Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

MATEUSZ NIEZABITOWSKI

ALGORYTMY FAKTORYZACJI W ZASTOSOWANIACH KRYPTOGRAFICZNYCH

PROMOTOR:
dr inż. Paweł Topa

Kraków 2012

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

. . . . . .

PODPIS

Abstract

Rozkład na czynniki pierwsze liczb całkowitych mimo pozornej prostoty do dziś pozostaje jednym z ważniejszych zagadnień obliczeniowych. Problem ten ma wieloraki wpływ na postać dzisiejszej informatyki, poczynając od czysto teoretycznych zagadnień związanych z teorią obliczeń i złożoności obliczeniowej, a kończąc na całkowicie praktycznym podejściu związanym z nowoczesnymi systemami kryptograficznymi. W ramach różnorakich badań co bardziej wydajne algorytmy faktoryzacji implementuje się na kolejnych maszynach. Poza oczywistą motywacją związaną z poszukiwaniem coraz to wydajniejszych procesorów, które umożliwią bicie rekordów w rozkładzie na czynniki dużych liczb, powstaje pewien efekt uboczny. Duża ilość działań arytmetycznych na liczbach całkowitych, która charakteryzuje algorytmy faktoryzacji jest dobrym testem mocy jednostek obliczeniowych. Dlatego implementacja tego typu stanowi świetny benchmark procesora urządzenia.

Jedną z platform, która do tej pory nie doczekała się takiej implementacji są telefony komórkowe. Jeszcze dekadę temu użycie urządzenia tego typu jako bazy dla algorytmów tak skomplikowanych, jak *QS* czy *GNFS* mogło wydawać się śmieszne. Jednak w dzisiejszych czasach przemysł telefoniczny rozwija się w takim tempie, że większość ludzi nosi w kieszeniach mini-komputery o mocy obliczeniowej większej niż te, które umożliwiły misji Apollo 11 lądowanie na księżycu w 1969 roku. Zwraca też uwagę fakt, że dostępna kilka lat temu specyfikacja topowych komputerów osobistych charakteryzuje dzisiaj telefony komórkowe, a przestrzeń pomiędzy możliwościami tych dwóch typów urządzeń gwałtownie się kurczy.

Celem tej pracy jest implementacja algorytmu Sita Kwadratowego (ang. *QS - Quadratic Sieve*) na urządzenia mobilne działające pod kontrolą systemu operacyjnego Android. Wykonanie tego zadnia, poza wartością merytoryczną, zaowocuje uzyskaniem ciekawego benchmarku obliczeniowego. Umożliwi on oczywiście testowanie konkretnych modeli telefonów, ale nie tylko: otrzymamy unikalną możliwości porównania mocy procesorów obecnych w dzisiejszych telefonach z tymi obecnymi w komputerach osobistych ostatnich lat.

Spis treści

1. Faktoryzacja liczb całkowitych

1.1. Charakterystyka problemu faktoryzacji

1.1.1. Zarys historyczny

Faktoryzacja (czy też inaczej: rozkład na czynniki pierwsze) liczby całkowitej jest problemem obecnym w matematyce od czasów starożytności. Definicje liczby pierwszej podał w swoim dziele *Elementy* już Euklides około 300 r. p.n.e, a powszechnie uważa się, że pojęcie to było znane jeszcze wcześniej w czasach Pitagorasa (około 500 r. p.n.e). Na deterministyczny algorytm rozwiązujący problem faktoryzacji trzeba było jednak czekać do XII w. Był to opracowany przez Fibonacciego *Algorytm Naiwny* (ang. *Trial Division Algorithm*). Mimo, że formalnie działa on dla dowolnych danych wejściowych, dla liczb, które nie posiadają małych czynników pierwszych (rzędu logarytmu faktoryzowanej liczby) jego użycie staje się niepraktyczne, gdyż czas potrzebny do zakończenia obliczeń stanowczo przekracza wszelkie sensowne granice [1].

Przez wiele lat temat faktoryzacji był lekko zaniedbany. Powszechna była opinia, że problem jest formalnie rozwiązany, nie ma więc powodów aby tracić na niego więcej czasu. Nie było też motywacji potrzebnej do rozwoju - nikt nie miał potrzeby aby rozkładać na czynniki pierwsze duże liczby całkowite. Mimo to, w XVII w. Pierre de Fermat opracował zupełnie nowy sposób faktoryzacji. Mimo, że algorytm (zwany dziś *Algorytmem Fermata*) w większości przypadków działał gorzej niż algorytm naiwny, jego podstawy umożliwiły w przyszłości stworzenie najszybszych znanych dzisiaj rozwiązań. Całkowicie nowy schemat faktoryzacji zaproponował również w XVIII w. Leonhard Euler, jednak jego algorytm działał tylko dla wybranych przypadków, a więc nie zyskał powszechnego rozgłosu [1].

Na prawdziwy postęp w dziedzinie faktoryzacji trzeba było czekać do XX w. Powstanie nowoczesnych metod kryptograficznych opierających się na trudności problemu rozkładu na czynniki pierwsze sprawiło, że szukanie nowszych i szybszych rozwiązań stało się koniecznością. Powstawały coraz to lepsze algorytmy. Część z nich była całkowicie innowacyjna w założeniach, większość jednak opierała się na metodzie którą opublikował na początku XX w. Maurice Kraitchik, a która z kolei bazowała na algorytmie Fermata. Do tej drugiej grupy należą, między innymi, algorytm *CFRAC*, algorytm *QS*, a także algorytm *GNFS*, z których każdy w swoim czasie był rekordzistą na polu faktoryzacji. [1, 2].

1.1.2. Definicja problemu

Faktoryzacja sama w sobie jest terminem bardzo ogólnym. Termin ten oznacza rozkład obiektu na czynniki będące od tegoż obiektu w pewnym sensie prostsze. Definicja tej prostoty zależy oczywiście od dziedziny. Przedmiotem faktoryzacji może być na przykład liczba całkowita, macierz lub wielomian. W ramach tej pracy jednak interesuje nas jedynie faktoryzacja liczb całkowitych.

Powiemy, że liczba całkowita $p > 1$ jest **liczbą pierwszą**, gdy zachodzi

$$\forall_{n \in \mathbb{N}^+} : n \mid p \Rightarrow (n = 1 \vee n = p) \tag{1.1}$$

czyli, że jedynymi dzielnikami p są 1 i ona sama (tak zwane **dzielniki trywialne**). Powiemy, że liczba c jest **liczbą złożoną**, gdy nie jest ona liczbą pierwszą (czyli, gdy c posiada co najmniej jeden nietrywialny dzielnik). Podstawowe twierdzenie arytmetyki mówi nam że:

$$\forall_{n \in \mathbb{N}, n>1} \exists_{p_i, \alpha_i} : n = \prod_i p_i^{\alpha_i} \tag{1.2}$$

gdzie każda liczba z ciągu p_i jest liczbą pierwszą, a każda liczba z ciągu α_i jest naturalnym wykładnikiem (oczywiście, ciągi te muszą być skończone). Iloczyn po prawej stronie znaku równania (1.2) nazywa się **postacią kanoniczną** liczby n, a proces uzyskiwania postaci kanonicznej danego n nazywa się **rozkładem** n **na czynniki pierwsze** lub **faktoryzacją** n. Podstawowe twierdzenie arytmetyki, poza gwarancją istnienia gwarantuje również unikalność postaci kanonicznej, a więc daną liczbę możemy przedstawić za pomocą czynników pierwszych w tylko jeden sposób. Oczywiście, gdy n samo w sobie jest liczbą pierwszą, jej postać kanoniczna składa się z jej samej.

Powiemy, że liczba n jest **liczbą półpierwszą** (ang. *semiprime*), gdy jej postać kanoniczna to

$$n = ab \tag{1.3}$$

gdzie a i b są liczbami o podobnym rzędzie wielkości i $a \neq b$.

1.1.3. Złożoność obliczeniowa faktoryzacji

O ile podstawowe twierdzenie arytmetyki gwarantuje nam istnienie i unikalność rozkładu na czynniki, nie pokazuje ono *w jaki sposób* uzyskać postać kanoniczną liczby. Okazuje się to być nietrywialne. W ogólnym przypadku, gdy faktoryzowana liczba n posiada małe czynniki pierwsze (a zakładając, że n zostało dobrane w sposób losowy mamy na to bardzo duże szanse, jako że co druga liczba dzieli się przez 2, co trzecia przez 3 itd.) można z powodzeniem użyć znanego od dawna algorytmu naiwnego. Niestety, w przypadkach szczególnych (konkretnie, dla dużych liczb półpierwszych) problem ten jest właściwie nie rozwiązany, gdyż najlepsze znane algorytmy nie są w stanie zakończyć działania w rozsądnym czasie. Dzieje się tak dlatego, że działają one wykładniczo w zależności od liczby bitów (cyfr) danej liczby.

Określenie klasy złożoności problemu wymaga zdefiniowania na początek dwóch jego wersji:

1. Wersja funkcyjna problemu: dla danego $n \in \mathbb{N}, n > 1$ znaleźć liczbę $1 < d < n$ będącą dzielnikiem n bądź też stwierdzić, że n jest liczbą pierwszą.

2. Wersja decyzyjna problemu: dla danych $n, m \in \mathbb{N}$: $1 < m \leq n$ stwierdzić, czy n posiada czynnik $1 < d < m$.

Wersja decyzyjna jest przydatna, jako że większość badań na temat klas złożoności dotyczy właśnie problemów decyzyjnych. Warto zauważyć, że wersja decyzyjna w połączeniu z binarnym algorytmem wyszukiwania daje rozwiązanie wersji funkcyjnej problemu, a rozwiązanie wersji funkcyjnej bezpośrednio rozwiązuje wersje decyzyjną. Wynika z tego, że obie te wersje są *de facto* równoważne.

Wersja funkcyjna problemu w sposób trywialny należy do klasy **FNP**, która jest funkcyjnym odpowiednikiem klasy **NP**. Nie wiadomo, czy należy ona do klasy **FP**, czyli funkcyjnego odpowiednika klasy **P**

Nikt nie wie dokładnie, do której klasy problemów należy wersja decyzyjna. Na pewno znajduje się ona w klasach **NP** i **co-NP**. Dzieje się tak, ponieważ zarówno odpowiedź *Tak*, jak i *Nie* dają się szybko zweryfikować w czasie wielomianowym. Dzięki podstawowemu twierdzeniu arytmetyki (patrz rozdział 1.1.2) wiemy, że dowolna instancja problemu posiada tylko jeden wynik, zatem faktoryzacja należy do klas zarówno **UP**, jak i **co-UP** (wersji problemów z klas **NP** i **co-NP** posiadających dokładnie jedno rozwiązanie). Nie wiadomo, czy rozkład na czynniki pierwsze należy do klasy problemów **P**, **NPC** (**NP-zupełne**), czy **co-NPC**. Gdyby problem faktoryzacji należał do klasy **NPC** lub **co-NPC**, oznaczałoby to, że **NP = co-NP** co byłoby bardzo zaskakującym wynikiem, uważa się więc, że nie znajduje się on w żadnej z tych klas. Z drugiej strony wielu badaczy próbowało znaleźć wielomianowe rozwiązanie dla problemu i poniosło porażkę. Z tego powodu uważa się, że faktoryzacja nie należy do klasy **P**, a zatem jest kandydatem do klasy **NPI** (**NP-pośrednie**).

W związku z rozwojem badań na temat komputerów kwantowych, w 1994 roku Peter Shor odkrył algorytm faktoryzujący liczby w czasie wykładniczym ($O((\log N)^3)$) przy użyciu maszyny tego typu. Dowodzi to, że problem zawiera się w klasie **BQP** (ang. *Bounded error Quantum Polynomial time*). W 2001 roku implementacja algorytmu Shora na 7-kubitowym komputerze kwantowym rozłożyła na czynniki pierwsze liczbę 35. Mimo ciekawych wyników, które w bardzo duży sposób mogą zmienić oblicze problemu faktoryzacji liczb, technologia komputerów kwantowych jest jeszcze zbyt młoda aby stanowić realne zagrożenie dla jego trudności [3, 4].

1.1.4. Zastosowania

Trudność problemu rozkładu na czynniki pierwsze liczb całkowitych, okazuje się też stanowić o jego przydatności. Właśnie dzięki tej trudności (a także dzięki faktowi, że analogiczny problem *Czy n jest liczbą pierwszą?* należy do klasy **P**, a zatem posiada szybkie rozwiązanie), mogło zaistnieć wiele algorytmów szyfrowania danych (jak na przykład *RSA*). Dokładniej (na przykładzie algorytmu *RSA*), możliwość jego użycia (wygenerowania klucza w rozsądnym czasie) opiera się na prostocie znajdowania liczb pierwszych, podczas gdy jego bezpieczeństwo (czyli niemożność odczytania oryginalnej wiadomości bez posiadania klucza) opiera się na trudności problemu faktoryzacji. Innymi słowy, istnieje teoretyczna możliwość złamania zaszyfrowanej wiadomości, ale czas jaki zajęłaby ta czynność jest za długi (oczywiście, przy odpowiednim doborze rozmiaru klucza). Głównie więc z tego powodu tak istotne jest szukanie coraz to lepszych algorytmów rozkładających liczby całkowite - zarówno w celu łamania

zaszyfrowanych wiadomości, jak i sprawdzaniu na ile są one bezpieczne. Dla przykładu, W 1976 roku szacowano, że 129-cyfrowa liczba półpierwsza (RSA-129) jest bezpieczna jako klucz przez najbliższe 40 kwadrylionów ($4 \cdot 10^{25}$) lat. Liczba ta została sfaktoryzowana niecałe 20 lat później z użyciem algorytmu *MPQS* [5, 6]

Poza wartością kryptograficzną, w związku z trudnością problemu i koniecznością długotrwałych obliczeń problem faktoryzacji może być też użyteczny przy tworzeniu benchmarków dla jednostek obliczeniowych. Pomaga temu fakt, że wiele osób dokonuje implementacji co bardziej znanych i skuteczniejszych algorytmów na różnego rodzaju platformy. Przykładem jest użycie w tym celu procesorów graficznych (GPU) firmy NVidia, a także użycie procesora Cell, obecnego w konsoli do gier Sony PlayStation 3. Dzięki temu można w ciekawy sposób porównywać wydajność różnych platform [7, 8, 9, 10].

1.2. Algorytmy faktoryzacji

1.2.1. Podział algorytmów faktoryzacji

Omawiając istniejące metody faktoryzacji, należy zacząć od przedstawienia ich podziału. Istnieją zasadniczo trzy grupy, na które dzieli się te algorytmy:

1. **Algorytmy Specjalnego Przeznaczenia** (ang. *Special-Purpose*)

   Składają się na nie algorytmy, które radzą sobie szczególnie dobrze dla pewnego typu liczb, ale zauważalnie gorzej dla pozostałych. Przykładem jest algorytm Fermata, który działa bardzo szybko, gdy liczba składa się z dwóch czynników będących bardzo blisko siebie.

   Szczególną podklasę algorytmów specjalnego przeznaczenia tworzą tak zwane algorytmy **Pierwszej Kategorii** (ang. *First Category*), których czas działania zależy od rozmiaru najmniejszego z czynników pierwszych. Przykładem algorytmu pierwszej kategorii jest algorytm naiwny.

   Poniżej zestawiono niektóre z algorytmów specjalnego przeznaczenia:

   - Algorytm Naiwny
   - Algorytm ρ-Pollarda
   - Metoda Fermata
   - Metoda Eulera
   - Algorytmy opierające się na wykorzystaniu grup algebraicznych (ang. *Algebraic-Group Factorisation Algorithms*), których przykładami są algorytm $p-1$ Pollarda, algorytm $p+1$ Williamsa, a także najszybszy z tej rodziny algorytm *ECM* (ang. *Elliptic Curve factorization Method*) Lenstry.

2. **Algorytmy Generalnego Przeznaczenia** (ang. *General-Purpose*)

   Zwane także **Algorytmami Drugiej Kategorii** (ang. *Second Category*) lub **Rodziną Algorytmów Kraitchika** (ang. *Kraitchik Family*), jako że wszystkie opierają się na zastosowaniu idei, które

opracował Maurice Kraitchik, a które były rozwinięciem metody Fermata. Algorytmy te charakteryzują się takim samym czasem działania bez względu na postać kanoniczną liczby. Dla trudnych liczb (półpierwszych) algorytmy te są asymptotycznie najszybsze w działaniu. Dość duże stałe ukryte w notacji $O(\cdot)$ powodują jednak, że dla liczb innego typu dużo lepiej jest używać prostszych algorytmów specjalnego przeznaczenia.

Mając do dyspozycji liczbę, której postać kanoniczna jest dla nas nieznana, często stosuję się następujący schemat: Zaczyna się od próby wyciągnięcia mniejszych czynników jednym z algorytmów pierwszej kategorii, a dopiero po pewnym czasie przechodzi się do zastosowania jednego z asymptotycznie szybszych algorytmów drugiej kategorii.

Poniżej wymienione są niektóre z algorytmów ogólnego przeznaczenia:

- Algorytm Dixona
- Algorytm *CFRAC*
- Algorytm *QS*
- Algorytm *GNFS*

3. Inne algorytmy

   Do trzeciej grupy algorytmów zaliczają się pozostałe algorytmy takie jak algorytm Shora, przeznaczony na komputery kwantowe, czy też różnego rodzaju algorytmy probabilistyczne.

1.2.2. Charakterystyka wybranych algorytmów

Algorytm naiwny

Algorytm naiwny jest najprostszym i najwcześniej wymyślonym z algorytmów faktoryzacji. Paradoksalnie, dzięki prostocie działania i faktowi, że losowo wybrana liczba z bardzo dużym prawdopodobieństwem będzie posiadać niewielkie czynniki pierwsze (powiedzmy rzędu jej logarytmu), w większości przypadków algorytm ten sprawdza się najlepiej. Gdy zachodzi jednak konieczność faktoryzacji dużej liczby półpierwszej stosowanie go traci sens.

Działanie algorytmu polega na próbie dzielenia faktoryzowanej liczby n przez kolejne potencjalne czynniki poczynając od 2. W przypadku, gdy dla wartości k dzielenie jest wykonalne, znaleźliśmy dwa czynniki: k oraz $\frac{n}{k}$ (i możemy ewentualnie przejść do ich dalszej faktoryzacji). Ponieważ dla każdego znalezionego czynnika k istnieje drugi, odpowiadający mu $\frac{n}{k}$, potrzebujemy jedynie sprawdzać wartości $2 \leq k \leq \lfloor\sqrt{n}\rfloor$. Od razu można zauważyć, że nie trzeba sprawdzać *wszystkich* liczb z tego zakresu, a jedynie liczby pierwsze. W teorii daje nam to zysk w postaci krótszego czasu działania, ale w praktyce okazuje się, że narzut związany z wyszukiwaniem liczb pierwszych (bądź też przeprowadzaniem testu pierwszości) powoduje, że ta prosta modyfikacja nie jest opłacalna i w praktycznej implementacji testuje się każdą liczbę po kolei.

Czas działania algorytmu dla liczby n posiadającej b cyfr ($n \approx 2^b$ - oczywiście podstawa potęgi jest dowolna, jako że funkcje potęgowe o różnych podstawach są asymptotycznie równe) wyraża się jako

$\Theta(p)$, gdzie p jest najmniejszym czynnikiem pierwszym n. Gdy p jest małe - rzędu $b \approx \log n$ wynika z tego, że

$$\Theta(p) = \Theta(\log n) = \Theta(b) \tag{1.4}$$

czyli algorytm działa liniowo względem liczby cyfr (bitów). Niestety, dla dużych liczb półpierwszych $n = p_1 p_2$, czynniki p_1 i p_2 są tego samego rzędu, a więc

$$p_1 \approx p_2 \approx \sqrt{n} \tag{1.5}$$

z czego wynika, że złożoność obliczeniowa algorytmu zwiększa się do

$$\Theta(\sqrt{n}) = \Theta(2^{\frac{b}{2}}) = \Theta(2^b) \tag{1.6}$$

a więc jest wykładnicza w stosunku do b.

Algorytm Fermata

Algorytm Fermata - mimo, że w większości przypadków wolniejszy niż naiwny - jest niezwykle istotny, gdyż stanowi podstawę do działania szybszych algorytmów. Metoda ta opiera się na znalezieniu pary liczb a, b spełniających równanie:

$$a^2 - b^2 = n \tag{1.7}$$

gdzie n jest liczbą którą chcemy sfaktoryzować. Mając do dyspozycji takie a i b, możemy zapisać równanie (1.7) w innej postaci

$$(a - b)(a + b) = n \tag{1.8}$$

skąd bezpośrednio otrzymujemy dwa czynniki n: $a - b$ i $a + b$.

Pojawia się pytanie czy dla każdego $n = cd$ możemy znaleźć taką parę. Można sprawdzić, że dla dowolnego n zachodzi

$$n = (\frac{1}{2}(c + d))^2 - (\frac{1}{2}(c - d))^2 \tag{1.9}$$

gdy założymy dodatkowo, że n jest nieparzyste (co możemy zrobić bez straty ogólności, gdyż ewentualny czynnik 2 można z n wyciągnąć bardzo niskim nakładem kosztów), c i d również muszą być nieparzyste, a więc po prawej stronie (1.9) mamy różnice kwadratów liczb całkowitych - naszą parę a, b.

Poszukiwanie najczęściej przeprowadza się następująco: dla kolejnych wartości a z zakresu $\lceil \sqrt{n} \rceil \leq a \leq n$, sprawdzamy czy $b' = a^2 - n$ jest kwadratem liczby całkowitej. Jeśli tak, znaleźliśmy naszą parę a i $b = \sqrt{b'}$, a zatem sfaktoryzowaliśmy n.

