Tampereen yliopisto

Joonas Pelttari

C++ -KIRJASTO SUURIEN LUKUJEN OPERAATIOIHIN

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Toukokuu 2021

TIIVISTELMÄ

Joonas Pelttari: C++ -kirjasto suurien lukujen operaatioihin
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Kandidaatintutkielma
Toukokuu 2021

Tässä työssä toteutetaan C++ -ohjelmointikielellä kirjasto, jolla voidaan laskea perusoperaatioita suuremmille luvuille kuin tavallisesti kokonaislukuja (engl. integer) käyttämällä pystyttäisiin. Toteutettu ohjelma käyttää lukujen käsittelyssä ja tallentamisessa merkkijonoja (engl. string), mikä mahdollistaa suurien lukujen operaatioiden laskemisen merkkijonon maksimipituuden ollessa 2^{32}-1 eli 4 294 967 295 merkkiä. Vertailuna C++ -kielen suurin mahdollinen kokonaisluku on 20 merkkiä pitkä (18 446 744 073 709 551 615).

Toteutetut operaatiot ovat yhteen-, vähennys- ja kertolasku, mutta tämän lisäksi työhön toteutettiin myös mahdollisuus suorittaa Karatsuba-algoritmi eripituisille luvuille ja näin todeta sen toteutuksen olevan linjassa asymptoottisen suoritusajan kanssa.

Työssä käsitellään algoritmien ja niiden pseudokoodien teoriaa sekä toteutetaan C++ -ohjelmointikielelle käsiteltävät algoritmit sillä tavalla, että niiden vertaaminen esitettyyn pseudokoodiin on mahdollista ja helpohkoa C++ -ohjelmointikieltä osaavalle henkilölle. Tämä palvelee työssä lyhyesti mainittua ideaa siitä, että työtä olisi mahdollista hyödyntää toteutettaessa vastaavat operaatiot jollekin toiselle ohjelmointikielelle.

Algoritmien asymptoottiset suoritusajat suuria lukuja käsittelevissä operaatioissa ovat tärkeässä osassa, joten niitä tarkastellaan sekä teoreettisella että käytännön tasolla. Käytännön tarkastelussa suoritetaan toteutettua C++ -ohjelmaa ja lasketaan sieltä saatavat asymptoottiset suoritusajat verraten niitä teoreettisiin. Käytännön tasolla on myös tärkeää huomioida, että parempi asymptoottinen tehokkuus ei välttämättä tee algoritmista toista nopeampaa, koska suuret vakiotermit saattavat jäädä huomioimatta ja siksi käytännössä tehokkuus riippuu käsiteltävien lukujen suuruudesta.

Avainsanat: arithmetic, integer, algorithm, asymptotic

SISÄLLYSLUETTELO

1. JOHDANTO

Tämän työn tarkoitus on käydä läpi teoriaa algoritmeista, joiden pohjalta luodaan funktiot käsittelemään suuria lukuja, joita ei käsiteltävällä ohjelmointikielellä itsellään ole mahdollista laskea. Käytännön tasolla työssä on tarkoitus toteuttaa operaatiot läpikäydyn teorian pohjalta C++ -ohjelmointikielellä niin, että vastaavanlaisten operaatioiden/funktioiden toteuttaminen myös muilla ohjelmointikielillä olisi mahdollista tämän työn avulla. Käsittely etenee operaatio kerrallaan esitellen teorian sekä siihen pohjautuvan toteutuksen C++ -kielellä.

Tehokkuus on avainasemassa käsiteltävien lukujen koon kasvaessa, minkä vuoksi työssä kiinnitetään huomiota sekä algoritmien asymptoottiseen tehokkuuteen, että käytännön toteutuksessa tehdyn C++ -ohjelman tehokkuuteen.

