NYILATKOZAT

Név: Csóti Kristóf

ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: EQCO7V

Szakdolgozat címe:
Számelméleti függvények hatékony implementációja

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023.06.05.

a hallgató aláírása

Eötvös Loránd Tudományegyetem

Természettudományi Kar

Algebra és Számelmélet Tanszék

Számelméleti függvények hatékony implementációja

Szerző:
Csóti Kristóf
Alkalmazott Matematikus BSc

Belső témavezető:
Dr. Gyarmati Katalin
egyetemi docens

Külső témavezető:
Seres István András
PhD hallgató

Budapest, 2023

Tartalomjegyzék

1. Bevezetés **3**

1.1. Motiváció . . . 3

1.2. Alapfogalmak, definíciók, jelölések . . . 4

1.3. Kezdetleges algoritmusok . . . 4

2. Szorzási technikák **6**

2.1. Karatsuba algoritmusa . . . 6

2.2. Toom–Cook algoritmus . . . 7

2.3. Diszkrét Fourier-transzformált . . . 8

2.3.1. Alapgondolat . . . 8

2.3.2. A Fourier-transzformált . . . 9

2.3.3. A gyors Fourier-transzformálás . . . 10

2.4. Schönhage – Strassen algoritmus . . . 13

2.4.1. Fourier-transzformáció gyűrűben . . . 13

2.4.2. A Schönhage – Strassen algoritmus . . . 14

2.5. Összegzés . . . 16

3. Moduláris számítások **19**

3.1. Barrett algoritmusa . . . 19

3.2. Euklideszi algoritmus . . . 20

3.3. Moduláris inverz . . . 22

3.4. Moduláris hatványozás . . . 23

3.4.1. A kitevő csökkentése . . . 23

3.4.2. Bináris hatványozás . . . 24

3.4.3. Nagy kitevővel hatványozás . . . 24

3.5. Kínai maradéktétel . . . 25

3.5.1. A maradékrendszer előállítása . . . 26

3.5.2. Maradékrendszer dekódolása . . . 27

4. Implementáció kriptográfiai elemzése **29**

5. Összegzés **31**

Köszönyetnyilvánítás **32**

Irodalomjegyzék **32**

Algoritmusjegyzék **35**

1. fejezet

Bevezetés

1.1. Motiváció

Az algoritmuselmélet egy komoly kérdése, hogyan lehet leghatékonyabban aritmetikai számításokat végezni. Ez különösképpen fontossá vált az egyre fejlettebb és bonyolultabb digitális felhasználások miatt.

Vizsgáljuk csak az elmúlt években berobbant decentralizált pénzügyi rendszereket (röviden DeFi). A DeFi megfelelő működéséhez elengedhetetlenek a komplex, nagy számokkal számoló algoritmusok, számelméleti függvények, hiszen a biztonságot egyedül ezen technikai megoldások biztosítják.

Egyes kriptográfiai titkosítási eljárások (pl. RSA-titkosítás) alapját a moduláris hatványozás képezi, mivel ez egy nehezen visszafejthető függvény. A megfelelő biztonság elérése érdekében 2048 bitesnél hosszabb számokat használnak, azonban a jövőben valószínűleg 4096 bites vagy hosszabb számokat fognak alkalmazni. Ilyen nagy kitevő esetén ez a számítás rendkívül költséges, ezért a hatékonyság maximalizálása elengedhetetlen. Emellett egy másik erős motiváló tényező is van.

Tekintsük a második legnagyobb forgalmú kriptovalutát, az Ethereum-ot működtető rendszert, az ún. Ethereum Virtual Machine-t. A valutához szükséges folyamatokat, tranzakciókat egy "Solidity" nevű nyelven írt "smart contract"-ok működtetik. Az ilyenek során elvégzett számítások költségeit egy bizonyos "gas" egységben mérik. Ennek a mennyisége reprezentálja az egyes műveletekbe fektetett energiát. Egy teljes tranzakció során elégetett "gas" mennyiség árát meg kell fizetni. Ez is motiválja azt, hogy a számítások minél hatékonyabbak legyenek.

