současnosti nejsou dost dokonalé), pro které existuje algoritmus umožňující rozklad velkých čísel v kubickém čase.

6.1. Asynchronní kódování RSA

Algoritmus RSA pro asynchronní šifrování byl sestrojen vědci Ron Rivestem, Adi Shamirem, a Leonardem Adlemanem. Vytvoření metody RSA předcházelo velké množství pokusů různých jednostranných funkcí jako problém batohum, permutace polynomu a další. Nicméně až vynásobení dvou čísel bylo dostatečně jednoduché, a přitom faktorizace tohoto čísla byla při použití velkých vstupů nemožná v přijatelném čase. Bezpečnost metody závisí na velikosti použitých prvočísel. Metoda RSA využívá klíče o délce 1024. . . 4096 a největší prolomený klíč (v roce 2010 měl 768 bitů). Předpokládá se, že klíče o délce 1024 může vybavený útočník prolomit už nyní, nebo v brzké budoucnosti.

Princip

Generování klíče:

1. Nalezení dvou velkých prvočísel p a q. Nalezení prvočísla se provádí výběrem náhodného lichého čísla a následné zkontrolování prvočíselnosti. Tento proces se opakuje dokud nejsou nalezena dvě požadovaná prvočísla. Pro složitější rozklad by měla prvočísla být podobně veliká ale neměla by být stejně dlouhá.

2. Výpočet $n = pq$.

3. Výpočet Eulerovy funcke $\varphi(n) = (p-1)(q-1)$. Od této chvíle již p a q nepotřebujeme a mohou být zapomenuty.

4. Výběr čísla e takového, že $e < \varphi(n)$ a $gcd(e, \varphi(n)) = 1$.

5. Nalezení d takového, že $de \equiv 1(\mod \varphi(n))$ nebo také $d \equiv e^{-1}(\mod \varphi(n))$.

Nyní je veřejný klíč kombinace (e, n). Jakákoliv správa, zakódovaná pomocí veřejného klíče, může být dekódována pouze pomocí soukromého klíče, tvořeného kombinací (d, n).

Distribuce klíče:
Příjemce budoucích zpráv pošle veřejný klíč ke svému privátnímu klíči otevřeně budoucímu odesilateli.

Kódování zprávy:

Odesilatel převede zprávu M na celé číslo m takové že $0 \leq m < n$ a $gcd(m, n) \equiv 1$. Takovouto zprávu zakóduje do $c = m^e(\mod n)$ a takto zakódovanou zprávu c odešle příjemci.

Dekódování zprávy:
Příjemce dekóduje zprávu m z c, pomocí svého privátního klíče, funkcí $c^d \equiv m^{e^d} \equiv m(\mod n)$

Příklad

1. Vybereme dvě různá prvočísla $p = 73$ a $q = 79$.
2. Vypočteme $n = pq = 73 \cdot 79 = 5767$.
3. Výpočet Eulerovy funkce $\varphi(5767) = (p-1)(q-1) = (73-1) \cdot (79-1) = 5616$.
4. Vybereme e nesoudělné s 5616, např. $e = 29$.
5. Vypočteme d z rovnice $de = 1(\varphi(n))$, např. $d = 581$.

Nyní máme veřejný RSA klíč (29, 5767) a soukromý (581, 5767). Šifrovací funkce je nyní $c = m^{29} \mod 5767$ a dešifrovací $m = c^{581} \mod 5767$.

- Zprávu $m = 60$ zašifrujeme jako $c = 60^{29} \mod 5767 = 2982$.
- Dešifrujeme $c = 2982$ pomocí $m = 2982^{581} \mod 5767 = 60$.

Pokud chceme šifrovanou komunikaci vést oběma směry je nutné aby oba účastníci vytvořili svuj vlastní soukromý a veřejný klíč a veřejné si vyměnili.

Slabiny

- Metoda je zranitelná pomocí útoku MITM(Man in the middle), kdy se útočníkovy podaří připojit mezi účastníky šifrované komunikace a při odesílání veřejného klíče tento odchytí a nahradí ho svým vlastním. Tímto mu bude schopen číst všechny zprávy odesílané daným příjemcem.
- Možnost nalezení privátního klíče v paměti počítače.
- Možnost odhadnutí vygenerovaných prvočísel na základě kompletní znalosti daného počítače.
- Násobek dvou zašifrovaných zpráv je roven zašifrování násobku dvou původních textů. Ze znalostí odeslaných zpráv můžeme detekovat některé nové.

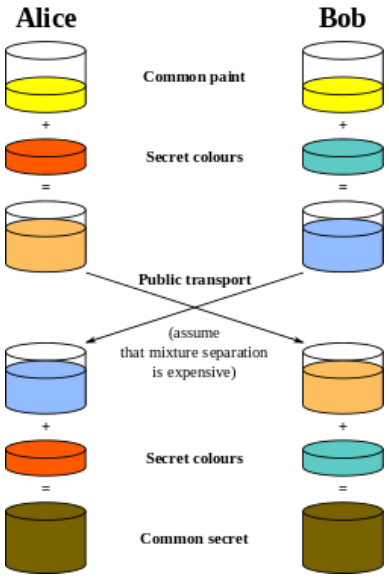
- Stejná zpráva je pomocí stejného veřejného klíče vždy převedena na stejnou šifru. Proto je pro útočníka možné zkoušet šifrovat různé odhadované zprávy pomocí stejného klíče a kontrolovat, zda se výsledek rovná odchycené zprávě. Těmto útokům se předchází doplněním m o náhodné číslo. Bezpečností RSA se zabývá standard PKCS.