C++ -kielelle on jo entuudestaan olemassa lukuisia kirjastoja suurien lukujen käsittelyyn, esimerkiksi GNU MP [1]. Tämän työn tarkoitus ole kilpailla niitä vastaan esimerkiksi suorituskyvyssä. Tämä työ tarjoaa sen sijaan teoriaan pohjautuvan lähestymistavan tällaisten operaatioiden kehittämiseen ja olemassa olevien algoritmien vertailuun. Työhön tehty C++ -ohjelma löytyy Githubista [2].

2. ASYMPTOOTTISET SUORITUSAJAT

Laskuoperaatioiden suorittaminen hidastuu käsiteltävien lukujen kasvaessa. Siksi on merkityksellistä huomioida laskuissa käytettävien algoritmien asymptoottiset tehokkuudet, jotta ohjelman tehokkuus ei kärsi huonojen algoritmien vuoksi. Taulukkoon 1 on koottu yleisimpien laskuoperaatioissa käytettyjen algoritmien asymptoottisia suoritusaikoja. Operaatiot suoritetaan kahdelle *n*-pituiselle kokonaisluvulle.

Taulukko 1. *Operaatioissa käytettävien algoritmien asymptoottiset suoritusajat* [3, s. 514][4, s. 15]

Operaatio	Algoritmi	Asymptoottinen suoritusaika
Yhteenlasku	Schoolbook	$O(n)$
Vähennyslasku	Schoolbook	$O(n)$
Kertolasku	Schoolbook	$O(n^2)$
	Karatsuba	$O(n^{\log_2 3} \approx n^{1.585})$
	Toom–Cook-3	$O(n^{\log_3 5} \approx n^{1.465})$
	Schönhage–Strassen	$O(n \log n \log\log n)$

3. KOKONAISLUKUJEN ALGORITMIT

Toteutetussa C++ -ohjelmassa algoritmit on tehty käyttäen kymmenjärjestelmää, vaikka luvun 3 alaluvuissa esitellyt algoritmit mahdollistaisivat myös muiden järjestelmien käytön, esim. binäärijärjestelmän [4].

3.1 Yhteenlasku kokonaisluvuilla

Algorithm 1.1 IntegerAddition

Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_i\beta^i$, carry-in $0 \leq d_{\text{in}} \leq 1$
Output: $C := \sum_0^{n-1} c_i\beta^i$ and $0 \leq d \leq 1$ such that $A + B + d_{\text{in}} = d\beta^n + C$

1: $d \leftarrow d_{\text{in}}$
2: **for** i **from** 0 **to** $n - 1$ **do**
3: $\quad s \leftarrow a_i + b_i + d$
4: $\quad (d, c_i) \leftarrow (s \text{ div } \beta, s \bmod \beta)$
5: return C, d.

Ohjelma 1. *Schoolbook-algoritmin pseudokoodi yhteenlaskulle [4]*

Yhteenlaskussa (ohjelma 1) algoritmi käy läpi laskettavia lukuja merkki kerrallaan aloittaen pienimmistä. Rivillä 2 esitetty silmukka suoritetaan *n* kertaa, missä *n* tarkoittaa algoritmille annetuista numeroista lyhyemmän pituutta. Rivillä 3 lasketaan yhteen samalla saman kantaluvun numerot ja niihin lisätään siirtobitti (engl. carry-bit), jonka arvo on aina 0 tai 1 käytettäessä kymmenjärjestelmää. Mikäli saatu summa *s* on suurempi kuin 9, seuraavan silmukan siirtobitti on 1 ja tämän silmukan summasta *s* vähennetään kymmenen, joka viedään siirtobitillä seuraavalle silmukalle.

3.2 Vähennyslasku kokonaisluvuilla

Vähennyslaskun algoritmi toimii lähes samalla tavalla verrattuna ohjelman 1 yhteenlaskuun. Erona on rivin 3 muuttaminen muotoon $s \leftarrow a_i - b_i + d$ [4, s. 3]. Tämä on toteutetussa ohjelmassa muokattu vielä niin, että *d* vähennetään eli saadaan $s \leftarrow a_i - b_i - d$.