A dolgozat során nem vizsgáljuk a "gas" költséget, hanem futásidő alapján osz-

tályozzuk az algoritmusokat, de természetesen ez a két jellemző erősen összefügg egymással.

A szakdolgozat megírása során főképp a [1] és [2] könyvekre támaszkodtam, de ezen kívül az eredeti cikkeket is használtam, amelyeket az irodalomjegyzékben adtam meg. Emellett a további felhasznált könyvek, cikkek is szerepelnek az irodalomjegyzékben. A pszeudokódok a [1] könyvből valóak.

1.2. Alapfogalmak, definíciók, jelölések

1. Definíció. Egy algoritmus lépésszáma a következő $\mathbb{N} \longrightarrow \mathbb{N}$ függvény: $t_A(n) := \max_{x:\,|x|=n}$ (a lépések száma az x input esetén)

A dolgozat során minden input és output is mindig egész szám: $x \in \mathbb{N}, \mid x \mid = n$, az x szám kettes számrendszerbeli alakjában a bitek száma. A kettes számrendszerbeli alak: $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, tehát x_i jobbról az $(i+1)$. bit.

2. Definíció. Legyen $f, g\colon \mathbb{N} \longrightarrow \mathbb{R}_0^+$ függvények. Ekkor azt mondjuk, hogy $f = O(g)$, ha $\exists n_0, c \in \mathbb{N}$, hogy $\forall n \geq n_0$ egészre $f(n) \leq c \cdot g(n)$.

Az imént definiált "ordó"-val fogjuk jellemezni az algoritmusokat az alapján, hogy hány lépést tesznek. A bitműveleteket egy lépésnek számoljuk.

1.3. Kezdetleges algoritmusok

Első lépésként vizsgáljuk meg a kezdetleges algoritmusokat.Ezek az írásban való műveletvégzéssel teljesen egyeznek.
Például az írásbeli összeadás pszeudo-kódja:

1. algoritmus Írásbeli összeadás

Input: $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$, $m \in \{0, 1\}$ (maradék)
Output: $S = \sum_{i=0}^{n-1} s_i \cdot 2^i$, $M \in \{0, 1\}$ (maradék), hogy $x + y + m = S + M \cdot 2^n$

1: $M \leftarrow m$
2: **for** $i = 0..n-1$ **do**
3: $\quad t \leftarrow x_i + y_i + M$
4: $\quad (M, s_i) = (t\ div\ 2, t\ mod\ 2)$
5: **end for**
6: **return** S, M

A 2-vel való egész-osztást (*div*) és 2-es maradék keresését (*mod*) itt egy ismert műveletként felhasználtuk, de mivel a t változó végig legfeljebb 3, ezért ezek egyszerű számítások.

Látható, hogy az algoritmus $O(n)$ lépést tesz n-bites számok esetén. Ez nem is javítható, azonban a konstans szorzó csökkentése hasznos lehet, hiszen a későbbiekben látjuk majd, hogy a szorzás összeadásokra lesz visszavezetve. A kivonás hasonlóan működik.

A szorzás során két n-bites szám szorzata legfeljebb $2n$ bites lesz.

2. algoritmus Írásbeli szorzás

Input: $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, $y = \sum_{i=0}^{n-1} y_i \cdot 2^i$
Output: $M = \sum_{i=0}^{2n-1} m_i \cdot 2^i$, hogy $x \cdot y = M$

1: $M \leftarrow x \cdot y_0$ ▷ vagyis x, ha $y_0 = 1$, egyébként 0
2: **for** $i = 1..n-1$ **do**
3: $\quad M \leftarrow M + 2^i(x \cdot y_i)$ ▷ a 2^i-vel való szorzás egy egyszerű bit shiftelés
4: **end for**
5: **return** M

A szorzás algoritmus tehát n-szer fog két darab maximum $2n$-bites számot összeadni, így a futásidő $O(n^2)$.
Az osztás szintén $O(n^2)$ futásidejű a hagyományos algoritmussal, hiszen mivel 2-es számrendszerbeli alakkal dolgozunk, ezért minden lépésben egy összehasonlítás kell, majd esetlegesen egy kivonás, ami így adja az $O(n^2)$ komplexitást.