Obserwując sposób poszukiwania naszej pary od razu widać, że algorytm Fermata działa specjalnie dobrze dla liczb, które posiadają dwa bardzo bliskie sobie czynniki (z tego powodu, przy generacji liczb pierwszych dla algorytmu *RSA*, liczby te powinny być podobne wielkością, ale nie za bliskie). Widać też jednak, że w pesymistycznym przypadku metoda potrzebuje

$$O(\sqrt{n}) = O(2^{\frac{b}{2}}) = O(2^b) \tag{1.10}$$

kroków, gdzie b jest liczbą cyfr (bitów) n, a więc pesymistyczna złożoność jest taka sama jak dla algorytmu naiwnego. W przeciwieństwie jednak do algorytmu naiwnego, gdzie optymistyczne dane wejściowe zdarzały się bardzo często, dla algorytmu Fermata przypadki, w których działa on dobrze zdarzają się bardzo rzadko (prawdopodobieństwo otrzymania na wejściu algorytmu n, które posiada dwa bardzo bliskie sobie czynniki jest bardzo małe). Powoduje to, że algorytm ten sam w sobie jest rzadko stosowany.

Schemat Kraitchika i liczby B-gładkie

W algorytm Fermata szukaliśmy par postaci $a^2 - b^2 = n$. Spróbujmy nieco rozluźnić warunek i rozważmy pary o postaci:

$$a^2 \equiv b^2 \pmod{n} \tag{1.11}$$

Zauważmy, że gdy mamy taką parę równanie to można zapisać inaczej

$$(a-b)(a+b) \equiv 0 \pmod{n} \tag{1.12}$$

a jeśli dodatkowo zachodzi warunek $a \not\equiv \pm b \pmod{n}$, to co najmniej jeden z dzielników $(a-b)$ i $(a+b)$ jest również dzielnikiem n, czyli otrzymujemy dwa czynniki n: $NWD(a-b, n)$ oraz $NWD(a+b, n)$. Algorytm Euklidesa na wyznaczanie *NWD* gwarantuje nam szybkie otrzymanie wyniku.

Oczywiście, nadal głównym problemem jest znalezienie odpowiedniej pary a, b. Przeglądanie po kolei liczb w nadziei, że trafimy na odpowiednią parę (tak jak w algorytmie Fermata) zdecydowanie zabiera zbyt dużo czasu. Z tego powodu wykorzystuję się inną metodę, którą w 1926 roku opracował Maurice Kraitchik [11]. Metoda wygląda następująco: generuje się pewną ilość par liczb a_i, b_i, takich że:

$$a_i \equiv b_i \pmod{n} \tag{1.13}$$

dla każdego i. Następnie, szukamy takiego podzbioru par a_i, b_i, że zachodzi równocześnie

$$\begin{aligned} \prod_j a_{i_j} &= x^2 \\ \prod_j b_{i_j} &= y^2 \end{aligned} \tag{1.14}$$

co daje nam parę liczb x, y

$$\begin{aligned} x &= \sqrt{\prod_j a_{i_j}} \\ y &= \sqrt{\prod_j b_{i_j}} \end{aligned} \tag{1.15}$$

dla której spełnione jest równanie (1.11).

Metoda Kraitchika nie rozwiązuje jednak całkowicie naszego problemu: w dalszym ciągu niewiadomymi są sposób generacji par a_i, b_i, a także sposób wyznaczania ich odpowiedniego podzbioru spełniającego (1.14).

W celu znalezienia rozwiązania, wprowadza się nowe pojęcie. Powiemy o $n \in \mathbb{N}, n > 1$, że jest **liczbą *B*-gładką**, jeśli dla jej dowolnego czynnika pierwszego p zachodzi $p \leq B$. Zauważmy przy tym, że liczby B-gładkie są bardzo proste do sfaktoryzowania (dla rozsądnych wartości B). Zamiast więc szukać par spełniających równanie (1.11), rozluźnijmy warunki i szukajmy par a, b, dla których spełnione jest równanie:

$$a^2 \equiv b \pmod{n} \tag{1.16}$$

gdzie b jest liczbą B-gładką (dla wcześniej ustalonego B). Mając odpowiednio duży zbiór par typu (1.16), możemy następnie skorzystać ze schematu Kraitchika. Zauważmy, że po lewej stronie kongruencji mamy zawsze kwadrat liczby całkowitej, więc szukając odpowiedniego podzbioru dla metody Kraitchika, rozpatrujemy tylko prawą stronę.

Pytanie, w czym pomaga nam informacja, że liczba b jest B-gładka? Zauważmy, że dowolną liczbę B-gładką możemy zapisać w postaci kanonicznej (1.2), gdzie wszystkie z czynników p_i są ograniczone przez B. Rozważmy ciąg (p_i) kolejnych liczb pierwszych mniejszych od B (nazywany także **bazą czynników pierwszych**, ang. *factor base*), a także odpowiadający mu ciąg (α_i), będący ciągiem wykładników w postaci kanonicznej dowolnej liczby b (jeśli dany czynnik p_j nie występuje w postaci kanonicznej liczby, odpowiadający mu wykładnik wynosi 0). Ciąg skończony (α_i) nazywany jest **wektorem wykładników** (ang. *exponent vector*) liczby b. Zauważmy, że wektor wykładników jednoznacznie określa b, dla danej bazy czynników pierwszych. Jest jasne, że liczba jest kwadratem, wtedy i tylko wtedy, gdy każdy element w jej wektorze wykładników jest parzysty. Zatem, mając dany zbiór takich wektorów (odpowiadających zbiorowi liczb B-gładkich), szukanie ich kombinacji, która w efekcie daje kwadrat sprowadza się do rozwiązania układu równań $\pmod 2$, a do tego istnieją już efektywne algorytmy. Warto przypomnieć, że aby zagwarantować sobie istnienie rozwiązania dla tego układu równań, powinniśmy mieć o jedną parę a, b więcej niż liczb w bazie czynników pierwszych.

Oczywiście, wykonując kolejno kroki, możemy dojść do rozwiązań, gdzie co prawda $a^2 \equiv b^2 \pmod n$, ale $a \equiv b \pmod n$ lub $a \equiv -b \pmod n$. W takim przypadku po wyznaczeniu *NWD* dostajemy zupełnie nieinteresujące nas czynniki trywialne 1 i n. Jeśli mamy do czynienia z tą sytuacją, można spróbować użyć innego rozwiązania układu równań, a jeśli wyczerpaliśmy już wszystkie rozwiązania, nie pozostaje nic innego jak szukać kolejnych par. Można jednak dowieść, ze dla n składającego się z co najmniej dwóch czynników, prawdopodobieństwo uzyskania nietrywialnej pary wynosi $\frac{1}{2}$, zatem szanse znalezienia interesującego nas rozwiązania rosną szybko przy kolejnych sprawdzanych parach.

Powyższy przepis rozszerzający schemat Kraitchika, mimo że nie jest jeszcze pełnoprawnym algorytmem, jest bazą dla wszystkich nowoczesnych algorytmów faktoryzujących.

Algorytm Dixona

Algorytm Dixona, który opracował w 1981 roku John Dixon [12] jest najprostszym zastosowaniem w praktyce schematu z rozdziału (1.2.2). Sposób działania jest bardzo prosty: losujemy kolejne liczby a takie, że $\lceil \sqrt{n} \rceil < a < n$ i sprawdzamy (używając algorytmu naiwnego), czy $b = a^2 \bmod n$ jest liczbą B-gładką, dla ustalonego B. Jeśli tak, oczywiście dodajemy znalezioną parę a, b do zbioru.

Algorytm Dixona, mimo, że lepszy dla dużych liczb niż metoda naiwna, jest bardzo wolny jak na metody z *Rodziny Kraitchika*, ustępując praktycznie każdemu z nich. Nie ma więc on zastosowania praktycznego.

Algorytm *CFRAC*

Algorytm *CFRAC* (ang. *Continued Fraction method*) jest historycznie pierwszym zastosowanym w praktyce algorytmem z *Rodziny Kraitchika*. Idea pozostaje bez zmian, definiowany jest natomiast sposób wybierania par spełniających (1.16). Zamiast losować liczby jak w algorytmie Dixona, używamy ciągu współczynników rozwinięcia $\sqrt{n}$ w ułamek łańcuchowy. Współczynniki te spełniają zależność (1.16), ale co najważniejsze, spełniają też dodatkową zależność $b_i < 2\sqrt{n}$. Ograniczenie wielkości liczb b_i jest bardzo istotne, jako że mniejsza liczba ma dużo większe szanse, by być liczbą B-gładką.

Metodę zaproponowali Derrick Lehmer i R. Powers w 1931 roku [13], a zaimplementowali ją Michael Morrison i John Brillhart w roku 1975 [14]. Algorytm *CFRAC* okazał się być bardzo skuteczny w praktyce i królował bijąc rekordy tamtych czasów w faktoryzacji liczb do opracowania algorytmu sita kwadratowego.

Algorytm GNFS

Wspominając o algorytmach faktoryzacji nie można pominąć Algorytmu Sita Ciała Liczbowego (ang. *GNFS*, *General Number Field Sieve*), który jest zarówno najnowszą metodą z *Rodziny Kraitchika*, jak i też najszybszą. Mimo, że dla dużych liczb półpierwszych posiadających mniej niż 150 cyfr *GNFS* jest wolniejszy niż algorytm *QS*, dla liczb większych jest on nieporównywalnie szybszy. Aktualnie wszystkie rekordy w faktoryzacji dużych liczb pierwszych należą do tej właśnie metody.

Zasada działania algorytmu jest generalizacją prostszej metody *SNFS* (ang. *Special Number Field Sieve*), która działa tylko dla niektórych liczb. *GNFS* jest algorytmem o dużo bardziej skomplikowanej strukturze, którego omawianie wychodzi poza ramy tej pracy.

Idee algorytmu zaproponował w 1988 roku John Pollard, lecz ostateczna wersja powstała dopiero w 1994 [15, 16]

1.2.3. Aktualne rekordy

Za rozwojem algorytmów podążają coraz większe rekordy faktoryzacji. Niewątpliwy wpływ na ten stan rzeczy mają również projekty takie jak *Cunningham project* [17], czy zawody takie jak *RSA Factoring Challenge* [18]. Poza wartością badawczą, wyzwania te pozwalają też na bieżąco oceniać siłę aktualnych algorytmów kryptograficznych i stosowanych do nich kluczy. Z tego powodu angażują się w nie prywatne firmy, jak *RSA Security*, fundując niemałe nagrody.

Początkowo, rekordy były bite za pomocą algorytmu CFRAC, który ustąpił dopiero w latach 80 XX w. algorytmowi sita kwadratowego. Za pomocą algorytmu QS sfaktoryzowano, między innymi, liczby RSA-100, RSA-110, RSA-120 i RSA-129 [5]. Następnie na scenę wkroczył algorytm sita ciała liczbowego, który do dziś pozostaje najszybszym ze znanych algorytmów faktoryzacji i dzierży rekordy - najpierw RSA-640 (640 bitów, 193 cyfry) i RSA-200 (200 cyfr) w 2005 roku, a następnie RSA-768 (232 cyfry) w 2009 roku, co pozostaje niepobitym rekordem do dziś [19, 20, 21].

2. System operacyjny Android

2.1. Urządzenia mobilne

2.1.1. Rozwój urządzeń mobilnych w ciągu ostatnich lat

Telefony komórkowe przeszły długą drogę od powstania do dzisiejszych czasów. Kiedyś wielkie, nieporęczne urządzenia, które nie sposób było nosić ze sobą i trzymano w autach - dziś kieszonkowe mini-komputery, które możliwościami nie ustępują swoim stacjonarnym odpowiednikom z przed zaledwie kilku lat.

Pierwsza rozmowa z użyciem urządzenia mobilnego odbyła się w 1946 roku. Należy przy tym zaznaczyć, że mobilność tego telefonu była w owym czasie mocno ograniczona - ważył on 40 kilogramów i był przeznaczony do przewożenia i prowadzenia rozmów jedynie w samochodzie. Postęp jednak był szybki - już w 1973 roku pracownik firmy Motorola - Martin Cooper dokonał pierwszej rozmowy przy użyciu telefonu, który można było przenosić w rękach. Ważył on niecały kilogram, co jak na dzisiejsze standardy jest wartością ogromną, jednak wtedy robiło wrażenie. Mimo, że kosztował on 4000 USD na urządzenie znalazły się rzesze chętnych, co oczywiście nakręcało rozwój podobnych produktów. W 1979 roku uruchomiono pierwszą komercyjną sieć komórkową, a cztery lata później powstała pierwsza sieć 1G (ang. *First Generation*). Sieć 2G (ang. *Second Generation*) zastąpiła ją w 1991 roku, aby samej zostać wypartą dziesięć lat później przez 3G (ang. *Third Generation*), a następnie modyfikacje tej ostatniej jeszcze bardziej przyspieszające szybkość przesyłania danych.

Oczywiście, wzrost możliwości sieci był powodowany wzrostem możliwości urządzeń w ich ramach działających. Koncepcja **smartfona** - telefonu, którego funkcje nie ograniczają się tylko do dzwonienia, była obecna już od 1973 roku, jeszcze przed powstaniem produktu Motoroli. Trzeba było jednak zaczekać do roku 1992 aby ujrzeć pierwszy prototyp urządzenia tego typu o nazwie Simon (konstrukcji firmy IBM), a do 1996 aby smartfony pojawiły się w sprzedaży detalicznej (Nokia 9000 Communicator). Dziś, piętnaście lat później, raporty statystyczne mówią o globalnej sprzedaży prawie 500 milionów smartfonów w roku 2011 - co stanowi ponad 30% urządzeń mobilnych sprzedanych w tym samym czasie [22, 23], a w niektórych krajach odsetek ten wynosi jeszcze więcej sięgając prawie połowy [24, 25].

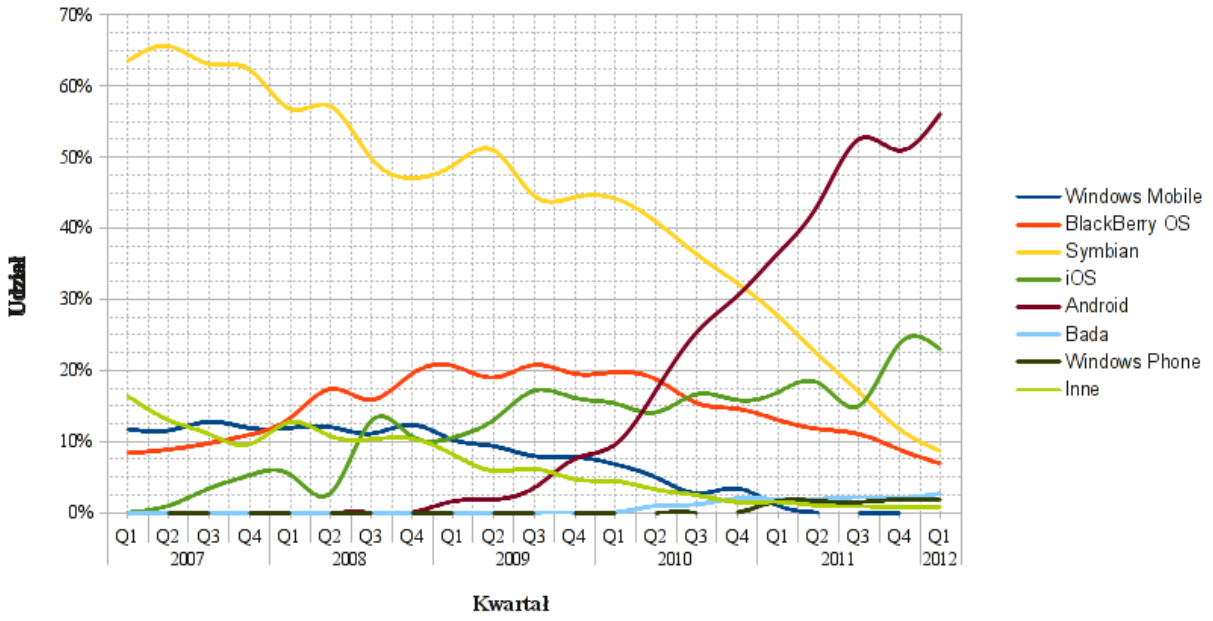
Niezwykle szybki przyrost możliwości (w tym możliwości sprzętowych, takich jak moc obliczeniowa) przywodzi na myśl podobną historie komputerów osobistych. Wydaje się jednak, że w przypadku telefonów proces ten jest jeszcze szybszy. Dla przykładu, ledwo dwadzieścia lat temu były to urządzenia służące właściwie tylko do prowadzenia rozmów głosowych. Dziś najnowsze dostępne na rynku modele oferują takie parametry jak czterordzeniowe procesory z osobnym układem graficznym, czy 2 GB

pamięci operacyjnej - wartości do nie tak dawna zarezerwowane dla urządzeń klasy PC. Wygląda na to, że w najbliższym czasie smartfony dogonią możliwościami komputery, a może nawet wypchną je z rynku zastępując ich miejsce (jako podłączane do stacji dokujących jednostki obliczeniowe). I choć dziś jeszcze nie ma o tym mowy i tak zajmują one bardzo ważne miejsce w naszym życiu codziennym.

2.1.2. Mobilne systemy operacyjne

Istnieje wiele różnych sposobów na klasyfikacje istniejących dziś telefonów komórkowych. Niewątpliwie jednak najważniejszym z nich jest podział ze względu na rodzaj systemu operacyjnego który tym telefonem zarządza. Powiemy o urządzeniu, że jest ono **smartfonem** (ang. *smartphone*), gdy jego system operacyjny jest typu *smart* (z ang. *mądry, inteligentny*), czyli gdy jest on pełnoprawną platformą deweloperską o spójnym *API* (ang. *Application Programming Interface*), umożliwiającym tworzenie rozbudowanych aplikacji klienckich przez firmy trzecie. Tradycyjne telefony nie spełniają tych wymagań, będąc zdolnymi jedynie do uruchamiania aplikacji wgranych na niego (i napisanych) bezpośrednio przez producenta, bądź też aplikacji zewnętrznych mających bardzo ograniczone możliwości jak na przykład oprogramowanie stworzone przy użyciu Java ME (ang. *Java Mobile Edition*).

Szybki rozwój telefonów w ostatnim czasie połączony z dużym na nie popytem spowodował, że aktualnie na rynku dostępnych jest kilka liczących się systemów operacyjnych diametralnie się od siebie różniących. Poniżej na rysunku 2.1 został przedstawiony aktualny podział rynku na konkretne systemy operacyjne, a dalej opisane są pokrótce najważniejsze z nich.



Rysunek 2.1: Podział rynku na poszczególne mobilne systemy operacyjne

Symbian

Symbian OS powstał dzięki porozumieniu pomiędzy firmami Nokia, NTT DoCoMo, Sony Ericsson oraz Symbian Ltd. Historycznie jest to najstarszy szeroko dostępny system typu smart. Dzięki wsparciu wielu producentów sprzętu, szczególnie Nokii będącej do niedawna największym z nich, stał się on

bardzo popularny w pierwszej połowie ostatniej dekady. Niestety, dość duże skomplikowanie platformy spowodowało że po pojawieniu się prostszych alternatyw system ten zaczął podupadać. Nawet Nokia, która zainwestowała duży kapitał w jego rozwój (m.in. kupiła w 2008 roku firmę Symbian Ltd.) aktualnie wypuszczając telefony z górnej półki korzysta ze swoich późniejszych autorskich rozwiązań jak MeeGo, czy Maemo lub też z Windows Phone 7. Mimo tego, Symbian wciąż utrzymuje wysokie noty, głównie w związku z faktem, że wiele prostych (i tanich) telefonów Nokii działa pod jego kontrolą. Nie ma jednak złudzeń, że sytuacja ta to tylko kwestia czasu.

BlackBerry OS

RIM (ang. *Research In Motion*) to kanadyjska firma, która dzięki linii telefonów BlackBerry wykorzystujących jej autorski system operacyjny, przez długi czas utrzymywała się w czołówce dostawców Smartfonów. Telefony BlackBerry charakteryzowały się zawsze biznesowym przeznaczeniem, choć w późniejszym czasie starano się rozszerzyć grupę docelowych klientów także o zwykłych użytkowników. Poza samym systemem operacyjnym RIM dostarcza dodatkowych aplikacji (takich jak na przykład klient e-mail, czy messenger), z których każda kładzie duży nacisk na bezpieczeństwo danych. Kilka lat temu telefon tej firmy był wyznacznikiem prestiżu, jednak dziś akcje RIM spadają, gdyż firma nie jest w stanie skutecznie konkurować z produktami działającymi pod kontrolą iOS, czy Androida.

Mobilne systemy operacyjne z rodziny Windows

Firma Microsoft - twórca systemu operacyjnego Windows, który od kilkunastu lat niezmiennie panuje na komputerach osobistych - nie mogła sobie pozwolić na nieobecność na szybko rozwijającym się rynku urządzeń mobilnych. Prace nad oprogramowaniem tego typu rozpoczęły się już w 1990 roku. Dwa lata później zaczęły powstawać pierwsze wersje Windows CE bazującego na desktopowym systemie Windows 95, nie zdobyły one jednak popularności na telefonach komórkowych, będąc używane głównie na urządzeniach klasy PDA (ang. *Personal Device Assistant*). W 2000 roku został zaprezentowany bazujący na Windows CE 3.0 Windows Mobile, który w przeciwieństwie do poprzednika okazał się marketingowym sukcesem zdobywając dużą część rynku i skutecznie konkurując z Symbianem i BlackBerry OS. W późniejszym okresie jednak został on zepchnięty na bok nie wytrzymując konfrontacji z iOS i Androidem.

W 2010 roku Microsoft ogłosił, że pracuje nad kolejnym systemem z serii - Windows Phone 7. Mimo, że w pewnym sensie jest on kontynuacją poprzednika, zmiany są bardzo duże i platformy te nie są ze sobą kompatybilne. Dodatkowo Windows Phone ma całkowicie odmieniony interfejs o nazwie Metro, który oferuje innowacyjne podejście do obsługi telefonu. Główną motywacją przy tworzeniu platformy było uzyskanie urządzenia, które działa bardzo szybko, i na którym w prosty sposób można dokonać podstawowych czynności. Ideologia systemu jest więc bardzo zbliżona do tej stworzonej przez firmę Apple: proste, intuicyjne, płynne i zamknięte oprogramowanie.