Vähennyslaskussa huomioitavaa on mahdollisuus negatiivisiin vastauksiin, joten C++ -toteutuksessa funktiolle voidaan antaa kolmantena parametrina totuusarvo (engl. bool), mikäli miinusmerkkiä ei haluta. Tämä on hyödyllistä, koska vähennyslaskua käytetään

kertolaskualgoritmissa (luku 3.3.2), joka tarvitsee kahden luvun erotuksen, mutta ilman miinusmerkkiä.

3.3 Kertolasku kokonaisluvuilla

Kertolaskuoperaatiossa tarkastellaan kolmea eri algoritmia, jotka ovat tavanomainen (engl. Schoolbook), Karatsuba ja Toom-Cook-3. Näistä parhaimman asymptoottisen suorituskyvyn tarjoaa Toom-Cook-3 (taulukko 1). Taulukosta 1 löytyvä Schönhage–Strassen-algoritmi on asymptoottisesti vielä parempi kuin Toom-Cook-3, mutta sitä ei käsitellä tässä työssä, koska O-notaatio jättää huomioimatta vakiokertoimen, jonka merkitys käytännön tehokkuuteen on suuri. [5]

Verrattuna tavanomaiseen kertolaskualgoritmiin Karatsuba tarjoaa paremman asymptoottisen suoritusajan (taulukko 1), mikä on merkittävä etu, kun tarkoituksena on käsitellä suuria lukuja.

Toom-Cook-3 on teoriassa Karatsuba-algoritmia tehokkaampi, mutta käytännön toteutuksissa sen käyttö jää huomattavasti vähäisemmäksi, koska se vaatii tarkkoja jakolaskuja (engl. exact division). Syy tarkkojen jakolaskujen välttelyyn on se, että niiden tehokas toteuttaminen on vaikeaa. [6]

3.3.1 Schoolbook

Tavanomainen kertolasku (ohjelma 2) sisältää kaksi silmukkaa, jotka käyvät numeroita läpi yksi kerrallaan kertoen ne toisillansa. Johtuen kahdesta sisäkkäisestä silmukasta, asymptoottinen suoritusaika on eksponentiaalinen, kuten todettu taulukossa 1. Tavanomainen kertolasku löytyy toteutetusta C++ -ohjelmasta (*MultiplySlow*), mutta se häviää tehokkuudessa luvussa 3.3.2 käsiteltävälle Karatsuba-algoritmille jo hyvinkin pienillä luvuilla.

Algorithm 1.2 BasecaseMultiply

Input: $A = \sum_0^{m-1} a_i\beta^i,\ B = \sum_0^{n-1} b_j\beta^j$
Output: $C = AB := \sum_0^{m+n-1} c_k\beta^k$

1: $C \leftarrow A \cdot b_0$
2: **for** j **from** 1 **to** $n-1$ **do**
3: $\quad C \leftarrow C + \beta^j(A \cdot b_j)$
4: return C.

Ohjelma 2. *Schoolbook-algoritmin pseudokoodi kertolaskulle [4, s. 4]*

3.3.2 Karatsuba

Verrattuna luvussa 3.3.3 käsiteltävään Tom-Cook-3 -algoritmiin Karatsuba on suhteellisen yksinkertainen toteuttaa, joten se palvelee tämän työn tarkoitusta toimia mahdollisena apuna kehitettäessä vastaavaa operaatiokirjastoa muihin ohjelmointikieliin tai projekteihin.

Karatsuba-algoritmi perustuu n-pituisen kertolaskun jakamiseen kolmeksi *n/2*-pituiseksi kertolaskuksi. Nämä kolme kertolaskua suoritetaan kutsuen Karatsuba-algoritmia rekursiivisesti uusilla parametreilla, jotka lasketaan käyttäen alkuperäisten lukujen maksimipituuden *n* puolikasta ylöspäin pyöristettynä. [4, s. 5]. Algoritmin tarkoitus on siis vähentää osittaisten tulosten määrää pitkissä kertolaskuissa vaadittavien yhteenlaskujen kustannuksella. Huomioitavaa tässä ”vaihtokaupassa” on modernien prosessorien kyky laskea kertolaskuja lähes yhtä nopeasti kuin yhteenlaskuja. [7]. Toteutetussa C++ -ohjelmassa myös yhteenlasku on toteutettu omana algoritminaan, joten sen tehokkuutta täytyisi arvioida erikseen.