6.2. Diffieho-Hellmanova výměna klíče

Diffieho-Hellmanova(DH) [9] výměna klíče je jednou z prvních metod pro bezpečnou výměnu klíče po veřejném kanálu. Před využitím této metody se klíč běžně dostával k odesilateli jinou bezpečnou metodou (kurýr, osobně, ...). Při použití metody DH není nutné aby se oba účastníci komunikace znali.

Princip

DH výměna klíče vytvoří sdílené tajemství mezi dvěma účastníky komunikace, které se využívá pro zabezpečení komunikace a dat na veřejné síti. Na začátku máme dvě strany (Alice, Bob). Alice a Bob se domluví na nějaké společné startovní barvě např. žlutá. Dále si každý z nich vybere jednu svou tajnou barvu Alice červenou a Bob tyrkysovou. Nyní Alice a Bob smíchají svoji tajnou barvu se žlutou a vznikne jim oranžová a modrá. Poté si vymění své výsledné barvy a do obdržené přidají svoji, tak, že oba mají barvu smíchanou ze své, cizí a veřejné (hnědá). Jak je vidět na obrázku 2.



Obrázek 2. Ilustrace generování sdíleného tajemství metodou Diffie-Hellman [9].

Pro funkčnost metody je nutné aby bylo velmi složité barvy rozdělit do původních (aby nebylo možné oranžovou rozdělit na žlutou, která je veřejná a tajnou

červenou). V praxi se místo barev používají velká čísla a místo míchání funkce multiplikativní grupy modulo p, kde p je prvočíslo.

Příklad

1. Alice a Bob se dohodnou na modulu $p = 19$ a základu $q = 7$ $(0 < q < p)$.
2. Alice si vybere tajný klíč $a = 4$ a pošle Bobovi $A = q^a \mod p = 7^4 \mod 19 = 7$.
3. Bob si vybere tajný klíč $b = 9$ a pošle Bobovi $B = q^b \mod p = 7^9 \mod 19 = 1$.
4. Alice spočítá $s = B^a \mod p = 1^4 \mod 19 = 1$.
5. Bob spočítá $s = A^b \mod p = 7^9 \mod 19 = 1$.
6. Alice a Bob nyní mají sdílené tajemství $s = 1$.

Alice a Bob dosáhli stejného sdíleného tajemství protože $(q^a \mod p)^b \mod p = (q^b \mod p)^a \mod p$. Pouze a,b a $q^{ab} \mod p = q^{ba} \mod p$ jsou drženy v tajnosti. Všechna ostatní čísla a části výpočtu mohou být posílaná přes veřejný kanál. V praxi se používají velká prvočísla, která se hledají stejně jako v metodě RSA náhodným výběrem a kontrolou.

Slabiny

- Podobně jako RSA se čísla a a b generují pseudonáhodně, takže můžou být s potřebnou znalostí odhadnuta.
- Tajemství může být nalezeno v paměti počítače.
- Původní verze neobsahuje autentizaci, takže je možné provést stejně jako v RSA útok typu MITM.
- Pokud nebude použito dostatečně velké p může dobře vybavený útočník prolomit šifru hrubou silou.

Metody RSA a DH jsou časově mnohem náročnější než šifry symetrické a proto se používají zpravidla pouze pro navázání komunikace a předání klíče pro symetrické šifrování následné komunikace.

6.3. Ostatní výskyt

Prvočísla se dále dají využít pro generování polygonů pomocí Fermatových prvočísel, v Sylowově větě, nebo v Lagrangeově větě.

Závěr

Práce kromě popisu prvočísel obsahuje popis vybraných metod a jejich detekce od nejjednodušších, přes pravděpodobnostní až po deterministické jako je AKS. Všechny popsané metody byly implementovány a jejich časová složitost byla vyzkoušena pro různě velké vstupy. Testování časové složitosti ukázalo že nejvýhodnější metodou s poměrem rychlost/přesnost je meotda Miller–Rabin. Mírným zklamáním je metoda Baillie-PSW, která dosahovala horší výsledky jako AKS. Práce dále obsahuje návod na sestrojení algoritmů a využití prvočísel v praxi.

Conclusions

Work contains definition of prime numbers, description of selected primality detection methods from the easiest through probabilistic to deterministic like AKS. All described method was implemented and their time complexity was tested for different size of input. Testing shows that best method by fast/accuracy is Miller-Rabins test. Little disappointment was Baillie-PSW method, which has worse results than ASK. Work also includes instruction for algorithm construction and usage of primes.

Reference

[1] Agrawal, M.; Kayal, N.; Saxena, N. *Primes is in P, Annals of Math.* (2002)

[2] Carvalho, Alexandra *AKS Primality Algorithm* (2003)

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford *Introduction to Algorithms* 3. vydání (2009)

[4] Hort D., Rachůnek J. Algebra 1 (2003)

[5] Pomerance, Carl; Selfridge, John L.; Wagstaff, Samuel S. Jr. *The pseudoprimes to 25·*$10^9 Mathematicsof Computation$35(1980)

[6] Goldwasser, Shafi; Bellare, Mihir *Lecture Notes on Cryptography* (2001)

[7] Rivest, R.L.; Shamir, A.; Adleman, L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* Communications of the ACM. 21 (1978)

[8] Solovay, Robert M.; Strassen, Volker *A fast Monte-Carlo test for primality* SIAM Journal on Computing. 6 (1977)

[9] Vinck, A.J. Han *Introduction to public key cryptography* (2012)

A. Obsah přiloženého DVD

Popis struktury přiloženého dvd.

bin/
: Obsahuje spustiltený program DPP.JAR.

doc/
: Obsahuje text diplomové práce a všechny potřebné soubory pro jeho vytvoření.

src/
: Obsahuje zdrojové texty programu..

readme.txt
: Instrukce pro spouštění programu.