System Windows Phone 7 wyparł całkowicie Windows Mobile w wersji 6.5 w 2011 roku. Mimo tego, jak dotąd (pomimo bardzo dużych wydatków na kampanie reklamową) nie zdołał osiągnąć znaczącego sukcesu na rynku, oscylując w granicach kilku procent.

iOS

Firma Apple weszła na rynek mobilny w 2007 roku wydając swój produkt - telefon iPhone, działający pod kontrolą autorskiego nienazwanego systemu operacyjnego, który bazował na desktopowym systemie Mac OS X 10.5. System ten w późniejszych wersjach zyskał miano iOS. Urządzenie okazało się być niezwykle innowacyjne, wyznaczając nowe trendy na rynku smartfonów, obowiązujące do dziś. Charakteryzował się on dużym, dotykowym ekranem zajmującym większość jego przedniej obudowy i brakiem fizycznej klawiatury. Do dziś nie zmienił się zbytnio ani wygląd zewnętrzny kolejnych jego generacji, ani wygląd systemu, a jedyne różnice stanowią coraz mocniejsze komponenty sprzętowe.

iOS jako system charakteryzuje się prostotą używania, szybkością i płynnością działania, a także zamkniętością na modyfikacje innego rodzaju niż instalacja aplikacji z oficjalnego sklepu AppStore. Pod kontrolą tego systemu działają jedynie produkty firmy Apple: tablet iPad, odtwarzacz muzyki iPod i sam telefon iPhone. Dzięki temu system może być bardzo mocno zoptymalizowany pod kątem konkretnych urządzeń. Od czasu debiutu na rynku popularność produktu wciąż rośnie i smartfon znajduje się w ścisłej czołówce.

Android

Android jest systemem wydanym w 2008 roku przez firmę Google. Okazał się on być wielkim marketingowym sukcesem i w krótkim czasie zdołał awansować na pierwsze miejsca rankingów popularności. Szczegółowy jego opis znajduje się w rozdziale 2.2

Pozostałe systemy

Na rynku jest dostępnych jeszcze kilka systemów, które nie zdobyły jak do tej pory tak dużej popularności. Należą do nich na przykład Bada OS (autorski system firmy Samsung, używany tylko w telefonach tej firmy), MeeGo i Maemo (wyprodukowane przez firmę Nokia, które mimo niedawnego debiutu zdołały w krótkim czasie zostać porzucone na rzecz Windows Phone 7), czy PalmOS i WebOS (Produkty firmy Palm wykupionej przez HP, aktualnie porzucone). Wiele z nich (na przykład MeeGo, Maemo, WebOS) bazuje podobnie jak Android na desktopowym systemie operacyjnym Linux. Mimo faktu, że niektóre stanowią bardzo udane konstrukcje, aktualnie ich penetracja rynku jest minimalna, a polityka firm je tworzących nie wróży zmianie sytuacji.

2.1.3. Wyzwania w programowaniu na urządzenia mobilne

Mimo, że możliwości dzisiejszych telefonów komórkowych są duże, tworzenie oprogramowania dla nich w wielu aspektach wciąż stanowi wyzywanie w związku z ich sprzętowymi barierami. Głównym powodem jest mały ekran, który ogranicza interfejs użytkownika - mimo że rozdzielczości wyświetlaczy ciągle rosną (aktualne rekordy oscylują w granicach HD 1080x720 pikseli), wzrost fizycznych ich rozmiarów jest dużo wolniejszy. Wyzwaniem jest też dostępna pamięć RAM - 1 GB do maksymalnie 2 GB jest ilością niemałą, jednak zauważalnie mniejszą niż ta dostępna na komputerach stacjonarnych. Ostatnim, ale bardzo ważnym problemem jest pojemność baterii. Nawet najmocniejsze urządzenia wytrzymują niewiele dłużej niż dzień przy umiarkowanym użytkowaniu. Komplikuje to proces tworzenia

aplikacji, gdyż wymagana jest optymalizacja pod tym kątem, a także odbija się na wydajności niektórych komponentów sprzętowych. Przykładem ograniczeń tego ostatniego typu są procesory, które mimo dużej teoretycznej mocy obliczeniowej w praktyce osiągają gorsze wyniki przez dążenie do energooszczędności.

2.2. Charakterystyka systemu Android

2.2.1. Powstanie i rozwój

Historia systemu Android rozpoczyna się założeniem firmy Android Inc. w 2003 roku. Firma ta miała projektować "mądrzejsze urządzenia mobilne", choć z początku nie było do końca jasne co miałoby to znaczyć, jako że działała ona w tajemnicy. W 2005 roku na przedsiębiorstwo zwrócił swoją uwagę internetowy gigant - Google, który zamierzał wejść na rynek mobilny z nowymi produktami. Standardową polityką tej korporacji stosowaną w takim wypadku jest wyszukiwanie małych firm działających w danej dziedzinie, a następnie wykupywanie ich. Tak też wyglądał scenariusz i w tej sytuacji - w sierpniu Android Inc. zmienił właściciela. Od tego też czasu jedynym zadaniem zespołu stało się tworzenie nowego autorskiego systemu operacyjnego bazującego na jądrze Linuksa. Kolejnym kamieniem milowym było ogłoszenie w 2007 roku powstania Sojuszu Otwartych Telefonów Komórkowych (ang. *Open Handset Alliance*), w którego skład wchodziły między innymi firmy takie jak Google, Samsung, HTC, Intel, Motorola, T-Mobile, a także wiele innych. Celem stowarzyszenia było stworzenie otwartych standardów dla urządzeń mobilnych na świecie. Ogłoszono też pierwszy produkt sojuszu - wersję beta mobilnego systemu operacyjnego Android, bazującego na Linuksowym jądrze w wersji 2.6.

Od początkowej demonstracji system rozwijał się w szybkim tempie. W 2008 roku zaprezentowano wersję 1.0 i wydano pierwszy telefon działający pod kontrolą Androida - HTC Dream. W roku 2009 zostały wypuszczone cztery kolejne wersje systemu - 1.1, 1.5, 1.6 i 2.0, powstało też więcej urządzeń działających pod jego kontrolą. Kolejne wersje 2.1, 2.2 i 2.3 pojawiły się na rynku w roku 2010. W tym też czasie miało miejsce inne bardzo ważne wydarzenie w branży mobilnej: w styczniu firma Apple ogłosiła wydanie swojego najnowszego urządzenia, tabletu iPad. Produkt ten będący powiększoną wersją smartfona iPhone i działający pod kontrolą tego samego systemu operacyjnego co swój mniejszy odpowiednik odniósł wielki sukces marketingowy. Mimo, że nie działał on pod kontrolą Androida, wpłynął bardzo mocno na cały rynek urządzeń mobilnych. Wszyscy producenci sprzętu zaczęli projektować tablety, chcąc załapać się na falę ich popularności. Samsung wypuścił pierwszy tablet z Androidem w wersji 2.3 - Galaxy Tab pod koniec roku 2010, Google utrzymywało jednak że w aktualnym stanie system nie jest gotowy do operowania na tak dużych ekranach, co spowodowało brak jego certyfikacji. Aby zapobiec podobnym problemom i zaspokoić żądania producentów na początku 2011 roku wydany został Android w wersji 3.0 (a następnie 3.1 i 3.2) przeznaczony tylko na urządzenia typu tablet. Niedługo później, w trzecim kwartale została zaprezentowana wersja 4.0, która łączyła dwie gałęzie produktu z powrotem w całość.

Systemowi Android towarzyszył nieustanny wzrost udziałów rynkowych. W drugim kwartale roku 2009 - nieco ponad pół roku od wydania HTC Dream - szacowano, że telefony tego typu stanowią około

2.8% wszystkich smartfonów na świecie, liczba ta jednak szybko się zwiększała. W 2010 roku kolejno ogłaszano że dziennie aktywowanych jest 100 000 (w maju), 200 000 (w sierpniu), a następnie 300 000 (w grudniu) urządzeń. Pod koniec czwartego kwartału udział procentowy wyniósł już 33%, dzięki czemu Android stał się najpopularniejszym na świecie mobilnym systemem operacyjnym. Kolejne rekordy padły w roku kolejnym. W maju mówiło się o 400 000 dziennych aktywacji, a niedługo później (w czerwcu) już o 550 000. Wtedy też ogłoszono, że całkowita liczba aktywowanych urządzeń (od powstania systemu) wyniosła 100 milionów. Do listopada zwiększyła się ona do 200 milionów, a w grudniu podano, że liczba dziennych aktywacji wzrosła do 700 000. Pod koniec roku 2011 szacowano, że ponad połowa wszystkich sprzedawanych smartfonów działa pod kontrolą Androida. W lutym 2012 firma Google poinformowała, że liczba dziennych aktywacji po raz kolejny się powiększyła i wynosi 850 000. Najnowsze dane (pochodzące z początku czerwca) wykazały kolejny wzrost, tym razem do 900 000. Patrząc na tendencje, można spodziewać się, że w najbliższym czasie przekroczona zostanie magiczna bariera miliona nowych urządzeń aktywowanych dziennie.

2.2.2. Właściwości systemu Android

Patrząc na tak szybki wzrost udziałów (od 0% do ponad 50% w przeciągu trzech lat) można się zastanowić co takiego wyróżnia system Android. W końcu w momencie wejścia, rynek smartfonów nie był nowy i był już dawno podzielony pomiędzy urządzenia z Symbianem (głównie Nokia), telefony BlackBerry i telefony z systemem Windows Mobile. W dodatku niedługo wcześniej debiutował niosący powiew świeżości iPhone, który zdobywał coraz większą grupę zwolenników. Mogło się więc wydawać, że nie ma zapotrzebowania na kolejny system operacyjny.

Jedną z głównych zalet Androida jest jego szeroko rozumiana **otwartość**. Otwarty jest kod źródłowy, otwarty jest także sklep z aplikacjami. Każdy może pobrać źródła systemu, po czym dostosować je do swoich potrzeb, a następnie używać - także komercyjnie. Było to zresztą powodem, dla którego właśnie ten system został wybrany na flagowy produkt Open Handset Alliance. Ponieważ Android bazuje na jądrze Linuksa (który sam w sobie jest otwarty) można bez problemu ingerować w jego wnętrze - na przykład aby uzyskać dostęp do konta root, co daje szereg dodatkowych uprawnień i możliwości. Dodatkowo nie istnieje procedura weryfikacyjna przy umieszczaniu aplikacji w sklepie Google Play (dawniej Android Market). Jedyne wymagania dla deweloperów to jednorazowa opłata na utrzymanie infrastruktury. Można też tworzyć oprogramowanie bez tej opłaty, a aplikacje rozprzestrzeniać w inny sposób. Są to duże różnice w stosunku na przykład do polityki firmy Apple, czy Microsoft.

Powyższe cechy powodują, że wielu producentów urządzeń mobilnych wybiera właśnie ten system, gdyż jest on darmowy i łatwo go dostosować do swoich urządzeń. Dzięki temu na rynku pojawił się wysyp telefonów o bardzo zróżnicowanych parametrach i w różnym przedziale cenowym dając konsumentom wybór, którego na przykład nie ma przy zakupie smartfona iPhone. Mogą powstawać zarówno modele charakteryzujące się gorszymi parametrami, ale o przystępnej cenie (dzięki czemu każdy może sobie pozwolić na ich zakup), jak i te mocniejsze o cenach wyższych.

Należy przy tym dodać, że otwartość ta nie jest całkowita, co często jest głównym zarzutem przeciwników systemu. Kod Androida jest rozwijany w firmie Google, która udostępnia swoje zmiany tylko przy

wydawaniu kolejnych wersji. Co więcej, na przykład po wydaniu wersji 3.x przeznaczonej na tablety, kod źródłowy nie został wydany publicznie w ogóle, będąc przekazanym tylko wybranym producentom sprzętu. Poza tym, nawet gdy źródła są już udostępnione, aby używać dodatkowych aplikacji takich jak Google Play, Gmail, Kalendarz i wiele innych (bez których telefon traci wiele swojej użyteczności) wymagana jest certyfikacja od firmy Google, która mimo, że niezbyt trudna, wiąże się z pewnymi kosztami. Nie ma też zupełnej samowoli w sklepie z aplikacjami. Publikować może w nim każdy, ale jest on nadzorowany przez firmę, która zastrzega sobie prawo do blokowania kont i usuwania poszczególnych produktów. Trzeba jednak podkreślić, że takie wypadki zdarzają się tylko wtedy, gdy aplikacje owe nie spełniają regulaminu sklepu, czyli na przykład naruszają prawa autorskie lub zawierają złośliwy kod.

Drugą bardzo ważną cechą systemu jest mocna integracja z innymi produktami firmy Google. Wyszukiwarka, Narzędzia takie jak Gmail, YouTube, Google Talk, Google Maps (a także wiele, wiele innych) są coraz częściej używane nie tylko na komputerach stacjonarnych, ale też na smartfonach. O ile większość z nich działa bez problemu na większości mobilnych systemów operacyjnych (albo jako natywne aplikacje albo przy pomocy mobilnej przeglądarki), jest dość oczywiste, że najlepiej są one związane z Androidem.

Nie sposób też nie wspomnieć o najważniejszym (z punktu widzenia korporacji) produkcie - reklamy. Firma Google zdecydowaną większość swoich dochodów opiera na serwowaniu spersonalizowanych treści sponsorowanych. Na takiej samej zasadzie Android zarabia na siebie. Dostarcza on API dla twórców aplikacji ułatwiając im wyświetlanie reklam użytkownikowi. W momencie kliknięcia, niewielka kwota z tego tytułu jest przekazywana deweloperowi. Taka praktyka ma pewien negatywny wpływ na jakość Google Play, gdyż znajduje się w nim dużo produktów przeładowanych reklamami. Z drugiej jednak strony, pozwala ona udostępniać programistom aplikacje za darmo i wciąż na nich zarabiać. Dzięki temu sklep firmy Google znajduje się na pierwszym miejscu, jeśli chodzi o ilość darmowych pozycji.

2.2.3. Wersje

System Android ciągle się rozwija. Aktualnie przyjęta polityka firmy Google, to wydawanie dwóch jego wersji w ciągu roku. Każda z wersji zawiera wiele poprawek i usprawnień, a także szereg nowych funkcjonalności. Mimo wielu pozytywnych stron takiego działania istnieją też związane z tym problemy, opisane dokładniej w rozdziale 2.2.4.

Google oznacza wersje przez nadanie numeru, a także nazwy kodowej. Nazwy te tradycyjnie są nazwami deserów rozpoczynających się od kolejnych liter alfabetu. Dodatkowo często wybierany jest konkretny model telefonu specjalnie na tę okazje przygotowany przez jednego z producentów sprzętu, który staje się na pewien czas modelem flagowym. Niektóre z tych urządzeń należą do wspólnej rodziny Nexus, co ma znaczyć że zawierają one niemodyfikowaną przez producentów ani przez operatorów wersje systemu (ang. *Pure Google Experience*).

pre-Cupcake

Pierwsze wersje Androida to beta wydana pod koniec 2007 roku, 1.0 (Apple Pie) wydana w trzecim kwartale 2008 roku i 1.1 (Banana Bread) wydana na początku roku 2009. Ciekawostką jest, że tylko jeden telefon używał tych wersji - HTC Dream (zwany też HTC G1), nie było to jednak urządzenie z

rodziny Nexus. Jak na inicjalną wersje systemu przystało brakowało wielu funkcjonalności, co jednak nie przeszkodziło mu znaleźć nabywców. Androida w tamtym czasie charakteryzowało wsparcie dla WiFi i Bluetooth (aczkolwiek dla tego drugiego niepełne), wsparcie dla komunikacji z serwerami e-mail, pasek notyfikacji, wsparcie dla aparatu fotograficznego, możliwość zmiany tapety i grupowania ikon w folderach, a także szereg aplikacji tworzonych przez Google takich jak Android Market, przeglądarka, Gmail, Contacts, Calendar, Maps, Talk, Search (i wiele innych) wraz z synchronizacją z serwerami.

Cupcake, Donut

Wersje 1.5 (Cupcake) oparta na linuksowym jądrze 2.6.27 i 1.6 (Donut) oparta na jądrze 2.6.29 pojawiły się odpowiednio w drugim i trzecim kwartale 2009. W tym czasie pojawiły się też kolejne modele telefonów, na których wersje te mogły działać. Lista nowych funkcjonalności w Cupcake obejmuje pełne wsparcie Bluetooth, wsparcie dla nagrywania wideo, wsparcie dla wirtualnych klawiatur tworzonych przez firmy trzecie, wsparcie dla widgetów, a także drobniejsze zmiany takie jak dodanie funkcji kopiuj-wklej w przeglądarce, dodanie zdjęć kontaktów, czy dodanie opcji autorotacji ekranu. Donut dodał do tej listy wsparcie dla wyszukiwania w aplikacjach, wsparcie dla CDMA/EVDO, 802.1x i VPN oraz wiele pomniejszych usprawnień.

Eclair, Froyo

Android w wersjach 2.0, 2.0.1 i 2.1 (Eclair) był wydawany między październikiem 2009 a styczniem 2010. Bazował on na na jądrze 2.6.29, a lista zmian obejmowała wsparcie dla Bluetooth 2.1, wsparcie dla synchronizacji wielu kont, optymalizacje sprzętowe, wsparcie dla Exchange i wiele innym mniej ważnych modyfikacji. Telefonem, na którym Google testowało wydanie była początkowo Motorola Droid (Milestone), a później pierwszy telefon z rodziny Nexus - Nexus One od firmy HTC. Pół roku później w maju została wydana wersja 2.2 (Froyo) oparta na jądrze 2.6.32. Dodawała ona wiele usprawnień głównie przyspieszających działanie telefonu, między innymi poprzez implementacje JIT (ang. *Just In Time Compilation*), a także nowy silnik JavaScript V8 dla przeglądarki, C2DM (ang. *Cloud to Device Messaging*) umożliwiające notyfikacje typu *push*, USB i WiFi tethering i wiele innych pomniejszych. Była to też pierwsza wersja obsługująca Adobe Flash. Froyo było tworzone w oparciu o telefon Nexus One.

Gingerbread

Wersja 2.3 (Gingerbread) bazująca na jądrze 2.6.35 została wydana w grudniu 2010 roku. Oferowała między innymi nowy interfejs, wsparcie dla większych ekranów, wsparcie dla SIP, wsparcie dla NFC (w tym później wsparcie dla nowej usługi: Google Wallet), wsparcie dla nowych sensorów: żyroskopu i barometru, współbieżny odśmiecacz pamięci (ang. *Garbage Collector*), a także usprawnienia klawiatury, menadżera pobierania, funkcji kopiuj-wklej i wielu innych. Wraz z wersją został wydany kolejny telefon z rodziny Nexus - Nexus S produkowany przez firmę Samsung.

Honeycomb, Ice Cream Sandwich

W odpowiedzi na powszechny tabletowy szał, w lutym 2011 wydano wersję 3.0 Androida (Honeycomb) opartą o jądro 2.6.36 i przeznaczoną tylko na tablety. Urządzeniem pilotażowym był tablet Xoom

firmy Motorola. W późniejszym czasie wydano kolejne jej iteracje, wersje 3.1 i 3.2 (również Honeycomb). Główną modyfikacją było dostosowanie interfejsu do urządzeń posiadających większy ekran, ale dodatkowo zmieniono GUI, dodano wsparcie dla przyspieszenia sprzętowego, wsparcie dla procesorów wielordzeniowych, wsparcie dla szyfrowania urządzenia, wsparcie dla akcesoriów USB, możliwość zmiany rozmiarów widgetów, dodano pasek akcji (ang. *Action Bar*) i pasek systemowy (ang. *System Bar*), usprawniono multitasking, klawiaturę, aparat i wiele innych elementów. Wersja ta nie została nigdy udostępniona w postaci kodu źródłowego ogółowi, gdyż Google utrzymywało, że nie jest gotowa na taki krok.

Od wydania wersji 3.x trwały prace, aby z powrotem połączyć podzielone gałęzie systemu w jedno. W taki sposób powstała wydana w październiku 2011 wersja 4.0.1 (Ice Cream Sandwich) systemu oparta na jądrze 3.0.1. Główne dodatkowe zmiany to kolejna modernizacja GUI (w tym dodanie wsparcia dla przycisków wirtualnych), wsparcie dla tworzenia zrzutów ekranu, nowa przeglądarka, wbudowany edytor zdjęć, wsparcie dla WiFi Direct, wsparcie dla nagrywania 1080p i szereg standardowych usprawnień dla aparatu, kopiuj-wklej, itp. Ice Cream Sandwich był tworzony z użyciem produkowanego przez firmę Samsung telefonu Galaxy Nexus.

2.2.4. Problemy i wyzwania

Będąc bardzo przyjazną platformą dla programistów, system Android nie uniknął niestety wpadek. Poza szeregiem problemów typowych dla wszystkich urządzeń tego typu (rozdział 2.1.3) istnieje też kilka minusów tego konkretnego systemu. Główną jego wadą jest jednocześnie jego główna siła - mnogość urządzeń i wersji. Producenci telefonów i operatorzy sieci mają tendencje do modyfikacji oprogramowania na własne potrzeby co powoduje później opóźnianie się aktualizacji, a w rezultacie fragmentacje (tabela 2.1) [26].

Mając dany tak wielki rozrzut wersji deweloperom ciężko jest pisać oprogramowanie, które będzie działało na każdej z nich. Sprawy nie ułatwia istnienie setek różnego rodzaju urządzeń, gdyż nawet teoretycznie ta sama wersja systemu okazuje się działać różnie na różnych telefonach. Powoduje to, że dokładne przetestowanie aplikacji jest właściwe niemożliwe. Sytuacja taka nie ma miejsca na przykład na telefonie iPhone, gdzie mamy do czynienia z garstką urządzeń dostarczanych przez jednego producenta.