Tässä työssä käytetään Karatsuba-algoritmin perinteisempää toteutustapaa *School-boy*, mutta on olemassa myös vähemmän tunnettu variaatio *arbitrary degree Karatsuba* (ADK), jonka mahdollinen paremmuus perustuu tarvittavien kerto- ja yhteenlaskujen erilaiseen suhteeseen. ADK:n paremmuus ei kuitenkaan ole itsestäänselvyys vaan riippuvainen olosuhteista, joten sen syvällisempi käsittely jätetään tässä työssä pois. [7].

Ohjelma 3 esittää Karatsuba-algoritmin toiminnan. Algoritmille annetaan parametrina kaksi kokonaislukua, joiden tulo palautetaan.

Liitteessä A esitetään Karatsuba-algoritmista C++ -kielen toteutus, joka on tehty käyttäen apuna ohjelman 3 pseudokoodia. Tähän toteutukseen vaaditaan mahdollisuus yhteen- ja vähennys- ja kertolaskuun. Yhteen- ja vähennyslaskun toteutukset ovat esitetty luvuissa 3.1 ja 3.2. Kertolaskussa käytetään luvussa 3.3.1 käsiteltävää tavanomaista kertolaskualgoritmia. Tämä esitetään ohjelmassa 3 nimellä **BasecaseMultiply.**

Algorithm 1.3 KaratsubaMultiply

Input: $A = \sum_0^{n-1} a_i\beta^i$, $B = \sum_0^{n-1} b_j\beta^j$
Output: $C = AB := \sum_0^{2n-1} c_k\beta^k$

if $n < n_0$ **then** return **BasecaseMultiply**(A, B)
$k \leftarrow \lceil n/2 \rceil$
$(A_0, B_0) := (A, B) \bmod \beta^k$, $(A_1, B_1) := (A, B) \operatorname{div} \beta^k$
$s_A \leftarrow \operatorname{sign}(A_0 - A_1)$, $s_B \leftarrow \operatorname{sign}(B_0 - B_1)$
$C_0 \leftarrow$ **KaratsubaMultiply**(A_0, B_0)
$C_1 \leftarrow$ **KaratsubaMultiply**(A_1, B_1)
$C_2 \leftarrow$ **KaratsubaMultiply**$(|A_0 - A_1|, |B_0 - B_1|)$
return $C := C_0 + (C_0 + C_1 - s_A s_B C_2)\beta^k + C_1\beta^{2k}$.

Ohjelma 3. *Karatsuba-algoritmin pseudokoodi [4, s. 5]*

3.3.3 Toom-Cook-3

Toom-Cook-algoritmi jakaa *n*-pituisen tulon useaan eri tuloon, joiden määrä riippuu käytettävästä Toom-Cook:in variaatiosta, jota merkitään numerolla *r* algoritmin lopussa (Toom-Cook-r). Jaettujen tulojen määrä saadaan kaavalla

$$2r - 1, \tag{1}$$

jossa *r* on mainittu Toom-Cook:in variaatio. [4, s. 7]

Tässä luvussa esitellään Toom-Cook-algoritmin ”3-suuntainen” variaatio, joka tarkoittaa kaavalla 1 laskettuna tulon jakamisen viiteen. Ohjelmassa 4 kertolaskun jako viiteen havaitaan käytännössä riveillä 3-7, joissa Toom-Cook-3 -algoritmia kutsutaan rekursiivisesti viisi kertaa.

Toom-Cook -variaatioiden asymptoottinen suoritusaika saadaan kaavalla

$$O(n^{\log_r(2r-1)}), \tag{2}$$

josta saadaan tulokseksi taulukossa 1 esitetty $O(n^{\log_3 5})$ käytettäessä arvoa 3 muuttujalle *r*. Kaavan 2 perusteella asymptoottinen suoritusaika voi teoriassa lähestyä lineaarisuutta muuttujan *r* kasvaessa. Käytännön toteutuksissa käytetyt arvot ovat kuitenkin yleensä 2, 3 tai 4 johtuen Schönhage–Strassen-algoritmin paremmasta asymptoottisesta suoritusajasta (taulukko 1). [8]

Luvussa 3.3.2 käsiteltävä Karatsuba-algoritmi on itseasiassa yleistys ”2-suuntaisesta” Toom-Cook-algoritmista, sillä siinä kertolasku jaetaan kolmeen pienempään kertolaskuun kuten kaavasta 1 voidaan laskea käyttämällä arvoa 2 muuttujalle *r*. [4, s. 6–7]

Algorithm 1.4 ToomCook3

Input: two integers $0 \leq A, B < \beta^n$

Output: $AB := c_0 + c_1\beta^k + c_2\beta^{2k} + c_3\beta^{3k} + c_4\beta^{4k}$ with $k = \lceil n/3 \rceil$

Require: a threshold $n_1 \geq 3$

1: **if** $n < n_1$ **then** return **KaratsubaMultiply**(A, B)

2: write $A = a_0 + a_1x + a_2x^2$, $B = b_0 + b_1x + b_2x^2$ with $x = \beta^k$.

3: $v_0 \leftarrow$ **ToomCook3**(a_0, b_0)

4: $v_1 \leftarrow$ **ToomCook3**$(a_{02}+a_1, b_{02}+b_1)$ where $a_{02} \leftarrow a_0+a_2, b_{02} \leftarrow b_0+b_2$

5: $v_{-1} \leftarrow$ **ToomCook3**$(a_{02} - a_1, b_{02} - b_1)$

6: $v_2 \leftarrow$ **ToomCook3**$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$

7: $v_\infty \leftarrow$ **ToomCook3**(a_2, b_2)

8: $t_1 \leftarrow (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$, $t_2 \leftarrow (v_1 + v_{-1})/2$

9: $c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$.

Ohjelma 4. *Toom-Cook-3 -algoritmin pseudokoodi [4, s. 7]*

4. TOTEUTETTU C++ -OHJELMA

Tässä luvussa käsitellään tarkemmin aiemmin sivuttua C++ -ohjelmaa [2]. Ohjelman tarkoitus on olla mahdollisimman yksinkertainen ja helppokäyttöinen, jotta sen voisi vaivatta lisätä esimerkiksi johonkin valmiiseen ohjelmaan ja aloittaa sen tarjoamien funktioiden käyttö. Funktiot ovat tehty jäljitellen esiteltyjen algoritmien pseudokoodia mahdollisimman hyvin sekä toiminnallisuuden että ulkonäön kannalta, jotta johdannossa mainittu mahdollisuus luoda vastaava operaatiokirjasto käyttäen tätä työtä apuna olisi mahdollista.

Toteutettu ohjelma on omana luokkanaan BigNumLib tiedostoissa *bignumlib.h* sekä *bignumlib.cpp*. Tämä luokka käyttää kirjastoja *<string>, <math.h> ja <algorithm>.* Ohjelmassa on sen lisäksi mukana *main.cpp*, joka sisältää esimerkin *BigNumLib:n* käyttämisestä sekä *testAsymptotic*-funktion, jolla voidaan havaita kertolaskualgoritmin tehokkuus samaan tapaan kuin luvun 4.2 taulukossa 2. Tämä funktio tarvitsee toimiakseen erillisen tiedoston *NumbersForTesting.h*.

Ohjelmaan ei ole toteutettu virhetarkasteluja funktioille annetuille syötteille, joten validien numeroiden ja syötteiden käyttäminen jää funktioiden kutsujan vastuulle.