2.3. Android jako platforma deweloperska

Firma Google Inc. zdawała sobie sprawę z potencjału szeroko rozwiniętego wsparcia zewnętrznych programistów. Z tego powodu system Android od początku był projektowany pod kątem przystosowania do obsługi programów trzecich. Rozbudowane narzędzia oraz przyjazne API były głównym celem twórców mobilnego systemu od firmy Google. Jednym z dowodów na to był fakt, że pierwsza publicznie wydana w drugiej połowie 2007 roku wersja Androida (1.0 *Apple Pie*) została udostępniona mimo tego, że pierwszy komercyjny telefon z tym systemem pojawił się na rynku dopiero rok później. Nie da się ukryć, że był to krok w dobrą stronę - mimo że jedyną możliwością uruchamiania skompilowanego kodu były emulatory, a początkowe narzędzia nie były wolne od błędów, to już tydzień po wydaniu SDK w

Wersja systemu	Nazwa kodowa	Wersja API	Udział
1.5	Cupcake	3	0.3%
1.6	Donut	4	0.6%
2.1	Eclair	7	5.2%
2.2	Froyo	8	19.1%
2.3 - 2.3.2	Gingerbread	9	0.4%
2.3.3 - 2.3.7		10	64.6%
3.1	Honeycomb	12	0.7%
3.2		13	2%
4.0 - 4.0.2	Ice Cream Sandwich	14	0.4%
4.0.3 - 4.0.4		15	6.7%

Tablica 2.1: Fragmentacja systemu Android

wersji 1.0 pojawił się pierwszy gotowy program. Dzięki temu, gdy na rynku pojawił się pierwszy telefon z systemem, w sklepie z aplikacjami znajdowała się już całkiem pokaźna liczba pozycji.

Oprogramowanie pod system Android może być tworzone na dwa sposoby (które w ramach jednej aplikacji można ze sobą dowolnie łączyć i przeplatać): na poziomie "wysokim" (pisanie programu z kompilacją do bajtkodu wirtualnej maszyny), i na poziomie niskim (kompilacja do natywnych bibliotek binarnych).

2.3.1. Programowanie wysokopoziomowe

Programowanie wysokopoziomowe na Androidzie opiera się na maszynie wirtualnej wykonującej bajtkod (ang. *bytecode*). Pisanie aplikacji jest dużo prostsze, a do wyboru jest wiele bibliotek pomocniczych. Ma to oczywiście swoją cenę - jest nią (potencjalnie) niższa wydajność. W rzeczywistości jednak, w zdecydowanej większości przypadków spadek wydajności jest niezauważalny i jak informuje oficjalna dokumentacja, korzystanie z niskiego poziomu opłaca się bardzo rzadko [26].

Dalvik VM

Językiem programistycznym używanym do tworzenia oprogramowania na wysokim poziomie platformy jest Java. Głównym powodem takiego wyboru jest fakt, że w czasie projektowania systemu język ten był (a także dalej jest) najbardziej rozpowszechnionym językiem programowania na rynku. Użycie go miało ułatwić programistom przejście od pisania aplikacji desktopowych do aplikacji mobilnych.

Jedną z głównych różnic pomiędzy Javą dla Androida, a Javą na PC jest wybór wirtualnej maszyny wykonującej bajtkod. W celu uniknięcia opłat licencyjnych z tytułu użytkowania Wirtualnej Maszyny Javy (ang. *Java Virtual Machine*, *Java VM*) - oprogramowania stworzonego przez firmę Sun (która w międzyczasie została wykupiona przez firmę Oracle), firma Google podjęła decyzje o stworzeniu od podstaw własnego produktu. Tak powstała Wirtualna Maszyna Dalvik (ang. *Dalvik Virtual Machine*, *Dalvik VM*). Obie aplikacje różnią się od siebie, ale większość tych różnic występuje w implementacji

(budowie wewnętrznej) i formacie bajtkodu. Z punktu widzenia programisty są one praktycznie niezauważalne - składnia języka, a także praktycznie wszystkie ogólnodostępne biblioteki pomocnicze z Java VM są dostępne w takiej samej postaci pod Dalvik VM.

Poza użyciem Javy, przy tworzeniu aplikacji używa się wielu rodzajów plików XML, które odpowiadają za opis komponentów programu (takich jak layout, struktury menu, zbiory komunikatów i etykiet). Generalnie, mimo że praktycznie każda część aplikacji może zostać zbudowana czysto programistycznie (to znaczy, z użyciem jedynie wywołań metod w kodzie programu), zaleca się aby te elementy były definiowane osobno, w formie tych właśnie dokumentów. Pozwala to na wyraźniejsze oddzielenie od siebie dwóch różnych części aplikacji - logiki działania i struktury, co upraszcza proces tworzenia i utrzymywania programu [26].

Android SDK

Android SDK (ang. *Software Development Kit*) jest określeniem obejmującym zestaw narzędzi i bibliotek służących do budowy oprogramowania. Obejmuje ono takie aplikacje jak kompilator kodu źródłowego do bajtkodu Dalvik VM, serwer monitorujący wirtualną maszynę *DDMS* (ang. *Dalvik Debug Monitor Server*), czy *ADB* (ang. Android Debug Bridge) - aplikacje umożliwiającą komunikacje pomiędzy telefonem, a komputerem na którym tworzony jest kod, w tym: przesyłanie i instalowanie plików `.apk` (aplikacji), nasłuchiwanie logów, wykonywanie komend shellowych i podłączanie zdalnego debuggera.

Wraz z programami wspierającymi budowę aplikacji dostarczane są wraz z SDK zewnętrzne biblioteki. Poza standardowym zbiorem pakietów znanych z desktopowej wersji Javy (takich jak `java.lang.*`, czy `java.util.*`), dostajemy do dyspozycji pokaźny zbiór bibliotek ściśle związanych ze środowiskiem Android. Składają się na nie między innymi biblioteki do tworzenia GUI (ang. *Graphical User Interface*) aplikacji, biblioteki definiujące strukturę i cykl życia (ang. *lifecycle*) aplikacji, a także w końcu biblioteki do obsługi komponentów sprzętowych nie występujących powszechnie w komputerach klasy PC, a znajdujących się w większości urządzeń mobilnych - na przykład obsługa czujników (światła, pozycji, żyroskopu) lub telefonii komórkowej.

Środowisko Android SDK jest dostępne do pobrania za darmo na stronie projektu Android. Aktualną wersją jest r18 (dla systemu w wersji 4.0 *Ice Cream Sandwich*). Produkt jest dostępny dla każdego z popularnych systemów operacyjnych - Windows, Linux i Mac OS, a aktualizacje wychodzą wraz z nowymi wersjami systemu operacyjnego średnio co pół roku [26].

Zintegrowane środowisko programistyczne (IDE)

Co prawda z użyciem Android SDK możliwe jest tworzenie aplikacji za pomocą dowolnego edytora tekstowego (podobnie jak za pomocą Java SDK możemy tworzyć aplikacje w ten sam sposób), nie należy jednak do tajemnic, że proces kreacji oprogramowania jest dużo szybszy i prostszy mając dostęp do zintegrowanego środowiska programistycznego (ang. *Integrated Development Environment*, *IDE*). Przy tworzeniu programów pod Androida, oficjalnie wspieranym środowiskiem jest Eclipse. Dobór takiego, a nie innego IDE był prawdopodobnie związany z dwoma faktami: po pierwsze środowisko to jest szeroko stosowane przez programistów Javy, co po raz kolejny umożliwia im możliwe jak najprostsze przej-

ście z tworzenia aplikacji desktopowych do tworzenia aplikacji mobilnych. Po drugie Eclipse udostępnia bardzo rozbudowany system wtyczek (ang. *plugins*), który umożliwia praktycznie nielimitowane rozszerzanie jego funkcjonalności, a co za tym idzie dopasowanie procesu tworzenia aplikacji do wymagań Android SDK.

Wtyczką wspomagającą tworzenie aplikacji Androidowych z użyciem programu Eclipse jest ADT (ang. *Android Development Tools*). Wykorzystuje ona narzędzia oferowane przez Android SDK (które musi zostać zainstalowane osobno) do budowy aplikacji, podpisywania jej, zdalnego instalowania na telefonie/emulatorze i wielu innych rzeczy. Proces tworzenia aplikacji mobilnych w dużej mierze opiera się na automatycznym generowaniu klas, opartych na zdefiniowanych przez użytkownika plikach XML. Tego rodzaju zadania wykonywane są automatycznie przez ADT i są praktycznie niezauważalne dla programisty. Wtyczka dostarcza też zestaw edytorów wizualnych plików XML, a także prosty edytor GUI typu WYSIWYG (ang. *What You See Is What You Get*). Oczywiście, cały czas są dostępne wszystkie funkcjonalności oczekiwane po dzisiejszych IDE, począwszy od kolorowania składni, na autouzupełnianiu kończąc [26].

2.3.2. Programowanie niskopoziomowe

Poza wysokopoziomowymi możliwościami programowania urządzeń z systemem Android, możliwe jest zejście poziom niżej i kompilacja części programów do kodu binarnego. Ten sposób tworzenia aplikacji jest odradzany w większości przypadków, istnieją jednak momenty, gdy warto skorzystać z tej możliwości. Oficjalna dokumentacja systemu Android wskazuje dwie takie sytuacje: po pierwsze, gdy istnieje duża ilość gotowego kodu stworzonego w C/C++, którego przepisywanie mija się z celem. Po drugie, gdy wykonujemy dużą ilość specjalistycznych obliczeń (na przykład w grach), gdzie zejście na niższy poziom umożliwia dokładniejszą optymalizacje. Zestaw narzędzi, który umożliwia budowanie aplikacji w taki sposób nazywa się Android NDK (ang. *Native Development Kit*) [26].

JNI

U podstaw NDK stoi technologia znana z wirtualnej maszyny Javy: JNI (ang. *Java Native Interface*). Umożliwia ona zdefiniowanie w ramach klasy w Javie interfejsu dostępowego do zewnętrznych bibliotek. Biblioteki mogą być utworzone w dowolnym języku, muszą zostać jednak skompilowane pod system, na którym jest wykonywana maszyna wirtualna Javy, w sposób zrozumiały dla tego systemu (czyli na przykład jako biblioteki `*.dll` pod Windowsem lub `*.so` pod Linuksem). Jeśli powyższe warunki są spełnione, możemy w bardzo prosty sposób podłączyć tą bibliotekę do naszego programu skompilowanego do bajtkodu Java VM.

Android NDK

Android NDK, mimo że oparty jest w dużym stopniu na JNI, różni się trochę możliwościami. Po pierwsze, mimo że formalnie nie ma ograniczeń na język, w którym powinna zostać napisana zewnętrzna biblioteka, w praktyce wymagane jest aby była ona skompilowana do kodu maszynowego danej platformy. Na ten moment, jedyne kompilatory tworzące kod maszynowy pod użytek urządzeń mobilnych Android to kompilatory języków C i C++. Dodatkowo, system udostępnia tylko pewien podzbiór ze-

wnętrznych bibliotek systemowych dla języka C w stosunku do tego co oferuje system Linux, a wsparcie dla bibliotek C++ jest jeszcze mniejsze. Widać więc, że technologia NDK, mimo że często bywa przydatna, jest mocno ograniczona, jako że wymaga kodu źródłowego napisanego w C lub C++ i korzystającego tylko z mocno ograniczonego zbioru bibliotek zewnętrznych.

Budowa natywnych bibliotek pod system Android jest możliwa dzięki zestawowi narzędzi udostępnianych w ramach NDK. Kompilacja opiera się na plikach `Makefile`, przy czym w związku ze skomplikowanym procesem budowania NDK używa swojego własnego formatu pliku: `Android.mf`. Definiuje się w nim podstawowe zależności (używane biblioteki i pliki nagłówkowe, pliki źródłowe konieczne do kompilacji) oraz konstrukcje sterujące (na przykład czy kompilacja ma się odbywać do biblioteki dynamicznej, czy statycznej). Każdy projekt (biblioteka) wchodząca w skład aplikacji musi mieć zdefiniowany plik tego typu, nawet jeśli jest już prekompilowana. Narzędzia budujące NDK wczytują przy tworzeniu aplikacji pliki `Android.mf` wymaganych bibliotek i na ich podstawie generują pliki `Makefile`, które następnie są poddane standardowemu procesowi przetwarzania, z tą różnicą, że kompilator tłumaczy kod języka na kod maszynowy danego urządzenia z Androidem. Po zbudowaniu, biblioteki są automatycznie kopiowane do odpowiednich katalogów projektu, gdzie mogą zostać wykryte przez narzędzia SDK, które przy tworzeniu pliku `*.apk` aplikacji, umieszczą je we właściwym miejscu. Dzięki tej automatyzacji proces dodawania zewnętrznych bibliotek jest stosunkowo prosty [26].

3. Algorytm sita kwadratowego

Algorytm sita kwadratowego jest jedną z najszybszych znanych metod na faktoryzację liczb całkowitych. Mimo, że w latach 90 oddał palmę pierwszeństwa metodzie *GNFS*, wciąż znajduje się na drugim miejscu, a dla liczb których rozmiar nie przekracza 150 cyfr w dalszym ciągu okazuje się być najlepszy.

Twórcą algorytmu jest Carl Pomerance, który opracował jego zasady na podstawie analizy złożonościowej algorytmów z rodziny Kraitchika autorstwa Shroeppela. Idea została zaproponowana w 1981 roku [2], a dokładne jej przedstawienie odbyło się w roku 1982 [27].

3.1. Powstanie algorytmu sita kwadratowego

3.1.1. Analiza Shroeppela

Mimo, że algorytm *CFRAC* powstał w 1931 roku, przez długi czas nie znano jego złożoności obliczeniowej. Dopiero w późnych latach 70 XX w. Richard Shroeppel w nieopublikowanej korespondencji dokonał pierwszej analizy złożoności obliczeniowej algorytmów z *rodziny Kraitchika* [2]. Okazało się następnie, że poza oczywistym zyskiem w postaci poznania asymptotycznego czasu działania algorytmów już istniejących, analiza ta pozwoliła opracować nowe metody faktoryzacji działające jeszcze szybciej.

Istotą analizy Shroeppela jest dobór parametru B, w celu wybrania liczb B-gładkich. Rzeczywiście, im mniejsze B obierzemy na początku, tym trudniej będzie nam znaleźć liczby, które rozkładają się na czynniki pierwsze mniejsze od B. Z drugiej strony obranie zbyt dużej wartości granicznej powoduje, że rozmiar naszej bazy czynników pierwszych wzrasta, a więc potrzebujemy znaleźć więcej liczb B-gładkich, co spowalnia działanie algorytmu zarówno poprzez wydłużenie wyszukiwania par, jak i wydłużenie działania algorytmu rozwiązującego układ równań liniowych (gdyż wzrasta zarówno ilość kolumn, jak i wierszy macierzy). Jest zatem jasne, że parametr B potrzebujemy wybrać *optymalnie*, aby zminimalizować czas działania.

Shroeppel zaczyna od zdefiniowania funkcji $\Psi(X, B)$: jest to ilość liczb całkowitych nie większych od X, które sfaktoryzowane nie posiadają czynników pierwszych większych niż B. Innymi słowy, jest to ilość liczb B-gładkich, mniejszych bądź równych X. W takim razie, zakładając losowy wybór liczb z przedziału $[1, X]$, prawdopodobieństwo trafienia na liczbę B-gładką, wynosi:

$$\frac{\Psi(X, B)}{X} \tag{3.1}$$

Ponieważ jak wiadomo istnieje w przybliżeniu $F \approx \frac{B}{\ln B}$ liczb pierwszych mniejszych bądź równych od B, zatem dla układu równań liniowych potrzebujemy mniej więcej tyle samo "wylosowanych" liczb B-gładkich (Właściwie, par liczbowych spełniających równanie (1.16)). Ponieważ wartość oczekiwana liczby "losowań" prowadzących do uzyskania jednej liczby B-gładkiej jest odwrotnością prawdopodobieństwa wylosowania takiej liczby, otrzymujemy że będziemy musieli sprawdzić mniej więcej

$$F \cdot \left(\frac{\Psi(X,B)}{X} \right)^{-1} = \frac{FX}{\Psi(X,B)} \tag{3.2}$$

liczb. Rozważmy teraz algorytm *CFRAC*. Sprawdzenie każdej liczby pod kątem B-gładkości przy użyciu metody naiwnej, wymaga przeprowadzenia tylu próbnych dzieleń, ile mamy liczb w bazie czynników pierwszych, a więc F. Otrzymujemy ostatecznie, że algorytm potrzebuje

$$\frac{F^2 X}{\Psi(X,B)} \tag{3.3}$$

kroków elementarnych (właściwie tyle kroków potrzebuje jego część związana z wyszukiwaniem liczb B-gładkich, ale jako że jest to dominująca składowa algorytmu *CFRAC* na potrzeby tej analizy możemy pominąć resztę). Pozostaje nam wyznaczyć funkcję $\Psi(X,B)$. Shroeppel założył, że

$$\frac{\Psi(X, X^{\frac{1}{u}})}{X} = u^{-(1+o(1))u} \tag{3.4}$$

gdzie

$$(\log X)^{\varepsilon} < u < (\log X)^{1-\varepsilon} \tag{3.5}$$

które to oszacowanie zostało w późniejszym czasie dowiedzione w [28]. Dla algorytmu *CFRAC* wiemy także, że $X \approx \sqrt{n}$, gdzie n jest faktoryzowaną liczbą (patrz rozdział 1.2.2). Korzystając z tych danych, okazało się że optymalny wybór granicy B to

$$L(n)^{\frac{1}{\sqrt{8}}+o(1)} \tag{3.6}$$

gdzie

$$L(n) = e^{\sqrt{\ln n \ln \ln n}} \tag{3.7}$$

a oczekiwany czas działania jest wtedy rzędu

$$L(n)^{\sqrt{2}+o(1)} \tag{3.8}$$

Oczywiście, należy zwrócić uwagę, że powyższe wyliczenia są *heurystyczne* - dla algorytmu *CFRAC* zakładamy, że kolejne sprawdzane współczynniki posiadają taki sam rozkład liczb B-gładkich jak liczby losowe. Późniejsze doświadczenia empiryczne potwierdziły jednak poprawność powyższej analizy.

3.1.2. Idea sita

Obserwując analizę złożonościową algorytmów z rodziny Kraitchika można zastanowić się które jej fragmenty nadają się do optymalizacji. Oczywistym pomysłem jest zwiększenie prawdopodobieństwa znalezienia odpowiednich par - czy to za pomocą użycia sekwencji, w której liczby B-gładkie występują częściej, czy też rozluźnienie warunku B-gładkości. Niestety, tego typu optymalizacja jest bardzo trudna. Istnieje jednak inny fragment, który nadaje się do usprawnienia: można zauważyć, że dla każdej liczby którą testujemy pod kątem B-gładkości, wykonujemy co najmniej F dzieleń. Jest to zdecydowana strata czasu, jako że jest jasnym że większość testowanych wartości nie jest B-gładka. Okazuje się jednak, że można zoptymalizować ten proces.

Jednym z elementarnych algorytmów teorii liczb jest sito Eratostenesa. Algorytm ten służy do znajdywania wszystkich liczb pierwszych w danym przedziale. Mając zadany przedział, zaczynamy od pierwszej liczby - 2 i wykreślamy kolejne jej wielokrotności z sita. Następnie przechodzimy do następnej nieskreślonej jeszcze liczby i wykreślamy jej wielokrotności. Powtarzamy tą czynność, dopóki nie dojdziemy do $\left\lceil \sqrt{M} \right\rceil$, gdzie M jest górną granicą naszego przedziału. W tym momencie, wszystkie nieskreślone liczby w sicie są liczbami pierwszymi.

Dość szybko można zauważyć następującą zależność: jeśli zamiast po prostu skreślać liczby złożone będziemy dzielić je przez liczby przez które akurat przesiewamy (a także przez ich potęgi), po zakończeniu otrzymamy wartości 1 w komórkach, które całkowicie rozłożyły się na używane czynniki - są więc on gładkie. Taki sposób szukania jest zdecydowanie dużo szybszy niż dzielenie każdej liczby przez wszystkie potencjalne czynniki. Gdy ustalimy granice gładkości B, otrzymujemy w ten sposób F - ilość liczb pierwszych, przez które będziemy przesiewać. W związku z tym naiwny sposób testowania liczb z przedziału $[1, M]$ wymaga od nas

$$MF \approx \frac{MB}{\ln B} \tag{3.9}$$

elementarnych kroków, czyli

$$F \approx \frac{B}{\ln B} \tag{3.10}$$

kroków średnio na liczbę. Z kolei używając idei sita musimy wykonać mniej więcej

$$\sum_{i=1}^{F} \frac{M}{p} = M \sum_{i=1}^{F} \frac{1}{p} \approx M \ln \ln B \tag{3.11}$$

operacji (gdzie ostatnie przejście wynika bezpośrednio z dolnego ograniczenia szeregu harmonicznego liczb pierwszych) albo, innymi słowy, jedynie

$$\ln \ln B \tag{3.12}$$

kroków na liczbę po uśrednieniu.

Okazuje się, że przedstawioną idee da się rozszerzyć. Poprawność sita Eratostenesa wynika z faktu, że liczby w sicie są ułożone regularnie, to znaczy zgodnie z ciągiem arytmetycznym $f(n) = n$. Stąd wysnuwaliśmy wniosek, że

$$f(n_0) = n_0 \equiv 0 \pmod{p} \Rightarrow f(n_0 + mp) = n_0 + mp \equiv 0 \pmod{p} \tag{3.13}$$

Można sprawdzić, że powyższa zależność zachodzi nie tylko dla ciągu $f(n) = n$, ale zawsze, gdy ciąg f ma postać wielomianu. Dla przykładu, jeśli określimy ciąg jako funkcje kwadratową

$$f(n) = an^2 + bn + c \tag{3.14}$$

i wiemy, że

$$f(n_0) \equiv 0 \pmod{p} \tag{3.15}$$

to dokonując prostych przekształceń otrzymujemy

$$\begin{aligned} f(n_0 + mp) &= a(n_0 + mp)^2 + b(n_0 + mp) + c = \\ &= an_0^2 + bn_0 + c + p(2an_0m + am^2p + bm) = \\ &= f(n_0) + p(2man_0 + am^2p + bm) \equiv 0 \pmod{p} \end{aligned} \tag{3.16}$$

Zademonstrowana powyżej idea przesiewania w celu poszukiwania liczb B-gładkich stoi u podstaw algorytmu sita kwadratowego.

3.2. Algorytm

3.2.1. Schemat działania

Ogólny schemat działania algorytmu przedstawia się zatem następująco: wybieramy pewną wartość M i określamy przedział $[-M, M]$. Następnie definiujemy funkcje

$$Q(x) = \left(x + \left\lfloor \sqrt{n} \right\rfloor\right)^2 - n \tag{3.17}$$

Zauważmy oczywiście, że oznaczając

$$\tilde{x} = x + \left\lfloor \sqrt{n} \right\rfloor \tag{3.18}$$

otrzymujemy

$$Q(x) = \tilde{x}^2 - n \equiv \tilde{x}^2 \pmod{n} \tag{3.19}$$

Taka, a nie inna postać funkcji Q bierze się z faktu, że podobnie jak w algorytmie *CFRAC* zależy nam aby jej wartości były jak najmniejsze, jako że mamy wtedy większą szansę, że będą one B-gładkie.

Mając dany przedział określony przez M i ciąg Q na tym przedziale będący wielomianem drugiego stopnia, możemy skorzystać z przedstawionego w rozdziale 3.1.2 schematu przesiewania w poszukiwaniu liczb B-gładkich. W taki sposób otrzymujemy zbiór par $\tilde{x}, Q(x)$, gdzie $x \in [-M, M]$, które spełniają

(1.16). Jeśli otrzymaliśmy ich wystarczająco dużo, możemy kontynuować algorytm - jego dalsza część nie różni się od żadnej innej metody z rodziny Kraitchika.

Można zaobserwować jeden potencjalnie słaby punkt naszej metody. Dla pewnych wartości (dokładnie dla $x < \lfloor\sqrt{n}\rfloor$) otrzymane wyrazy ciągu Q(x) są ujemne. Okazuje się, że nie stanowi to jednak problemu. Wystarczy potraktować -1 jako dodatkowy "czynnik" w naszej bazie. Sprawdzenie czy $Q(x)$ zawiera -1 sprowadza się do sprawdzenia znaku, a na etapie szukania kombinacji par (schemat Kraitchika) liczba po prawej stronie równania będzie kwadratem, wtedy i tylko wtedy, gdy wykładnik przy tym czynniku będzie parzysty.

3.2.2. Opis algorytmu

Można przejść teraz do pełnego opisu algorytm sita kwadratowego.

- **Wejście:** n - nieparzysta liczba złożona, nie będąca potęgą liczby całkowitej
- **Wyjście:** n_1, n_2 - dwa nietrywialne czynniki n

1. Wyznacz wartości: B - granica liczb B-gładkich i M - granica przedziału w ramach którego będziemy działać w czasie przesiewania.
2. Stwórz bazę czynników pierwszych, na podstawie B
3. Stwórz tabele S indeksowaną od $-M$ do M i wypełnij ją aby w miejscu $S[x]$ była wartość $Q(x)$
4. Dla każdej z liczb p należącej do bazy czynników wykonaj następujące kroki:
   4.1 Rozwiąż równanie kwadratowe $\tilde{x}^2 \equiv n \pmod p$ otrzymując rozwiązanie (rozwiązania) $\tilde{x}_i$.
   4.2 Dla każdego $x \in [-M, M]$, dla którego jest spełnione $x \equiv \tilde{x}_i \pmod p$ wyciągnij z $S[x]$ maksymalną potęgę p
5. Zapamiętaj wszystkie pozycje x, gdzie $S[x] = 1$
6. Stwórz macierz A o rozmiarze $k \times l$, gdzie k jest liczbą czynników w bazie, a l liczbą uzyskanych par
7. Sfaktoryzuj otrzymane w punkcie 5 pary, tworząc dla każdej wektor wykładników $\bar{v} \pmod 2$, a następnie zapisując ten wektor w odpowiedniej kolumnie macierzy A
8. Korzystając z macierzy A, rozwiąż równanie $A \times \bar{w} = \bar{0}$, otrzymując wektor (wektory) $\bar{w}$
9. Dla każdego z uzyskanych w punkcie 8 rozwiązań, zbuduj

$$\begin{aligned} a &= \prod_i w_i \tilde{x_i} \\ b &= \prod_i w_i Q(x_i) \end{aligned} \tag{3.20}$$

   gdzie w_i jest i-tym elementem wektora rozwiązania (i wynosi 0 lub 1).

10. Na podstawie wcześniejszych rozważań wiemy, że $a^2 \equiv b^2 \pmod{n}$. Sprawdź zatem czy $a \equiv \pm b \pmod{n}$. Jeśli tak jest, ta para do niczego nam się nie przyda: wróć do punktu 9 i wybierz następne z rozwiązań.

11. W przeciwnym wypadku, przypisz $n_1 = NWD(a-b, n)$ i $n_2 = NWD(a+b, n)$ i zakończ algorytm

Kilka kroków powyższego schematu wymaga dokładniejszych wyjaśnień.

W punkcie 1 wyznaczenie odpowiednich wartości wiąże się z analizą złożoności obliczeniowej, dokładny więc wzór zostanie podany w rozdziale 3.2.3.

W punkcie 2 można użyć dowolnego algorytmu, na przykład sita Eratostenesa. Należy przy tym zwrócić uwagę, że w przyszłości będziemy próbowali rozwiązywać równanie postaci $x^2 \equiv n \pmod{p}$, zatem dla każdego czynnika w bazie p powinna istnieć reszta kwadratowa z $n \pmod{p}$. W praktyce łatwo to sprawdzić obliczając symbol Legendre'a $(\frac{n}{p})$ i sprawdzając czy symbol ten wynosi 1.

W punkcie 4.1 do rozwiązania równania można użyć algorytmu Shanksa-Tonellego. Oczywiście, dzięki odpowiedniemu zbudowaniu bazy czynników mamy pewność, że istnieją zawsze dwa rozwiązania. Specjalnym (aczkolwiek prostym) przypadkiem jest liczba 2, kiedy w związku ze specyfikacją danych wejściowych (n jest nieparzyste) zawsze istnieje tylko jedno rozwiązanie: 1.

Do rozwiązania macierzy w punkcie 8 można użyć na przykład metody eliminacji Gaussa, aczkolwiek jak zostanie to wyjaśnione w rozdziale 3.3.3 zalecane jest użycie szybszego algorytmu.

W powyższym schemacie nie uwzględniono specjalnych sytuacji, gdy skończymy przesiewanie przedziału nie otrzymując odpowiedniej liczby par $(F+1)$ lub gdy skończą nam się rozwiązania w punkcie 9. W drugim przypadku dobieramy kolejną parę (o ile jeszcze istnieje) i dołączamy ją do naszej macierzy, a następnie przechodzimy do punktu 8. W przypadku pierwszym (bądź drugim, jeśli nie mamy już więcej dodatkowych par) powtarzamy przesiewanie na jednym z sąsiednich przedziałów i przechodzimy do punktu 7.

3.2.3. Analiza złożoności obliczeniowej

Należy udowodnić, że nasz algorytm działa zgodnie z zapowiedzią wydajniej niż dotychczasowe algorytmy z rodziny Kraitchika. Zdecydowana większość analizy Shroeppela pozostaje bez zmian, jedynie czas spędzony na test pojedynczej liczby zostaje mocno skrócony. Tak jak i w innych algorytmach z rodziny, wybór wartości B i M jest kluczowy dla optymalizacji działania metody. Okazuje się, że dla sita kwadratowego, optymalne wartości to

$$F = L(n)^{\frac{\sqrt{2}}{4}} = (e^{\sqrt{\ln n \ln\ln n}})^{\frac{\sqrt{2}}{4}} \tag{3.21}$$

gdzie F jest rozmiarem bazy czynników, a więc jest zależne od B, a M okazuje się być sześcianem tej wartości:

$$M = L(n)^{\frac{3\sqrt{2}}{4}} = (e^{\sqrt{\ln n \ln\ln n}})^{\frac{3\sqrt{2}}{4}} \tag{3.22}$$

Dla tak dobranych parametrów, czas działania algorytmu wynosi mniej więcej

$$O(e^{\sqrt{\frac{9}{8}\ln n \ln\ln n}}) \tag{3.23}$$

A więc wykładnik jest mniejszy o czynnik $\sqrt{\frac{16}{9}}$ w stosunku do CFRAC. Ta pozornie niewielka zmiana, a także kilka opisanych w kolejnych punktach usprawnień sprawiają, że algorytm sita kwadratowego działa dużo szybciej niż swój poprzednik - *CFRAC*.

3.3. Modyfikacje i usprawnienia

Algorytm sita kwadratowego był obiektem wielu badań, które doprowadziły do usprawnień jego działania. Mimo, że żadna z optymalizacji nie zmienia jego złożoności obliczeniowej radykalnie, należy wspomnieć o tych najważniejszych, które zauważalnie go przyspieszają.

3.3.1. Użycie przybliżonych logarytmów

Pierwszą modyfikacją (towarzyszącą metodzie sita kwadratowego właściwie od początku) jest zmiana typu elementarnego kroku przy przesiewania przedziału. Załóżmy, że rozważamy pozycje x w przedziale, której odpowiada wartość $Q(x)$. Na podstawie podstawowego twierdzenia arytmetyki (patrz rozdział 1.1.2) wiemy, że

$$Q(x) = \prod_i p_i^{\alpha_i} \tag{3.24}$$

Korzystając z właściwości logarytmu przekształcamy powyższe równanie następująco

$$\log_2 Q(x) = \log_2 \prod_i p_i^{\alpha_i} = \sum_i \alpha_i \log_2 p_i \tag{3.25}$$

Możemy zatem w naszej tablicy przechowywać nie wartości $Q(x)$, a wartości $log_2(Q(x))$ i w trakcie przesiewania zamiast dzielić przez kolejne liczby pierwsze - odejmujemy ich logarytmy. Na końcu szukamy wtedy takich komórek w przedziale, gdzie wartość wynosi 0 (czyli wartość oryginalnej komórki to $2^0 = 1$). Mając dane takie pola możemy odtworzyć oryginalne $Q(x)$ i za pomocą algorytmu naiwnego wyznaczyć jego wektor wykładników. Oczywiście, otrzymanych liczb B-gładkich jest na tyle mało, że używane w tym miejscu naiwne dzielenie ma minimalny narzut. Naturalnie są minusy takiego postępowania. Po pierwsze, nie możemy już wyciągać potęg czynników pierwszych z liczb $Q(x)$, zamiast tego musimy przesiewać przedział przez wyższe potęgi p_i. Nie jest to jednak duży problem, gdyż całościowo wykonujemy tyle samo kroków przy przesiewaniu (wykonamy tyle samo odejmowań co dzieleń), a jedyny narzut jest związany z wyznaczaniem reszt kwadratowych $(\mathrm{mod}\ p_i^k)$. Jak się zresztą okaże później - rozdział 3.3.2 - także ten krok może zostać pominięty. Drugim minusem jest konieczność wyznaczania logarytmów z liczb $Q(x)$ i z czynników pierwszych p_i. Wyliczanie dokładnego logarytmu jest zajęciem dość czasochłonnym i mogłoby zniwelować korzyści płynące z ich używania. Aby zapobiec temu, stosuje się następującą metodę: zamiast wyliczania dokładnych logarytmów wylicza się ich wartości przybliżone, czyli ilość cyfr binarnych, które posiada dana liczba - działanie takie na komputerze

trwa bardzo szybko. Oczywiście, wprowadzamy wtedy pewien błąd zaokrąglenia, dlatego po skończeniu przesiewania szukamy wszystkich komórek, w których logarytm jest *bliski* zera (a nie mu równy).

Co zyskujemy przy tej modyfikacji? Po pierwsze, elementarny krok naszego działania to teraz *odejmowanie* liczby całkowitej, a nie *dzielenie* takiej liczby, co dość mocno przyspiesza obliczenia zarówno w stosunku do oryginalnej wersji sita kwadratowego, jak i do algorytmu *CFRAC*. Po drugie, przechowując jedynie *logarytmy* liczb, nie musimy przy przesiewaniu używać liczb całkowitych o dowolnej precyzji: dla dużych n, wartości $Q(x)$ wykraczają poza standardowe typy całkowite obecne na komputerach i musimy sięgać po zewnętrzne biblioteki. Użycie logarytmu z $Q(x)$ niweluje tą potrzebę, a zatem obliczenia są prostsze i szybsze (bo liczby na których są przeprowadzane są krótsze), a także (co również jest bardzo istotne) używane sito zajmuje dużo mniej miejsca w pamięci.

Wykorzystanie przybliżonych logarytmów niesie za sobą praktycznie same korzyści, a minusy tej modyfikacji właściwie nie istnieją. Dlatego praktycznie zawsze gdy mówi się o algorytmie sita kwadratowego, ma się na myśli sito kwadratowe w tej wariacji.

3.3.2. Ograniczenie przesiewania

Można zastanowić się, czy musimy zawsze przesiewać liczby przez wszystkie czynniki p i ich potęgi. Dla przykładu, używając metody naiwnej sprawdzenie czy dana liczba jest podzielna przez 2, czy przez 103 zajmuje tyle samo czasu. Z drugiej strony, przesiewanie przez 2 w algorytmie sita kwadratowego zajmuje około $\frac{103}{2}$ razy więcej czasu niż przesiewanie przez 103. Dodatkowo przesiewanie przez 2 daje nam najmniej informacji, jako że odejmowany logarytm jest bardzo mały. Z tego powodu często stosuje się następującą metodę: przesiewamy sito jedynie przez czynniki (bądź ich potęgi), które są większe niż pewna ustalona wartość graniczna, powiedzmy 30. Na przykład nie przesiewamy w takim wypadku przez 3, ani przez 9, ani przez 27, natomiast przesiewając przez $3^4 = 81$ odejmujemy $4 \log_2 3$. Oczywiście, musimy wtedy ostrożnie dobrać wartość graniczną "bliskości" otrzymanych wyników do 0 podczas przeglądania tablicy po jej przesianiu. Okazuje się jednak, że przy odpowiednim doborze parametrów nie tracimy praktycznie żadnej liczby B-gładkiej.

Drugą, zupełnie odwrotną możliwością, jest rezygnacja z przesiewania przez wyższe potęgi liczb pierwszych. Tak samo jak w poprzednim przypadku, odpowiedni dobór wartości granicznych "bliskości" pozwala mocno przyspieszyć proces, bez problemów związanych z traceniem potencjalnych kandydatów. Używając tego wariantu musimy w szczególny sposób potraktować potęgi liczby 2. Na szczęście nie jest to trudne: zakładając że $\tilde{x}$ jest nieparzyste, mamy

$$2 \mid Q(x) = \tilde{x}^2 - n \tag{3.26}$$

Rozważmy $y = n \mod 8$. Jeśli $y = 3$ lub $y = 7$, to $2 \mid Q(x)$. Jeśli $y = 5$, to $4 \mid Q(x)$. Ostatecznie, jeśli $y = 1$, to $8 \mid Q(x)$. Mamy teraz dwie możliwości: możemy lekko zmodyfikować n na potrzeby przesiewania aby zawsze uzyskać $8 \mid Q(x)$ gdy $2 \mid Q(x)$. W tym celu, ustalamy $n \leftarrow 3n$ jeśli $n \equiv 3 \pmod 8$, $n \leftarrow 5n$ jeśli $n \equiv 5 \pmod 8$ i $n \leftarrow 7n$ jeśli $n \equiv 7 \pmod 8$. Dokonawszy tej zmiany, przesiewamy całość odejmując zawsze $3 \log_2 2$. Drugą, prostszą możliwością, jest przesiewanie przez

2 tylko wtedy, gdy $n \equiv 1 \pmod 8$, a rezygnacja z przesiewania w innym wypadku. Rzeczywiście, wprowadzony wtedy błąd nie wyniesie więcej niż $2\log_2 2$, a więc jest bardzo mały.

3.3.3. Wybór pomocniczych algorytmów

W opisie metody sita kwadratowego kilkakrotnie odwołujemy się do algorytmów pomocniczych. Potrzebujemy wyznaczyć bazę czynników pierwszych, rozwiązać równanie kwadratowe $\pmod p$, czy też rozwiązać układ równań liniowych. Wybór metod do osiągnięcia niektórych z powyższych celów może mieć zauważalny wpływ na szybkość działania sita kwadratowego.

Najprostszym algorytmem na znajdowanie bazy czynników pierwszych jest sito Eratostenesa, którego złożoność obliczeniowa wynosi $O(N \log\log N)$ lub $O(N)$ z prostymi optymalizacjami, gdzie $N = B = L(n)^{\frac{\sqrt{2}}{4}}$ (patrz rozdział 3.2.3). Znane są szybsze metody takie jak sito Atkina, którego złożoność wynosi $O(\frac{N}{\log\log N})$, przy lepszym wykorzystaniu pamięci. Można jednak zaobserwować, że użycie lepszego algorytmu uzyskiwania bazy czynników ma znikomy wpływ na wydajność sita kwadratowego, jako że czas ten jest i tak zdominowany przez późniejsze operacje.

Przeciwnie, znaczny wpływ na szybkość ma dobór algorytmu rozwiązywania układu równań liniowych. Po pierwsze, macierz opisująca ten układ jest **binarna** (wartości macierzy to jedynie 0 i 1). Można zatem zamiast liczb całkowitych użyć pól bitowych. Wynikiem tego będzie znacząco zmniejszone (o stały czynnik) zapotrzebowanie na pamięć. Dodatkowo przy algorytmie (np. Gaussa) będziemy w stanie wykonywać operacje na grupie kolumn jednocześnie, co będzie miało także wpływ na szybkość działania.

Drugą własnością macierzy jest fakt, że jest ona **rzadka** (duża liczba jej komórek ma wartość 0). Ta informacja pozwala nam dobrać lepszy asymptotycznie algorytm rozwiązujący układ przez nią opisany. Dla macierzy rzadkich dużo lepiej niż metoda eliminacji Gaussa sprawdzają się metody Wiedemanna [29] albo Lanczosa [30]. Z tej pary wyróżnia się algorytm Wiedemanna, który trochę lepiej zachowuje się przy rozwiązywaniu macierzy binarnych. Użycie go daje wyraźny efekt przyspieszający działanie sita kwadratowego. Przedstawiona w równaniu (3.23) złożoność czasowa *QS* zakłada użycie algorytmu Gaussa. Wykorzystując metodę Wiedemanna możemy zmniejszyć ją do

$$O(e^{\sqrt{\ln n \ln\ln n}}) \tag{3.27}$$

a więc jeszcze bardziej redukujemy czynnik w wykładniku potęgi.

3.3.4. Duże czynniki pierwsze

Kolejną wariacją algorytmu sita kwadratowego jest użycie dużych czynników pierwszych (ang. *Large Prime Variation*). Zauważmy, że gdy po przesianiu naszego przedziału w danej komórce mamy wartość istotnie większą od 0, ale mniejszą od $2\log p_{max}$ (Gdzie p_{max} - największy czynnik pierwszy w naszej bazie), oznacza to, że znaleźliśmy liczbę, która jest *prawie* B-gładka, czyli nie posiada czynnika pierwszego większego od B za wyjątkiem jednego p, spełniającego nierówność $p_{max} < p < p_{max}^2$. Gdy w naszym przedziale istnieje tylko ta jedna liczba $Q(x_1)$ podzielna przez p, jest ona dla nas bezużyteczna. Sytuacja zmienia się, gdy znajdziemy drugą liczbę $Q(x_2)$ tego samego typu. Mnożąc $Q(x_1)$ i

$Q(x_2)$ przez siebie otrzymujemy liczbę, która rozkłada się całkowicie w bazie, za wyjątkiem czynnika p^2. Możemy zatem dodać ten iloczyn do naszego zbioru liczb B-gładkich (który właściwie przestaje być wtedy zbiorem liczb B-gładkich), a następnie powiększamy bazę czynników o liczbę pierwszą p. Zauważmy, że informację o liczbach tego typu dostajemy praktycznie gratis - wystarczy odpowiednio zmodyfikować wartość graniczną podczas przesiewania. Oczywiście, może się zdarzyć, że znajdziemy $k > 2$ liczby posiadających czynnik p - wtedy dodajemy je wszystkie do naszego zbioru jak powyżej. Należy też dodać, że *paradoks dnia urodzin* sugeruje, że przypadki gdy $k \geq 2$ będą częstsze niż się tego można spodziewać.

Wariację tą zaproponował w 1982 roku Carl Pomerance w [27]. Szacuje się, że użycie tej metody przyspiesza działanie algorytmu sita kwadratowego około 6 razy [31].

3.3.5. *MPQS*

Najczęściej stosowaną i zdecydowanie najwięcej dającą modyfikacją sita kwadratowego jest użycie algorytmu *Wielokrotnie Wielomianowego Sita Kwadratowego* (ang. *MPQS*, *Multiple Polynomial Quadratic Sieve*). Jak sama nazwa wskazuje metoda ta polega na użyciu więcej niż jednego wielomianu Q. Istnieją dwa warianty tego podejścia: pierwszy, który zastosował w implementacji algorytmu *QS* Davis [32] i drugi, który zaproponował Peter Montgomery w prywatnej korespondencji z autorem algorytmu *QS* - Carlem Pomerance [2]. Z tych dwóch wariacji lepsza jest metoda Montgomery'ego i aktualnie mówiąc o *MPQS* ma się ja z reguły na myśli. Zaproponował on użycie wielomianów Q w postaci:

$$Q(x) = ax^2 + 2bx + c \tag{3.28}$$

gdzie współczynniki a, b i c są dobierane w odpowiedni sposób [2, 33, 34].