4.1 Funktiot

string sum(string number1, string number2);

Suorittaa yhteenlaskun kahdelle kokonaisluvulle. Mikäli toinen luvuista on negatiivinen, muuttaa yhteenlaskun vähennyslaskuksi ja käyttää *substraction*-funktiota. Esimerkiksi lasku -10+7 voidaan ilmoittaa laskuna 7-10. Molempien lukujen ollessa negatiivisia suoritetaan tavallinen yhteenlasku positiivisilla luvuilla ja lisätään miinusmerkki lopussa.

string substraction(string number1, string number2, bool minusSign);

Suorittaa vähennyslaskun kahdelle positiiviselle kokonaisluvulle. Kolmantena parametrina annettava *minusSign* määrittelee, halutaanko lopulliseen vastaukseen miinusmerkki. Tämän parametrin ollessa epätosi (engl. false), ei vastauksessa ole miinusmerkkiä, vaikka se olisi negatiivinen.

string multiplyFast(string number1, string number2);

Suorittaa kertolaskun kahdelle positiiviselle kokonaisluvulle käyttäen Karatsuba-algoritmia. Tämä on suositeltu funktio kertolaskuun.

string multiplySlow(string number1, string number2);

Suorittaa kertolaskun kahdelle positiiviselle luvulle käyttäen Schoolbook-algoritmia. Tämä on lähes aina hitaampi tapa laskea kertolasku kuin edellä mainittu *multiplyFast*, joten tämän käyttöä ei suositella. Tämä funktio on nopeampi pienillä luvuilla, jotka voi kertoa yhteen ilmankin erillistä algoritmia käyttämällä tavallista kokonaislukujen kertolaskua.

4.2 Suorituskyky

Taulukkoon 2 on kirjattu ylös suoritusaikoja, kun *multiplyFast*-funktiolla suoritetaan tulo kahdelle *n*-pituiselle luvulle. Taulukon 2 tarkoituksena on havaita, että toteutetun kertolaskun suorituskyky vastaa hyvin lähelle sitä, mikä sen pitäisi teoriassakin olla.

Taulukko 2. *Kertolaskun suorituskyky käytännössä ja teoriassa*

n	Toteutunut aika (ms)	Teoreettinen aika (ms)
100	4	-
200	15	12
400	45	45
800	125	135
1600	366	375
3200	1138	1098
6400	3337	3414
12800	10283	10011

Teoreettiset arvot ovat laskettu käyttämällä Karatsuba-algoritmin asymptoottista suoritusaikaa (taulukko 1) ja toteutuneita aikoja. Esimerkiksi taulukossa 2 *n*-arvon ollessa 200 toteutuneeksi ajaksi saadaan 15 millisekuntia. Asymptoottisen suoritusajan avulla laskettaessa ajan pitäisi olla kolminkertainen, kun *n*-arvo on 400 eli kaksinkertainen.

5. KEHITYSIDEAT

Toteutettu ohjelma ei sisällä jakolaskua, joten se olisi hyvin looginen lisä ohjelmaan. Ohjelmassa ilmenneitten ongelmien takia vain yhteenlaskulle voi antaa negatiivisia lukuja parametrina. Tämä mahdollisuus pitäisi lisätä myös muille operaatioille.

Suunnitelmissa ollut aika-arvio -funktio ei myöskään toteutunut. Tämän funktion idea olisi arvioida halutun operaation kesto eri suuruisilla luvuilla. Käytännössä funktio suorittaisi operaation ennalta määritellyillä luvuilla ja ottaisi ylös tähän kuluvan ajan. Tämän jälkeen se arvioisi asymptoottisen suoritusajan ja käyttäjän lukujen koon perusteella operaation suorittamiseen kuluvan ajan.

Myös virhetarkastelut olisivat käytännön käyttötarkoituksia varten hyödyllisiä, jotta käyttäjän ei tarvitsisi liian tarkasti huolehtia syötteiden oikeellisuudesta. Näiden lisääminen ei ole monimutkaista, koska merkittävin asia on tarkistaa merkkijonojen sisältävän ainoastaan numeroita ja mahdollinen miinusmerkki ensimmäisenä alkiona.