Powodem, dla którego ta modyfikacja jest przydatna jest fakt, że im większy interwał przesiewania M obierzemy, tym większe będą wartości $Q(x)$ gdy x będzie dążył do końców tego przedziału, a jak już niejednokrotnie wspomniano, większe wartości $Q(x)$ oznaczają mniejsze prawdopodobieństwo że $Q(x)$ jest liczbą B-gładką. Użycie całej rodziny wielomianów pozwala nam znacząco zmniejszyć rozmiar przedziału. W metodzie *MPQS* ustalamy interwał przesiewania (dużo mniejszy niż w sicie kwadratowym bez tej modyfikacji), a następnie wykonujemy standardowy algorytm. Gdy po przesianiu okaże się, że ilość uzyskanych liczb B-gładkich jest zbyt mała, wyznaczamy nowy wielomian i ponownie przesiewamy ten sam przedział z jego użyciem. Powyższy schemat ponawiamy do momentu, w którym uzyskamy odpowiednią ilość liczb B-gładkich.

Pomerance zauważa w [35], że używanie rodziny wielomianów ma sens tylko wtedy, gdy zmienianie ich zajmuje nie więcej niż $25 - 30\%$ czasu działania całego algorytmu. Analiza czasowa metody Montgomery'ego jest trudna i dość mocno zależy od implementacji. Szacuje się jednak, że przy użyciu *MPQS* można zmniejszyć wielkość bazy czynników pierwszych (F) dziesięciokrotnie, a więc wielkość interwału M (będącą sześcianem F) tysiąckrotnie.

3.3.6. Zrównoleglenie sita kwadratowego

Algorytm sita kwadratowego jest dobrym kandydatem do implementacji na maszynach działających wielowątkowo. Zauważmy, że główny narzut obliczeniowy - wyszukiwanie odpowiednich liczb B-gładkich można prosto zrównoleglić dzieląc przedział przesiewania pomiędzy poszczególne jednostki obliczeniowe. Oczywiście, podział ten nie będzie idealny, jako że w dalszym ciągu musimy wyznaczać odpowiednie pierwiastki $\tilde{x}^2 \equiv n \pmod{p}$, co pociąga za sobą konieczność stosunkowo częstej komunikacji. Przykładowa implementacja tego typu znajduje się w [36]

Dużo lepszym pomysłem jest zrównoleglenie wariacji algorytmu: *MPQS* [34, 37]. Zamiast dzielić interwał, każda jednostka przesiewa swoją kopię tego interwału używając swojego własnego wielomianu. Takie działanie zdecydowanie redukuje komunikacyjny narzut, jako że poszczególne procesory wymieniają informacje tylko przy pobraniu wielomianu i po zakończeniu przesiewania, a cały reszta procesu działa niezależnie.

Niestety, nawet w metodzie *MPQS* pozostaje problem związany z rozwiązaniem układu równań liniowych. Nie jest aktualnie znany algorytm, który mógłby dokonać tego zadania współbieżnie, zatem ta część pozostaje wąskim gardłem równoległej implementacji.

4. Implementacja i testy

4.1. Implementacja

Celem niniejszej pracy jest implementacja wybranego algorytmu faktoryzacji na telefon komórkowy. Dokonanie tego pozwoli ocenić w zupełnie nowy sposób wydajność dzisiejszych urządzeń mobilnych, włącznie z porównaniem ich możliwości z maszynami klasy PC.

Jakkolwiek dzisiejsze smartfony charakteryzują się coraz lepszymi parametrami, w dalszym ciągu nie dogoniły możliwości komputerów osobistych. Jest zatem jasne, że faktoryzacja na tego typu urządzeniu nie umożliwi bicia globalnych rekordów i górna granica dla wielkości liczb które można rozłożyć na czynniki będzie mniejsza niż dzisiejsze 232 cyfry. Z tego powodu wybranym do implementacji algorytmem jest Sito Kwadratowe, które jest najszybszą znaną metodą faktoryzującą liczby o ilości cyfr mniejszej niż 150.

Jako platformę mobilną wybrano telefony z systemem operacyjnym Android. Powodów takiej, a nie innej decyzji było kilka. Po pierwsze system ten umożliwia bezproblemowe tworzenie i uruchamianie aplikacji bez konieczności dokonywania opłat na rzecz twórcy platformy, co odróżnia go na przykład od produktów firm Apple i Microsoft. Dodatkowo Android bazuje na Linuksie, co ułatwia wykorzystanie dodatkowych bibliotek, poczynając od tych systemowych, a kończąc na zewnętrznych jak, na przykład, biblioteka do obsługi liczb całkowitych dowolnej precyzji. Trzecim powodem jest istnienie przejrzystego API, zarówno na wysokim poziomie, jak i na tym niskim. Szczególnie ta druga część jest istotna, gdyż na niższym poziomie mamy możliwość większej kontroli kodu, a co za tym idzie jego dokładniejszej optymalizacji co jest szczególnie ważne gdy tworzymy program, który musi dokonywać dużej ilości obliczeń matematycznych w jak najkrótszym czasie.

4.1.1. Szczegóły implementacji algorytmu

Do implementacji wybrano standardowy algorytm Sita Kwadratowego, bez modyfikacji typu MPQS (rozdział 3.3.5), czy wariacji z dużymi czynnikami pierwszymi (rozdział 3.3.4). Optymalizacje te poprawiają osiągi głównie dla górnej granicy liczb, na których działa metoda, nie przydałyby się zatem w naszym przypadku. Użyto standardowych wartości parametrów $F = \pi(B)$ (wielkość bazy czynników pierwszych) i M (rozdział 3.2.3).

$$\begin{aligned} \pi(B) = F &= e^{\sqrt{\frac{1}{8}\ln n \ln\ln n}} \\ M = F^3 &= e^{\sqrt{\frac{9}{8}\ln n \ln\ln n}} \end{aligned} \tag{4.1}$$

Wybór wartości B opisany jest w rozdziale 4.1.2.

Użyto wariacji z przybliżonymi logarytmami (rozdział 3.3.1), a także zrezygnowano z przesiewania przez wyższe potęgi czynników pierwszych (rozdział 3.3.2). W związku z tym ostatnim krokiem, krytyczne staje się wybranie górnej wartości logarytmu którą uznajemy za dostatecznie bliską zera. Zgodnie z sugestią z [33] granica ta jest ustalona na

$$G = \frac{1}{2}\ln n + \ln M - T \ln p_{max} \tag{4.2}$$

gdzie M jest rozmiarem naszego przedziału przesiewania, p_{max} największą liczbą pierwszą obecną w bazie czynników, a T pewną wartością bliską 2, zależną od wielkości faktoryzowanej liczby. Silverman w [34] sugeruje $T = 1.5$ dla liczb 30-cyfrowych, $T = 2$ dla liczb 45-cyfrowych i $T = 2.6$ dla liczb 66 cyfrowych. Na potrzeby programu, uogólniono te wartości używając następującego wzoru:

$$T(x) = \frac{1}{2} + \frac{x}{30} \tag{4.3}$$

gdzie x jest ilością cyfr n. Bez względu na to jak dokładnie będziemy się starali dobrać parametr G i tak pewna ilość liczb nie spełniających warunku B-gładkości prześlizgnie się przez nasze sito. Dlatego po wstępnym ich odsianiu dokonuje się ponownego sprawdzenia czy otrzymane wartości na pewno spełniają pożądane własności. Test ten wykonywany jest przy pomocy naiwnej metody faktoryzacji.

Ważnym elementem algorytmu jest sprawdzenie czy dane wejściowe spełniają jego założenia. Na początku działania testowane jest czy faktoryzowana liczba jest nieparzysta, czy jest złożona (za pomocą testu pierwszości), a także, czy nie jest potęgą liczby całkowitej. W przypadku negatywnej odpowiedzi na dowolny z powyższych warunków metoda Sita Kwadratowego nie działałaby prawidłowo. Dodatkowo, testowane jest czy liczba jest wystarczająco duża (przyjęto dolną granice równą 10^5), a także, czy nie posiada czynników pierwszych nie większych niż jej logarytm. Jakkolwiek w tych przypadkach algorytm zadziałałby, jego wykonanie byłoby wysoce nieoptymalne i prostsze metody (jak naiwna) sprawią się dużo lepiej.

Największym z problemów przy implementacji algorytmu na urządzenia mobilne to ilość dostępnej pamięci operacyjnej. Rzeczywiście, słabszy procesor powoduje, że obliczenia zakończą się później, podczas gdy za mała ilość pamięci nie pozwala się im odbyć w ogóle. Problem ten nie jest czysto teoretyczny - dla liczby 40-cyfrowej mamy $M = 2503937714$, a więc potrzebujemy sita o rozmiarze $2M + 1 = 5007875429$, co nawet przy założeniu, że pojedynczym elementem sita jest liczba jednobajtowa daje nam tablice, która zajmuje prawie 5 GB RAM. Aktualnie nie istnieją urządzenia wyposażone w tak dużą jego ilość. Możliwym rozwiązaniem jest wykorzystanie pamięci zewnętrznej telefonu (większość dzisiejszych smartfonów jest wyposażona w karty o pojemności kilku gigabajtów), jednak jak wiadomo dostęp do niej jest bardzo wolny. Zdecydowano się więc na inną modyfikację: program pobiera ilość dostępnych na urządzeniu zasobów, po czym zakłada, że może bezpiecznie użyć mniej więcej

połowę z nich (reszta jest wymagana do działania samego systemu operacyjnego). Jeśli parametr M wymaga alokacji większej ilości pamięci niż dopuszczalna, jest on sztucznie zaniżany do tej granicy. Zmiana ta może mieć istotny narzut na czas obliczeń, gdyż przy faktoryzacji większych liczb musimy dokonywać kilku przesiań zamiast jednego. W przeciwnym jednak wypadku dla takich danych wejściowych wykonanie programu nie byłoby w ogóle możliwe.

Dodatkową optymalizacją wykorzystania pamięci jest użycie pól bitowych dla przechowywania wektorów wykładników $\pmod 2$, a co za tym idzie skonstruowanej z nich macierzy. Ta modyfikacja nie ma negatywnego wpływu na szybkość działania algorytmu.

4.1.2. Wybór algorytmów i bibliotek pomocniczych

Metoda Sita Kwadratowego korzysta z wielu dodatkowych algorytmów. Ich wybór i konfiguracja ma istotny wpływ na szybkość działania finalnej aplikacji.

Liczby całkowite o dowolnej precyzji

Dopóki wszystkie działania w metodzie Sita Kwadratowego mają wyniki ograniczone około 20 cyframi (64 bity) możemy bez przeszkód korzystać ze standardowo dostępnych typów całkowitych (na przykład 64-bitowy typ `long long int` dostępny w C). Oczywiście, 20 cyfr nie jest żadnym wyzwaniem dla dzisiejszych algorytmów. Jest zatem jasne, że do poważniejszych testów będziemy musieli użyć liczb całkowitych o dowolnej precyzji (ang. *arbitrary-precision integers*, *multiple precision integers*). Jak sama nazwa wskazuje, liczby tego typu nie mają górnych ograniczeń na swoją wartość (w przeciwieństwie do standardowych typów wbudowanych), za wyjątkiem fizycznych właściwości maszyny, takich jak dostępna pamięć operacyjna. Istnieje kilka bibliotek oferujących tego typu funkcjonalność. Na potrzeby niniejszej pracy zdecydowano się na użycie jednej z najbardziej znanych z nich, **GMP** (ang. *GNU Multiple Precision Arithmetic Library*) [38]. Spełnia ona wszystkie wymagania stawiane produktom tego rodzaju, a dodatkowo udostępnia swój kod źródłowy do użytku publicznego.

Dostępność kodu źródłowego jest niezwykle istotna w związku z faktem, że docelową platformą na której biblioteka ta będzie używana jest telefon z systemem Android. Nie istnieją do tej pory gotowe (skompilowane) wersje przeznaczone na takie urządzenia, dlatego w ramach pracy trzeba było zbudować ją ręcznie. Użyto w tym celu jednego z narzędzi dostarczanych wraz z NDK, *Standalone Toolchain*. Program ten - będący jeszcze w wersji eksperymentalnej - umożliwia kompilacje na komputerze stacjonarnym zewnętrznych bibliotek do ABI (ang. *Application Binary Interface*) używanego przez procesory obecne w urządzeniach z systemem operacyjnym Android.

Tworzenie bazy czynników pierwszych

W celu tworzenia bazy czynników pierwszych stosuje się standardowy algorytm Sita Eratostenesa. Jak wspominano w rozdziale 3.3.3, ewentualny zysk w postaci użycia wydajniejszej metody byłby praktycznie niezauważalny. Wymagane były jednak dodatkowe modyfikacje aby uzyskać odpowiednią jej rozmiar. Po pierwsze, w ramach początkowych parametrów ustala się nie górną granice poszukiwań, ale ilość konkretnych liczb pierwszych. Dodatkowo, nie wszystkie liczby które znajdziemy mogą trafić do naszej bazy czynników. Rzeczywiście, mając do czynienia z liczbą pierwszą p rozważmy jej symbol

Legendre'a $\left(\frac{n}{p}\right)$ (gdzie n jest faktoryzowaną liczbą). Jeśli wynosi on -1, a więc nie istnieje takie b, że $b^2 \equiv n \pmod{p}$, to rozwiązanie równania w punkcie 4.1 algorytmu (rozdział 3.2.2) nie da nam żadnych wyników, a więc przesiewanie przedziału przez dany czynnik się nie odbędzie. Można zatem z góry odrzucać takie wartości.

Ogólny schemat budowania bazy czynników przedstawia się zatem następująco:

1. Ustal wartość $\tilde{B}$ - wielkość przedziału, na którym dokonamy wstępnego przesiewania
2. Na przedziale $\left[2, \tilde{B}\right]$ wykonaj algorytm Sita Eratostenesa, otrzymując zbiór liczb pierwszych
3. Ze zbioru czynników otrzymanych w punkcie 2 usuń te, dla których symbol Legendre'a $\left(\frac{n}{p}\right)$ jest różny od 1
4. Jeśli ilość liczb pierwszych otrzymanych po krokach 1 - 3 jest mniejsza niż wymagana (B), za pomocą dowolnej metody znajdź brakujące czynniki.

Granice $\tilde{B}$ w punkcie 1 przyjmuje się zgodnie z sugestią z [39] taką samą jak dla algorytmu Dixona (rozdział 1.2.2)

$$\tilde{B} = e^{\frac{1}{2}\sqrt{\ln(n)\ln\ln n}} \tag{4.4}$$

W punkcie 4 nie opłaca się po raz kolejny wykonywać algorytmu sita, gdyż ilość brakujących liczb pierwszych jest z reguły bardzo mała. Zamiast tego dla kolejnych liczb (większych niż $\tilde{B}$) wykonujemy test pierwszości, a także (jeśli się on powiedzie) sprawdzamy ich symbol Legendre'a. W przypadku obu pozytywnych wyników dodajemy taką liczbę do naszej bazy.

Szukanie reszt kwadratowych

Przy szukaniu reszt kwadratowych dla konkretnych liczb pierwszych (punkt 4.1 algorytmu, rozdział 3.2.2) użyto standardowej metody Shanksa-Tonellego. Jej oczekiwana złożoność czasowa wynosi $O\left(\log^2 p\right)$ co jest wartością całkowicie wystarczającą na potrzeby aplikacji.

Rozwiązywanie układu równań liniowych

Do rozwiązywania końcowego układu równań liniowych (punkt 8 algorytmu, rozdział 3.2.2) wykorzystano standardowy algorytm Eliminacji Gaussa (GEM). Co prawda jak wspomniano w rozdziale 3.3.3 użycie szybszej metody mogłoby zmienić złożoność czasową Sita Kwadratowego, jednak w praktyce na testowanych przykładach okazało się, że przesiewanie jest operacją dominującą czas działania, szczególnie w związku z optymalizacją wykorzystania pamięci.

Drobną modyfikacją metody GEM jest sposób tworzenia rozwiązań. Jak wiadomo, dowolne z nich prowadzi do nietrywialnej faktoryzacji liczby z prawdopodobieństwem większym bądź równym $\frac{1}{2}$. W związku z tym jest jasne, że w większości przypadków niepotrzebne jest marnowanie czasu i pamięci na tworzenie i przechowywanie każdego rozwiązania. Zamiast tego, po sprowadzeniu macierzy do postaci schodkowej zlicza się tylko ich ilość, a konkretne instancje buduje się na bieżąco.

4.1.3. Struktura programu

Kod aplikacji dzieli się na dwie główne składowe - napisaną w Javie częścią wysokopoziomową, która odpowiada za interakcje z użytkownikiem oraz napisaną w C z użyciem NDK części niskopoziomowej, odpowiedzialną za samo wykonanie faktoryzacji. Poniższe podrozdziały opisują szczegóły implementacji programu przedstawiając jej fragmenty w postaci pseudokodu.

Wysoki poziom (Java)

Wysokopoziomowa część aplikacji odpowiada właściwie tylko za obsługę GUI. Większość składowych programu (*Activities*) używa odwołania do natywnej metody, zdefiniowanego w klasie `FactorizatorWrapper` (listing 4.1).

Listing 4.1: Wrapper na wywołania metody natywnej

```
public class FactorizatorWrapper {
  public native String[]
    factorize(String number, FactorizationMethod method);
}
```

Metoda natywna przyjmuje dwa parametry. Pierwszym z nich jest liczba którą chcemy sfaktoryzować (podana jako `String`), drugim algorytm faktoryzacji który mamy użyć (listing 4.2).

Listing 4.2: Dostępne algorytmy faktoryzacji

```
public enum FactorizationMethod {
  NAIVE, FERMAT, DIXON, QS;
}
```

Przykładowe użycie wrappera znajduje się w klasie `BenchmarkActivity`, która umożliwia uruchomienie benchmarku telefonu (listing 4.3).

Listing 4.3: Wykorzystanie wrappera

```
public class BenchmarkActivity extends Activity {
  private String[] benchmarkNumbers = { /* ... */ };

  protected Date runBenchmark() {
    Date start = new Date();

    FactorizatorWrapper wrapper = new FactorizatorWrapper();

    for (String number : benchmarkNumbers) {
      wrapper.factorize(number, FactorizationMethod.QS);
    }

    return new Date(new Date().getTime() - start.getTime());
  }
}
```

Niski poziom (C)

Implementacja algorytmu (algorytmów) znajduje się w części natywnej aplikacji. Pierwszym krokiem, który musi zostać zrobiony, to translacja typów danych JNI (Java) na te powszechnie używane w C (listing 4.4).

Listing 4.4: Konwersja typów danych

```
jobjectArray Java_FactorizatorWrapper_factorize(
  JNIEnv * env, jobject thiz, jstring number, jobject method) {
    const char * c_number = (*env)->GetStringUTFChars(env, number, NULL);

    jclass enum_class = (*env)->GetObjectClass(env, method);
    jmethodID enum_ordinal_method = (*env)->GetMethodID(env, enum_class, "ordinal", "()I");
    jint jint_method = (*env)->CallIntMethod(env, method, enum_ordinal_method);

    factors_num = factorize(c_number, &c_result, (int) jint_method, &time);

    jobjectArray result = (jobjectArray)(*env)->NewObjectArray(
      env, factors_num, (*env)->FindClass(env, "java/lang/String"),
      (*env)->NewStringUTF(env, ""));

    for (long i = 0; i < factors_num; i++)
      (*env)->SetObjectArrayElement(env, result, i, (*env)->NewStringUTF(env, c_result[i]));

    return result;
}
```

Po konwersji danych, działanie przenosi się do fasady biblioteki, która dokonuje faktoryzacji. Zawiera ona dwie metody: `do_factorize`, która dokonuje pełnej faktoryzacji do postaci kanonicznej liczby i `factorize`, która ją opakowuje. Obie metody zostały przedstawione na listingu 4.5

Listing 4.5: Publiczny interfejs faktoryzacji

```
long do_factorize(mpz_t number, mpz_t ** factors, factorization_method method) {
  mpz_t factor1 = 1, factor2 = 1;

  method(number, factor1, factor2);

  factors_cnt = 0;

  if (factor1 > 1 && factor2 > 1) {
    long factor1_factors_cnt = do_factorize(factor1, &factor1_factors, method);
    long factor2_factors_cnt = do_factorize(factor2, &factor2_factors, method);

    factors_cnt += (factor1_factors_cnt > 1 ? factor1_factors_cnt : ++factor1_factors_cnt);
    factors_cnt += (factor2_factors_cnt > 1 ? factor2_factors_cnt : ++factor2_factors_cnt);

    (*factors) = (mpz_t *) malloc(sizeof(mpz_t) * factors_cnt);

    /* Skopiuj czynniki z factor1_factors i factor2_factors do factors */
  }

  return factors_cnt;
}
```

```
long factorize(const char * number, char *** result, int method, long int * time) {
  mpz_t mpz_number = mpz_init(number);

  long factors_cnt = 0;

  factors_cnt = do_factorize(mpz_number, &factors, getMethod(method));

  if (factors_cnt > 0) {
    /* Skonwertuj otrzymane czynniki do stringów i skopiuj do results */
  }

  return factors_cnt;
}
```

Metoda `do_factorize` szuka dwóch czynników liczby. Jeśli je znajdzie, wywołuje rekurencyjnie samą siebie w celu ich faktoryzacji. Jednym z jej parametrów jest `method` - wskaźnik na funkcję, która implementuje konkretny algorytm.

Ponieważ dwie z czterech metod faktoryzacji dostępnych w programie należą do rodziny Kraitchika, starano się wydzielić część wspólną dla wszystkich algorytmów tego typu. Dzięki temu przy implementacji konkretnych wariacji można skupić się jedynie na specyficznych dla danej odmiany fragmentach, gdyż szkielet jest już gotowy. Wspólne funkcje przedstawione są na listingu 4.6.