Edistyksellisempiä kehitysmahdollisuuksia olisi esimerkiksi uudet algoritmit kertolaskua varten ja ohjata ohjelma käyttämään optimaalisinta syötteiden suuruuden mukaan tavoitteena minimoida käytetty aika. Tällainen hybridialgoritmi on käytössä esimerkiksi C++ -kielen Standard Template Library -kirjaston lajittelualgoritmissa (engl. sort) [9].

6. YHTEENVETO

Tässä kandidaatintutkielmassa tarkoitus oli käydä läpi teoriaa algoritmeista, joiden avulla laskuoperaatioita voidaan toteuttaa ja on nykymaailmassa toteutettu erilaisissa sovelluskohteissa. Käsittely jäi vajaaksi jakolaskun puuttuessa, mutta yhteen-, vähennys- ja kertolaskuun työssä saatiin toteutettua toimivat funktiot. Myös ohjelmasta löytyvä funktio kertolaskun asymptoottisen tehokkuuden tutkimiseen on toimiva ja todistaa asymptoottisen suoritusajan olevan ohjelman kertolaskussa linjassa teorian kanssa.

Yhteenvetona operaatioiden toteutuksesta voidaan todeta niiden olevan suhteellisen yksinkertaisia ja täten mahdollisia toteuttaa muilla ohjelmointikielillä, joihin ei ole saatavilla esimerkiksi GNU MP:n kaltaista kirjastoa.

LÄHTEET

[1] https://gmplib.org/. (viitattu 27.05.2021)

[2] https://github.com/JoonasPel/kandiOhjelma. (viitattu 31.05.2021)

[3] Muller J, Brisebarre N, de Dinechin F, Jeannerod C, Lefèvre V, Melquiond G, et al. Handbook of Floating-Point Arithmetic. 1st ed. Boston, MA: Birkhäuser Boston; 2010.

[4] Brent RP, Zimmermann P. Modern computer arithmetic. Cambridge: Cambridge University Press; 2011.

[5] Bodrato M. Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. 2007;4547:116-133.

[6] Gu Z, Li S. A Division-Free Toom-Cook Multiplication-Based Montgomery Modular Multiplication. TCSII 2019;66(8):1401-1405.

[7] Scott M. Missing a trick: Karatsuba variations. Cryptogr Commun 2018;10(1):5-15.

[8] Bodrato M, Zanoni A. What About Toom-Cook Matrices Optimality?: Universitá di Roma Tor Vergata; 2006.

[9] https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/. (viitattu 25.05.2021)

LIITE A: KARATSUBA C++ -KIELELLÄ

```
string multiply_karatsuba(string number1, string number2)
{
    if ( (number1.length() < 2) || (number2.length() < 2) )
    {
        return to_string(stoi(number1) * stoi(number2));
    }

    // Lasketaan lukujen koko
    auto m = max(ceil(number1.length() / 2), ceil(number2.length()
/ 2));

    auto low1 = number1.substr(number1.length()-m, m);
    auto high1 = number1.substr(0, number1.length()-m);

    auto low2 = number2.substr(number2.length()-m, m);
    auto high2 = number2.substr(0, number2.length()-m);

    string c0 = multiply_karatsuba(low1,low2);
    string c1 = multiply_karatsuba( sum(low1,high1),
sum(low2,high2));
    string c2 = multiply_karatsuba(high1,high2);

    string appendFirst(m*2, '0');
    string appendSecond(m, '0');

    auto c = findDiff(c1, c2);
    c = c.erase(0, c.find_first_not_of('0'));

    c = findDiff(c, c0);
    c = c.erase(0, c.find_first_not_of('0'));

    auto first = c2.append(appendFirst);
    auto second = c.append(appendSecond);

    return sum(first, sum(second,c0));
}
```