Listing 4.6: Część wspólna algorytmów z rodziny Kraitchika

```
fixed_int get_standard_smooth_boundary(mpz_t number) {
  double l = mpz_sizeinbase(number, 2)/1.4426950408889634;

  return (fixed_int) exp(0.5*sqrt(l*log(l)));
}

void get_factors(mpz_t number, mpz_t factor1, mpz_t factor2,
  mpz_t * z_numbers, mpz_t * z2_mod_n_numbers, bit_field_size cols, bit_field solution) {
    mpz_t a = 1, b = 1;

    for (bit_field_size j = 0; j < cols; j++) {
      if (get_bit(solution, j)) {
        a *= z_numbers[j];
        b *= z2_mod_n_numbers[j];
      }
    }

    b = sqrt(b)

    if (!mpz_congruent(a, b, number) && !mpz_congruent(a, -b, number)) {
      factor1 = gcd(abs(a - b), number)
      factor2 = gcd(a + b, number)
    }
}

fixed_int check_preconditions(mpz_t number, mpz_t factor1, mpz_t factor2) {
  // Sprawdź czy number jest liczbą pierwszą używając testu pierwszości
  if (mpz_probab_prime(number)) {
    return 0;
  }
```

```
  // Sprawdź, czy number jest potęgą liczby całkowitej
  if (mpz_perfect_power_p(number)) {
    mpz_t root = /* Znajdź pierwiastek */;

    factor1 = root;
    factor2 = number / root;

    return 0;
  }

  // Jeśli number jest mniejszy niż 10^5 nie ma sensu używać algorytmu QS
  if (number < 10^5) {
    naive_method(number, factor1, factor2);
    return 0;
  }

  // Sprawdź, czy number ma czynniki pierwsze mniejsze niż swój logarytm
  bounded_naive_method_ui(number, factor1, factor2, log(number));

  if (factor1 > 1 && factor2 > 1) {
    return 0;
  }

  // number spełnia założenia wstępne algorytmów z rodziny Kraitchika
  return 1;
}

void general_congruence_of_squares_method(mpz_t number, mpz_t factor1, mpz_t factor2,
  factor_base_function get_factor_base, b_smooth_numbers_function get_b_smooth_numbers) {
  // 1. Sprawdź założenia wstępne
  if (!check_preconditions(number, factor1, factor2))
    return;

  // 2. Stwórz bazę czynników pierwszych
  factor_cnt = get_factor_base(number, &factor_base);

  pairs_cnt = 0, old_pairs_cnt = 0;

  // 3. Powtarzaj pętle, dopóki nie uzyskamy nietrywialnych czynników
  while (factor1 == 1 && factor2 == 1) {
    // 3.1 Get next B-smooth numbers portion
    pairs_cnt = get_b_smooth_numbers(
      number, factor_base, factor_cnt, &z_numbers, &z2_mod_n_numbers, pairs_cnt);

    if (pairs_cnt < factor_cnt + 1 || old_pairs_cnt == pairs_cnt) {
      // 3.2 Powtórz pętle, jeśli mamy za mało par
      continue;
    }

    // 3.3 Wypełnij macierz wykładników
    fill_exponents_matrix(
      exponents_matrix, factor_base, factor_cnt, pairs_cnt, z2_mod_n_numbers);

    // 3.4 Rozwiąż macierz
    current_matrix_solver(exponents_matrix, factor_cnt, pairs_cnt, solutions_cnt);

    // 3.5 Sprawdź, czy uzyskaliśmy nietrywialną faktoryzacje
```

```
    for(/* Wszystkie rozwiązania macierzy */)
      get_next_solution(exponents_matrix, solution);
      get_factors(number, factor1, factor2, z_numbers, z2_mod_n_numbers, pairs_cnt, solution);

      if (factor1 > 1 && factor2 > 1)
        break;
    }
}
```

Najważniejszą funkcją jest `general_congruence_of_squares_method`. Implementuje ona ogólny schemat działania metod z rodziny Kraitchika. jako parametry pobiera ona `factor_base_function` i `b_smooth_numbers_function`. Dzięki przekazaniu w tym miejscu odpowiednich ich implementacji (pierwsza, która tworzy bazę czynników, a druga, która jest w stanie produkować pary spełniające równanie 1.11) możliwe jest stworzenie konkretnej instancji algorytmu z rodziny.

Funkcja `get_standard_smooth_boundary` wylicza standardową wartość B taką jakiej używa się na przykład w algorytmie Dixona (ale też pośrednio w Sicie Kwadratowym). Jako że biblioteka GMP nie posiada funkcji wyliczającej logarytm liczby, przy wyliczeniu korzystamy z tożsamości $\ln x = \frac{\log_2 x}{log_2 e}$. Wartość $\log_2 n$ jest wyliczana w sposób przybliżony, jako liczba cyfr binarnych n. Funkcja `get_factors` dostaje jako argumenty parę spełniającą równanie 1.11 i jeśli jest to możliwe zwraca wyliczone z niej dwa czynniki. Funkcja `check_preconditions` sprawdza warunki wstępne działania algorytmów (rozdział 4.1.1).

Jednym z konkretnych zaimplementowanych algorytmów z rodziny jest oczywiście Sito Kwadratowe. Fragmenty przedstawiające jego działanie znajdują się na listingu 4.7.

Listing 4.7: Metody specyficzne dla algorytmu Sita Kwadratowego

```
fixed_int get_qs_factor_base_size(mpz_t number) {
  double l = mpz_sizeinbase(number, 2)/1.4426950408889634;
  return (fixed_int) exp(sqrt(0.125*l*log(l)));
}

fixed_int get_qs_sieving_interval_size(mpz_t number) {
  double l = mpz_sizeinbase(number, 2)/1.4426950408889634;
  return (fixed_int) (exp(sqrt(1.125*l*log(l))) + 0.5);
}

fixed_int get_qs_factor_base(mpz_t number, fixed_int ** factor_base) {
  fixed_int tmp_factor_cnt = current_prime_factors_function(
    &tmp_factor_base, get_standard_smooth_boundary(number));

  factor_base_cnt = get_qs_factor_base_size(number);
  (* factor_base)[0] = -1;
  factor_cnt = 1;

  for (fixed_int i = 0; factor_cnt < factor_base_cnt && i < tmp_factor_cnt; i++)
    if (mpz_legendre(number, tmp_factor_base[i]) == 1)
      (* factor_base)[factor_cnt++] = tmp_factor_base[i];

  current = (* factor_base)[factor_cnt - 1];
```

```
  while (factor_cnt < factor_base_cnt) {
    current++;

    if (mpz_probab_prime(current) && mpz_legendre(number, current) == 1)
      (* factor_base)[factor_cnt++] = mpz_get_fi(current);
  }

  return factor_cnt;
}

fixed_int quadratic_sieve(mpz_t number, fixed_int interval_size, mpz_t interval_offset,
  fixed_int * factor_base, fixed_int factors_cnt, fixed_int pairs_cnt, mpz_t ** z_numbers,
  mpz_t ** z2_mod_n_numbers) {
    for (fixed_int i = 0; i < interval_size; i++)
      sieve[i] = mpz_sizeinbase(Q(sieve[i]), 2);

    // factor_base[0] to -1, pomiń

    // factor_base[1] to 2, szczególny przypadek
    if (mpz_congruent(number, 1, 8)) {
      q = Q(sieve[0]) % 2 == 0 ? Q(sieve[1]) : Q(sieve[0]);

      /* Przesiej co drugą liczbę zaczynając od q */
    }

    // Pozostałe czynniki
    for (fixed_int i = 2; i < factors_cnt; i++) {
      shanks_tonelli_algorithm(number, factor_base[i], solution);
      do_sieve(solution, factor_base[i], sieve, interval_offset, interval_size, fsqrt_n);
    }

    double T = 0.5 + ((double) mpz_sizeinbase(number, 10)) / 30.0;
    double threshold = (0.5 * mpz_sizeinbase(number, 2))/1.4426950408889634;
    threshold += (log(interval_size / 2) - T*log(factor_base[factors_cnt - 1]));

    for (fixed_int i = 0; i < interval_size; i++)
      if (sieve[i] < threshold) {
        if(trial_check(i, interval_offset, number, fsqrt_n, factor_base, factors_cnt))
          new_pairs_cnt++;
        else
          sieve[i] = threshold + 1;

    for (fixed_int i = 0, zi = pairs_cnt; i < interval_size; i++) {
      (* z_numbers)[zi] = i*interval_offset + sqrt(number)
      (* z2_mod_n_numbers)[zi++] = Q(sieve[i])
    }

    return pairs_cnt + new_pairs_cnt;
}

fixed_int get_next_b_smooth_numbers_qs(mpz_t number, fixed_int * factor_base,
  fixed_int factor_cnt, mpz_t ** z_numbers, mpz_t ** z2_mod_n_numbers, fixed_int pairs_cnt) {
    pairs_cnt = quadratic_sieve(number, sieving_interval_size, sieving_interval_offset,
      factor_base, factor_cnt, pairs_cnt, z_numbers, z2_mod_n_numbers);

    sieving_interval_offset *= -1
```

```
    if (sieving_interval_offset < 0)
      sieving_interval_offset -= sieving_interval_size

    return pairs_cnt;
}

void qs_method(mpz_t number, mpz_t factor1, mpz_t factor2) {
  sieving_interval_size = get_qs_sieving_interval_size(number);

  max_ram = get_max_ram();

  if ((2*sieving_interval_size + 1) * sizeof(fixed_int) > max_ram)
    sieving_interval_size = ((max_ram / sizeof(fixed_int)) >> 1) - 1;

  sieving_interval_offset = -sieving_interval_size;

  sieving_interval_size = 2*sieving_interval_size + 1;

  general_congruence_of_squares_method(number, factor1, factor2,
    get_qs_factor_base, get_next_b_smooth_numbers_qs);
}
```

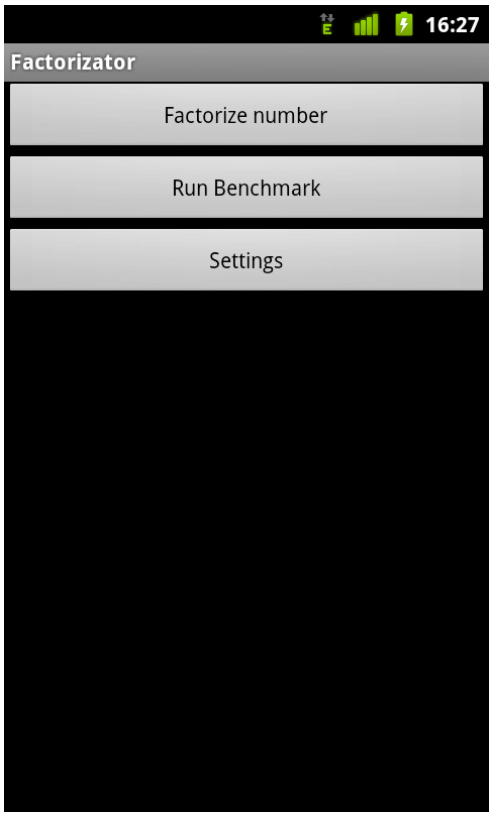
Implementatorem funkcji tworzącej bazę czynników pierwszych jest `get_qs_factor_base`. Korzysta ona z pomocy metody tworzącej tą bazę w ogólnym przypadku (Sito Eratostenesa). Funkcja `get_next_b_smooth_numbers_qs` natomiast implementuje wyszukiwanie kolejnych par liczbowych. Podczas pojedynczego wywołania przesiewa ona cały przedział (którego granice się zmieniają) i zwraca wszystkie znalezione pary. Samo przesiewanie odbywa się wewnątrz metody `quadratic_sieve`. Metoda `qs_method` jest metodą, która łączy wszystko w całość, dokonując pełnej faktoryzacji. Wykonuje ona na początku dodatkową prace (inicjalizacja przedziału przesiewania, ustalenie maksymalnej ilości pamięci operacyjnej), a następnie przekazuje sterowanie do opisywanej wcześniej metody `general_congruence_of_squares_method` z odpowiednimi parametrami.

4.2. Aplikacja

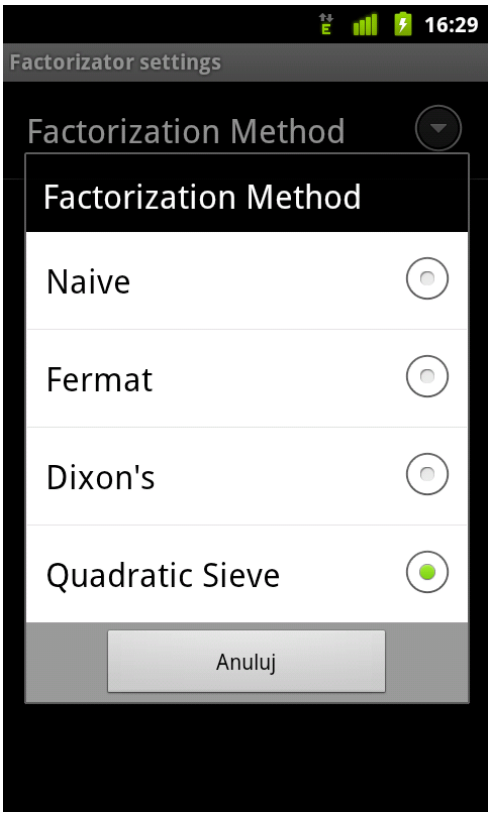
Po uruchomieniu aplikacji ukazuje się jej główny ekran (rysunek 4.1). Jego podstawową składową jest menu, z którego mamy możliwość wybrać interesującą nas funkcjonalność. Możemy sfaktoryzować dowolną liczbę (przycisk “Factorize number”), uruchomić test wydajności telefonu (przycisk “Run Benchmark”) i zmienić ustawienia programu (przycisk “Settings”).

Ekran ustawień aplikacji zaprezentowany jest na rysunku 4.2. Jedynym parametrem działania programu, który możemy modyfikować jest algorytm użyty do faktoryzacji dowolnej liczby (benchmark zawsze używa metody Sita Kwadratowego). Po wybraniu “Factorization Method” pokazuje się nam lista dostępnych algorytmów. Do wyboru - poza Sitem Kwadratowym - mamy metodę naiwną, Fermata i Dixona.

Rysunek 4.3 przedstawia funkcjonalność rozkładu na czynniki. Po wybraniu przycisku “Number” aplikacja przenosi nas do formularza, gdzie możemy wybrać liczbę którą chcemy sfaktoryzować (rysunek 4.4). Istnieją dwie możliwości: ręczne wpisanie wartości albo wybór pliku, w którym jest ona



Rysunek 4.1: Główny ekran aplikacji



Rysunek 4.2: Ustawienia aplikacji

zapisana. W drugim przypadku możliwe jest skorzystanie z zewnętrznego oprogramowania, które wyświetla nam strukturę katalogów gdzie możemy wybrać nasz plik, bez konieczności wpisywania jego pełnej ścieżki w odpowiednie pole tekstowe. W obu przypadkach możemy podać kilka liczb do faktoryzacji - wystarczy oddzielić je przecinkami.

Gdy już wybierzemy liczbę, wciśnięcie przycisku “Factorize” uruchomi faktoryzacje. Oczywiście, może ona potrwać dłuższą chwilę. Po wyliczeniu wynik jest prezentowany na ekranie.

Aby pomóc w śledzeniu postępów aplikacja loguje swoje działanie za pomocą standardowo dostępnych w Androidzie mechanizmów. Informacje te można czytać korzystając z zewnętrznych narzędzi do debugowania dostarczanych wraz z SDK (`adb`, `ddms`) lub z dodatkowych aplikacji na telefonie, takich jak **aLogcat**. Przykładowe logowane dane przedstawione są na rysunku 4.5.

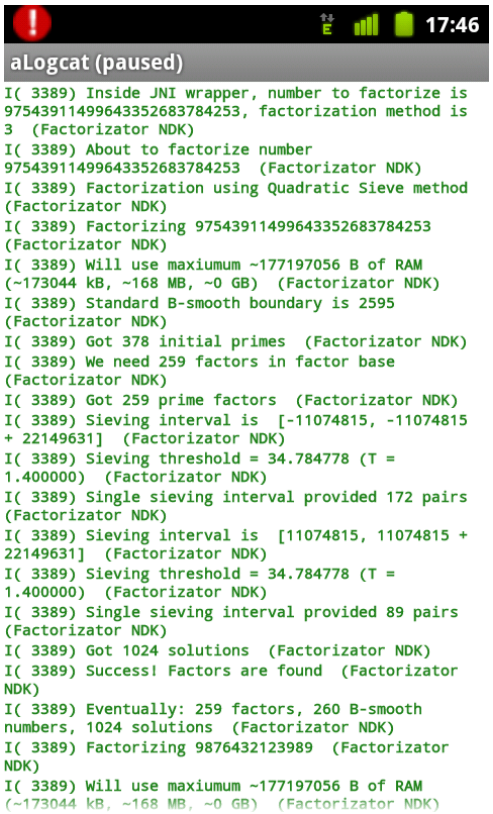
Ostatnią funkcjonalnością aplikacji jest przeprowadzenie benchmarku. Możemy go dokonać po wybraniu z głównego menu opcji “Run Benchmark”. Następnie, po potwierdzeniu chęci uruchomienia telefon zaczyna wykonywać pracę, po której skończeniu prezentuje wyniki na ekranie jak na rysunku 4.6. Dla potrzeb pracy, wyniki benchmarku można wysłać po wciśnięciu przycisku “Send results” - aplikacja skonstruuje wtedy wiadomość e-mail zawierającą wynik testu, a także parametry telefonu.

4.3. Testy

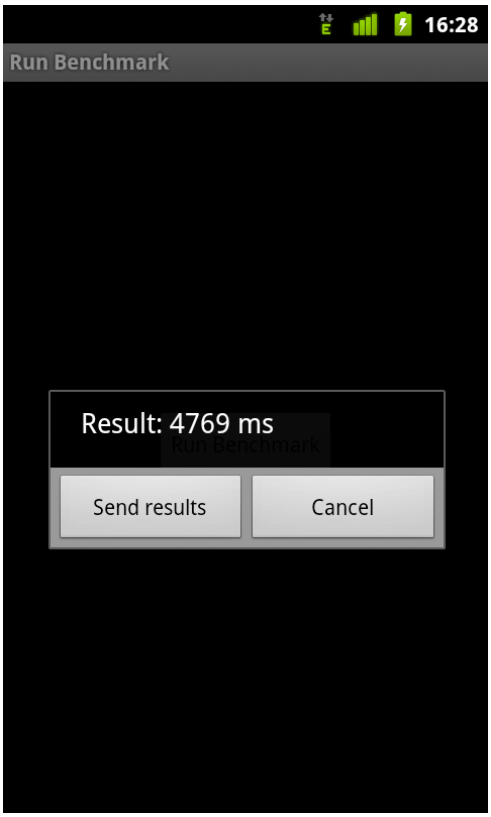
Uruchomienie i przetestowanie aplikacji podzielono na dwie fazy. Po pierwsze dokonano faktoryzacji dużej liczby półpierwszej, aby przetestować maksymalne możliwości programu. Druga część polegała

Rysunek 4.3: Faktoryzacja liczby

Rysunek 4.4: Wprowadzanie liczby



Rysunek 4.5: Logi aplikacji



Rysunek 4.6: Benchmark telefonu

na uruchomieniu benchmarku porównującego na wybranych urządzeniach mobilnych.

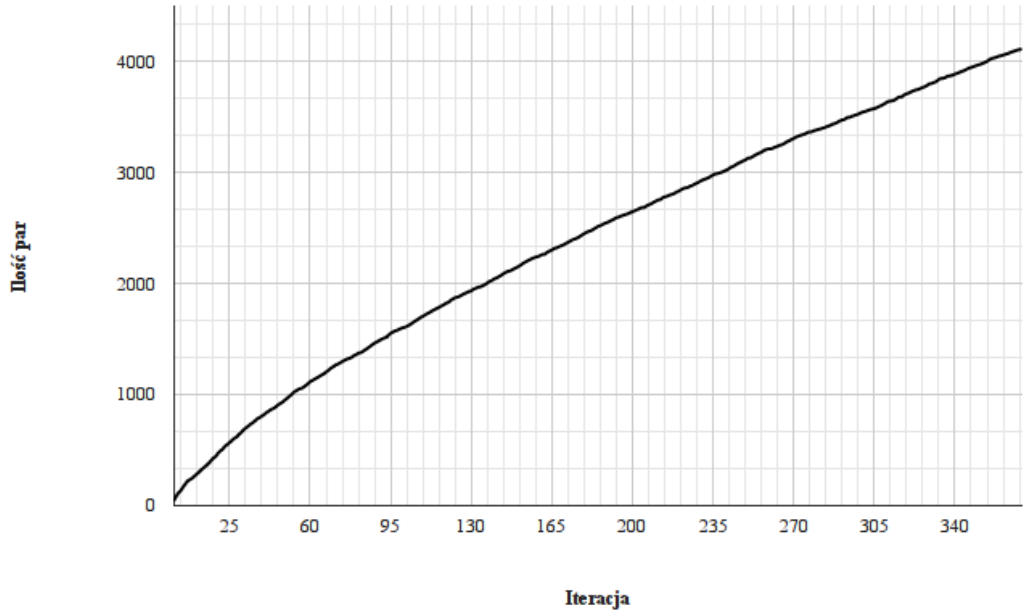
4.3.1. Limity

W celu zbadania maksymalnych możliwości algorytmu, użyto telefonu GT-I9000 (dokładna jego specyfikacja znajduje się poniżej w tabeli 4.1). Aby jak najbardziej wyeliminować wpływ zewnętrznych procesów systemowych na czas działania algorytmu, telefon znajdował się w tak zwanym trybie samolotowym, czyli miał wyłączone wszystkie połączenia bezprzewodowe. Krok ten jest wystarczający aby uzyskać obiektywne wyniki, gdyż zdecydowana większość serwisów działających w tle odpowiada za synchronizacje danych przez sieć. Wyłączenie anten powoduje automatyczne wyłączenie tych usług.

Po oszacowaniu czasu faktoryzacji zdecydowano się na rozłożenie na czynniki pierwsze 50-cyfrowej liczby T_{50} (4.5).

$$\begin{aligned} T_{50} =& 20813251728932894244634802862215740589474983559260 7 = \\ & 45621542859632546895229 39 \cdot 45621542859632546895229613 \end{aligned} \tag{4.5}$$

Po 15 minutach działania algorytm znajdował 225 z wymaganych 4111 par. Można zatem było stwierdzić, że zakończy on swoje działanie nie szybciej niż w ciągu 5 godzin. W rzeczywistości czas ten będzie oczywiście dużo dłuższy: im dalej oddalamy się od początkowego przedziału, tym mniej znajdujemy par (jako że wartości $Q(x)$ rosną, a więc maleje szansa że będą B-gładkie). Potwierdzeniem tego faktu jest wykres przedstawiony na rysunku 4.7 uzyskany w ramach przebiegu algorytmu dla liczby T_{50}. Przedstawia on całkowitą ilość znalezionych wartości po przesianiu kolejnych przedziałów. Jak widać, szybkość ich przyrastania wyraźnie maleje wraz z postępem działania.



Rysunek 4.7: Przebieg algorytmu przy faktoryzacji 50-cyfrowej liczby - ilość liczb B-gładkich

Do otrzymania wyniku konieczne było przesianie 369 przedziałów o rozmiarze 22149631 (w wyniku ograniczenia przez ilość dostępnej pamięci). Czas działania wyniósł 41609947 ms, czyli około

$11,5$ godziny. Warto zauważyć, że ostateczna macierz miała prawie $8 \cdot 10^{45}$ rozwiązań, widać więc że w rzeczywistości potrzebowaliśmy mniej par niż zakładano. Gdyby nie oszczędność pamięci, nasz przesiewany (jednokrotnie) przedział przybrałby postać $[-69477219631, 69477219631]$, gdyż $M(T_{50}) = 69477219631$. W związku z tym, że aby otrzymać 4111 par przesialiśmy (fragmentami) łączny przedział $[-4086606919, 4086606919]$ (około 10 razy mniejszy) można się zastanawiać czy "idealna" implementacja (bez ograniczeń pamięciowych) nie okazałaby się gorsza. Należy jednak pamiętać o poniesionym w algorytmie narzucie na zwalnianie/alokacje pamięci, rozwiązywanie równań, itp. Ciężko zatem stwierdzić, która z wersji byłaby bardziej optymalna.

Warto zauważyć, że cały proces rozwiązywania macierzy zajął mniej niż pół minuty, co potwierdza brak konieczności stosowania wydajniejszych algorytmów niż GEM (Metoda Eliminacji Gaussa) w naszym przypadku.

Zrezygnowano z prób faktoryzacji większych liczb. Piętnastominutowy test wykazał, że konieczne będzie co najmniej 17 godzin na faktoryzacje liczby T_{55} (55-cyfrowe)j i co najmniej 44 godziny dla faktoryzacji liczby T_{60} (60-cyfrowej). Wyniki otrzymane przy badaniu liczby T_{50} pokazują, że rzeczywisty czas byłby prawdopodobnie dwukrotnie dłuższy, a więc grubo przekraczałby umowne sensowne granice.

Wyniki są dość zadowalające. W 1970 roku nie była możliwa faktoryzacja liczb 20-cyfrowych, a granica 50 cyfr została przekroczona dopiero na początku lat 80 [15]. 50-cyfrowa liczba może się wydawać niewielka w porównaniu do obecnego rekordu 232 cyfr, a nawet dla górnej granicy Sita Kwadratowego - 150 cyfr. Należy jednak pamiętać, że obliczenia dla wartości tego rodzaju zawsze są przeprowadzane na specjalnie skonstruowanych superkomputerach lub w środowiskach rozproszonych. Nawet mocne urządzenia klasy PC nie byłyby w stanie osiągnąć dużo lepszych rezultatów.

4.3.2. Benchmark

Przeprowadzono testy wydajnościowe 14 urządzeń działających pod kontrolą systemu Android. W skład tej grupy wchodziły zarówno urządzenia budżetowe, jak i najnowsze dostępne na rynku modele. Poza telefonami uruchomiono benchmark także na dwóch różnych tabletach.

Benchmark polega na faktoryzacji za pomocą Sita Kwadratowego trzech 16-cyfrowych liczb półpierwszych (4.6).

$$\begin{aligned} 9754399201265819 &= 98764327 \cdot 98764397 \\ 9754408090061549 &= 98764397 \cdot 98764417 \\ 9754410657936391 &= 98764423 \cdot 98764417 \end{aligned} \tag{4.6}$$

W każdym przypadku testowym uruchomiono benchmark, zignorowano pierwszy rezultat i od razu uruchomiono go kolejny raz. W taki sposób drugi z wyników nie uwzględnia czasu potrzebnego na zwolnienie pamięci przez procesy pozostające w tle, a jedynie realny czas obliczeń.

Lista testowanych urządzeń wraz z wynikami przedstawiona jest w tabeli 4.1, natomiast wizualne porównanie znajduje się na rysunku 4.8.

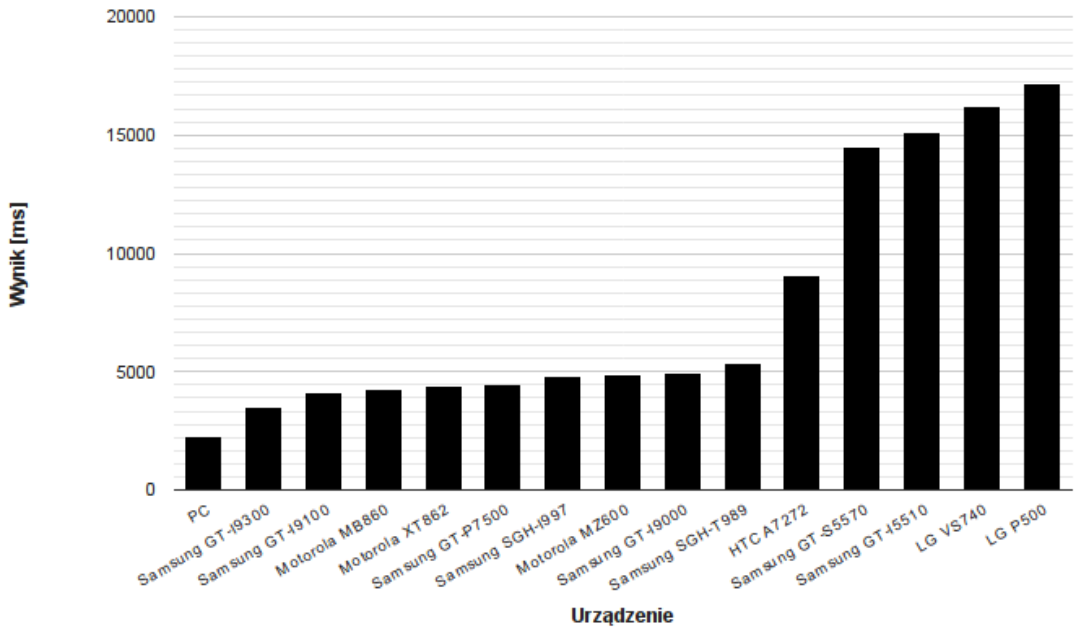
Patrząc na wyniki można od razu zauważyć, że taktowanie procesora to nie wszystko. Dla przykładu mimo pozornie słabszej obliczeniowo jednostki i trochę mniejszej ilości pamięci RAM GT-I9300 zdo-

Model	Wersja systemu		Procesor		Pamięć operacyjna	**Wynik**
	Jądro	Android OS	Model (Rdzeń)	Taktowanie		
Samsung Galaxy Mini GT-S5570	2.6.32	2.2 Froyo	MSM7227 (1 x ARM1136EJ-S)	600 MHz	286068 kB	**14525 ms**
Samsung Galaxy 551 GT-I5510	2.6.32	2.2 Froyo	MSM7227 (1 x ARM1136EJ-S)	600 MHz	290200 kB	**15095 ms**
Samsung Galaxy S GT-I9000	2.6.35	2.3.7 Gingerbread	S5PC110 (1 x ARM Cortex-A8)	1000 MHz	346088 kB	**4959 ms**
Samsung Galaxy S Infuse 4G SGH-I997	2.6.32	2.2 Froyo	S5PC111 Exynos 3110 (1 x ARM Cortex-A8)	1200 MHz	438848 kB	**4809 ms**
Samsung Galaxy S II GT-I9100	3.0.15	4.0.3 Ice Cream Sandwich	S5PC210 Exynos 4210 (2 x ARM Cortex-A9)	1200 MHz	850540 kB	**4132 ms**
T-Mobile Samsung Galaxy S II SGH-T989	2.6.35	2.3.6 Gingerbread	APQ8060 (2 x Qualcomm Scorpion)	1500 MHz	803212 kB	**5366 ms**
Samsung Galaxy S III GT-I9300	3.0.15	4.0.4 Ice Cream Sandwich	Exynos 4412 (4 x ARM Cortex-A9)	1400 MHz	798512 kB	**3513 ms**
Samsung Galaxy Tab 10.1 GT-P7500	2.6.36	3.2 Honeycomb	Tegra 2 250 T20 (2 x Cortex-A9)	1000 MHz	741516 kB	**4433 ms**
Motorola XOOM MZ600	2.6.39	4.0.3 Ice Cream Sandwich	Tegra 2 250 T20 (2 x Cortex-A9)	1000 MHz	735056 kB	**4857 ms**
HTC Desire Z A7272	2.6.35	2.3.3 Gingerbread	MSM7230 (1 x Qualcomm Scorpion)	800 MHz	377200 kB	**9064 ms**
Motorola Atrix 4G MB860	2.6.32	2.3.6 Gingerbread	Tegra 2 250 AP20H (2 x Cortex-A9)	1000 MHz	836220 kB	**4239 ms**
Motorola DROID 3 XT862	2.6.35	2.3.4 Gingerbread	OMAP 4430 (2 x Cortex-A9)	1000 MHz	449788 kB	**4384 ms**
LG Ally VS740	2.6.32	2.2 Froyo	MSM7627 (1 x ARM1136EJ-S)	600 MHz	173848 kB	**16190 ms**
LG Optimus One P500	2.6.32	2.2 Froyo	MSM7227 (1 x ARM1136EJ-S)	600 MHz	428704 kB	**17188 ms**
PC	3.3.0	–	Intel Pentium 4 (1)	2800 MHz	1019916 kB	**2239 ms**

Tablica 4.1: Wyniki testów

łał uzyskać wynik lepszy niż SGH-T989 o prawie dwie sekundy. Prawdopodobne przyczyny takiego rezultatu, to nowsza wersja systemu operacyjnego, a także dwukrotnie większa ilość rdzeni. Mimo, że aplikacja działa w ramach jednego wątku, Android wykonuje wiele zadań w tle (których to zadań nie można zatrzymać). Więcej rdzeni procesora oznacza, że zadania te mają mniejszy wpływ na działanie benchmarku. Nawet jednak biorąc pod uwagę te przyczyny okazuje się, że jeszcze słabsze urządzenia jak GT-I9100, a nawet GT-I9000 i SGH-I997 poradziły sobie lepiej mimo takiej samej (lub mniejszej) ilości rdzeni. Okazuje się więc, że procesory Exynos od firmy Samsung sprawdzają się lepiej w ciężkich obliczeniach tego typu niż te produkowane przez firmę Qualcomm.

Kolejnym spostrzeżeniem jest fakt, że urządzenia MB8600, MZ600 i GT-P5700 mimo procesora o tym samym taktowaniu pokonały w benchmarku GT-I9000. Jednym z powodów jest wspomniana wcześniej obecność dodatkowego rdzenia w procesorach Tegra 2, ale może istnieć też inna przyczyna.



Rysunek 4.8: Wyniki testów - wykres

GT-I9000 ma dwukrotnie mniej pamięci operacyjnej niż wyżej wspomniani konkurenci. Jak opisano w rozdziale 4.1.1, mniejsza jej ilość może powodować konieczność użycia więcej niż jednego przedziału, na którym dokonujemy przesiewania. Szczególnie objawia się to w testach słabszych urządzeń, takich jak GT-S5570, GT-I5510, czy VS740. Mimo procesora tylko dwukrotnie słabszego, wyniki benchmarku są słabsze trzykrotnie niż dla SGH-I997. Warto też zauważyć, że na przykład telefon A7272, mimo niewiele lepszego (pod względem taktowania) procesora osiąga zauważalnie lepsze od nich rezultaty dzięki dużo większej ilości pamięci RAM.

Ostatni wiersz tabeli przedstawia wyniki uruchomienia testu na komputerze stacjonarnym o parametrach typowych dla przeciętnej domowej maszyny. W celu przeprowadzenia obliczeń skompilowano część napisaną w C na maszynie PC i użyto jej za pomocą zewnętrznego wywołania. W związku z tym porównanie jest bardzo dokładne, gdyż uruchamiany jest ten sam kod (z dokładnością do optymalizacji różnych kompilatorów). Jak widać, dwa razy szybszy pod względem taktowania procesor stacjonarny wygrywa z najszybszym z telefonów o mniej więcej sekundę. Z jednej strony różnica ta jest nie mała, z drugiej jednak nie jest dwukrotna, co sugeruje, że procesor obecny w GT-I9300 charakteryzuje się lepszą budową wewnętrzną. Na pewno też wyniki wyglądałyby inaczej, gdyby wykorzystać wszystkie cztery rdzenie tego ostatniego. Porównanie komputera stacjonarnego z jednym z topowych na dzień dzisiejszy telefonów potwierdza tezę, że o ile te pierwsze urządzenia są wciąż dominujące pod względem obliczeniowym, to te drugie gonią je w zawrotnym tempie.

4.4. Wnioski

Osiągnięto wszystkie założone cele pracy. Została potwierdzona teza, że urządzenia mobilne znajdują się już nie tak bardzo w tyle w stosunku do komputerów osobistych jeśli chodzi o moc oblicze-

niową. Oczywiście, istnieją granice, ale obserwując wzrost możliwości telefonów w przeciągu ostatnich lat można być pewnym, że szybko zostaną one usunięte.

Dzisiejszy telefon ze średniej półki - Samsung Galaxy S - jest w stanie rozłożyć na czynniki pierwsze 50-cyfrową liczbę w rozsądnym czasie. Wynik dość ciekawy zważywszy na fakt, że większość ludzi nosi dziś podobne urządzenia w kieszeni. Oczywiście, problem faktoryzacji nie został rozwiązany całkowicie i dzisiejsze metody kryptograficzne są póki co całkowicie bezpieczne z tego punktu widzenia.

Dalszym rozwojem dla niniejszej pracy byłoby w pierwszej kolejności dalsza optymalizacja algorytmu. Użycie MPQS, czy wariacji z dużymi czynnikami pierwszymi mogłoby dać lepsze rezultaty. Oczywiście, większość rekordów pobitych przy udziale Sita Kwadratowego używała tych właśnie modyfikacji. Dodatkowo można by usprawnić wyliczanie rozwiązań używając skuteczniejszej metody niż Eliminacja Gaussa.

Najciekawszy kierunek rozwoju wygląda jednak inaczej. Jak zostało już nie raz wspomniane, algorytm Sita Kwadratowego (a w szczególności MPQS) bardzo dobrze się zrównolegla (z wyłączeniem rozwiązywania układu równań). Konkretniej, największy z rekordów tej metody (zanim oddała ona palmę pierwszeństwa algorytmowi GNFS) - faktoryzacja liczby 129-cyfrowej padł dzięki setkom wolontariuszy podłączonych do internetu i ich maszynom [5]. Od razu nasuwa się pomysł, aby tego samego rodzaju koncepcje zastosować dla telefonów. Po pierwsze, jak już udowodniono są one jeszcze wydajniejsze niż komputery na początku lat 90. Po drugie, w naturalny sposób są one non stop podłączone do sieci i aktywne, w przeciwieństwie do komputerów, które większość ludzi przynajmniej okazyjnie wyłącza. Po trzecie są co najmniej tak samo liczne (jeśli nie liczniejsze) niż urządzenia stacjonarne. Użycie smartfonów jako węzłów obliczeniowych jawi się więc jako bardzo ciekawa inicjatywa. Nie należy się spodziewać pobicia rekordu 232 cyfr (nie z użyciem metody QS w każdym razie), wyniki byłyby jednak na pewno interesujące.

Bibliografia

[1] R. A. Mollin, "A Brief History of Factoring and Primality Testing B. C. (Before Computers)," *Mathematics Magazine*, vol. 75, no. 1, pp. 18–29, 2002.

[2] C. Pomerance, "The quadratic sieve factoring algorithm," in *Advances in Cryptology* (T. Beth, N. Cot, and I. Ingemarsson, eds.), vol. 209 of *Lecture Notes in Computer Science*, pp. 169 – 182, Springer-Verlag Berlin Heidelberg, 1985.

[3] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124 – 134, 1994.

[4] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, p. 883–887, 2001.

[5] D. Atkins, M. Graff, A. K. Lenstra, and P. C. Leyland, "The magic words are squeamish ossifrage," in *Proceedings of ASIACRYPT '94*, vol. 917 of *Lecture Notes in Computer Science*, pp. 261–277, 1995.

[6] M. Gardner, "Mathematical games: a new kind of cipher that would take millions of years to break," *Scientific American*, vol. 237, pp. 120–124, 1977.

[7] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on Graphics Cards," in *Advances in Cryptology - Eurocrypt 2009 (28th Annual International Conference on the Theory and Applications of Cryptographic Techniques)* (A. Joux, ed.), vol. 5479 of *Lecture Notes in Computer Science*, pp. 483–501, Springer, 2009.

[8] D. Bonenberger and M. Krone, "Factorization of RSA-170," tech. rep., Department of Computer Science Ostfalia University of Applied Sciences, 2010.

[9] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang, "The Billion-Mulmod-Per-Second PC," in *SHARCS*, pp. 131–144, 2009.

[10] J. W. Bos, M. E. Kaihara, and P. L. Montgomery, "Pollard Rho on the PlayStation 3," in *Workshop record of SHARCS 2009*, pp. 35–50, 2009.

[11] M. Kraitchik, *Théorie des Nombres*, vol. Tome II. Paris: Gauthier-Villars, 1924.

[12] J. D. Dixon, "Asymptotically Fast Factorization of Integers," *Mathematics of Computation*, vol. 36, no. 153, pp. 255–260, 1981.

[13] D. H. Lehmer and R. E. Powers, "On factoring large numbers," *Bulletin of the American Mathematical Society (New Series)*, vol. 37, no. 10, pp. 770–776, 1931.

[14] M. A. Morrison and J. Brillhart, "A Method of Factoring and the Factorization of F_7," *Mathematics of Computation*, vol. 29, no. 129, pp. 183–205, 1975.

[15] C. Pomerance, "A Tale of Two Sieves," *Notices of the AMS*, vol. 43, no. 12, pp. 1473–1485, 1996.

[16] A. K. Lenstra and H. W. Lenstra, Jr., *The Development of the Number Field Sieve*, vol. 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, 1993.

[17] "The Cunningham Project." `http://homes.cerias.purdue.edu/~ssw/cun/`.

[18] "The RSA Factoring Challenge." `http://www.rsa.com/rsalabs/node.asp?id=2093`.

[19] F. Bahr, M. Boehm, J. Franke, and T. Kleinjung, "RSA-640 is factored!," 2005. `http://www.crypto-world.com/announcements/rsa640.txt`.

[20] F. Bahr, M. Boehm, J. Franke, and T. Kleinjung, "Factorization of RSA-200," 2005. `http://www.loria.fr/~zimmerma/records/rsa200`.

[21] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit RSA modulus." Cryptology ePrint Archive, Report 2010/006, 2010. `http://eprint.iacr.org/`.

[22] "Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth." `http://www.gartner.com/it/page.jsp?id=1924314`.

[23] "Smartphone Market Hits All-Time Quarterly High Due To Seasonal Strength and Wider Variety of Offerings, According to IDC." `http://www.idc.com/getdoc.jsp?containerId=prUS23299912`.

[24] "More US Consumers Choosing Smartphones as Apple Closes the Gap on Android." `http://blog.nielsen.com/nielsenwire/consumer/more-us-consumers-choosing-smartphones-as-apple-closes-the-gap-on-android/`.

[25] "comScore Releases the "2012 Mobile Future in FocusŻeport." `http://www.comscore.com/Press_Events/Press_Releases/2012/2/comScore_Releases_the_2012_Mobile_Future_in_Focus_Report?piCId=66038`.

[26] "Android Developers." `http://developer.android.com/index.html`.

[27] C. Pomerance, "Analysis and comparison of some integer factoring algorithms," in *Computational Methods in Number Theory: Part 1* (H. W. Lenstra, Jr. and R. Tijdeman, eds.), vol. 154 of *Mathematical Centre Tract*, pp. 89–139, 1982.

[28] E. R. Canfield, P. Erdös, and C. Pomerance, "On a problem of Oppenheim concerning "factorisatio numerorum"," *Journal of Number Theory*, vol. 17, no. 1, pp. 1 – 28, 1983.

[29] D. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE Transactions on Information Theory*, vol. 32, no. 1, pp. 54–62, 1986.

[30] C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *Journal of Research of the National Bureau of Standards*, vol. 45, no. 4, pp. 255–282, 1950.

[31] H. Riesel, "Prime Numbers and Computer Methods for Factorization," *Birkhäuser Progress in Mathematics*, vol. 57, 1985.

[32] J. Davis and D. Holdridge, "Factorization using the quadratic sieve algorithm," tech. rep., Sandia National Laboratories, 1983.

[33] E. Landquist, "The Quadratic Sieve Factoring Algorithm," *Math*, 2001.

[34] R. D. Silverman, "The Multiple Polynomial Quadratic Sieve," *Mathematics of Computation*, vol. 48, no. 177, pp. 329–339, 1987.

[35] S. Goldwasser, J. C. Lagarias, A. K. Lenstra, K. S. McCurley, and A. M. Odlyzko, *Cryptology and Computational Number Theory*. American Mathematical Society, 1990.

[36] O. Åsbrink and J. Brynielsson, "Factoring large integers using parallel Quadratic Sieve," 2000.

[37] T. R. Caron and R. D. Silverman, "Parallel implementation of the quadratic sieve," *The Journal of Supercomputing*, vol. 1, pp. 273–290, 1988.

[38] "The GNU Multiple Precision Arithmetic Library." `http://gmplib.org/`.

[39] I. Šimeček and P. Kováč, "An Overview of Factorization of Large Integers Using the GMP Library," tech. rep., Department of Computer Science , Faculty of Electrical Engineering, Czech Technical University.