2. fejezet

Szorzási technikák

Az 1. fejezetben láttuk az alapműveletek legegyszerűbb kiszámítási algoritmusait.
Az összeadáson illetve kivonáson lényegesen már nem lehet javítani.
Ebben a fejezetben a fejlettebb szorzási algoritmusokat fogjuk vizsgálni.

2.1. Karatsuba algoritmusa

Ebben a részben Karatsuba algoritmusát ismertetjük. [3]

A módszer az 'oszd meg és uralkodj' elven működik. Tegyük fel, hogy mindkét szám legfeljebb n-bites, ahol $n = 2m$ páros. Tehát pl. az a szám előáll $a = 2^m a_0 + a_1$ alakban, ahol a_0 és a_1 legfeljebb m-bites számok. Hasonlóan $b = 2^m b_0 + b_1$. Ekkor $a \cdot b = 2^{2m} a_1 b_1 + 2^m (a_1 b_0 + a_0 b_1) + a_0 b_0$.
Ezzel az átírással látható, hogy egy $2m$-bites szorzás kiváltható négy darab m-bites szorzással. Az algoritmus alapját képző legfontosabb észrevétel az, hogy ez levihető háromra is.

Mivel $a_1 b_0 + a_0 b_1 = (a_1 - a_0)(b_0 - b_1) + a_0 b_0 + a_1 b_1$, ezért $a \cdot b = (2^{2m} + 2^m) a_1 b_1 + (2^m + 1) a_0 b_0 + 2^m (a_1 - a_0)(b_0 - b_1)$.
Így már csak három rekurzívan végzendő szorzás szerepel, illetve néhány összeadás és 2-hatvánnyal szorzás, ami mind $O(m)$ lépés, tehát ha két n-bites szám összeszorzása során elvégzett műveletet $T(n)$-nel jelöljük, akkor $T(2m) \leq 3T(m) + cm$, ahol c valami alkalmas konstans.

Mivel n hosszú számok szorzása esetén a következő rekurzív hívás során az elvégzendő szorzásban a tényezők már csak $\lceil \frac{n}{2} \rceil$-bitesek, ezért hívásonként kb. feleződik a

számok mérete. Ha egy n-bites számból indulunk, akkor $\lceil \log n \rceil$ hívás kell összesen. Ha $k = \lceil \log n \rceil$, akkor $T(n) \leq T(2^k)$.

Állítás:

$$T(2^k) \leq 3^k + c(3^k - 2^k) \tag{2.1}$$

Bizonyítás: Indukcióval bizonyítjuk: $k = 0$-ra az állítás nyilvánvalóan igaz (két bit összeszorzása egy lépés).
Ezután $T(2^k) \leq 3T(2^{k-1}) + c2^{k-1}$ egyenlőtlenségbe beírva az indukciót (azaz az állítást $k-1$ esetén), vagyis $T(2^{k-1}) \leq 3^{k-1} + c(3^{k-1} - 2^{k-1})$-t kapjuk, hogy $T(2^k) \leq 3^k + c(3^k - 3 \cdot 2^{k-1}) + c2^{k-1}$, amiből c kiemelésével adódik az állítás k-ra is.

Tehát $T(n) \leq T(2^k) \leq (c+1)3^k < (c+1)3^{1+\log n} = 3(c+1)n^{\log 3}$, ami miatt a Karatsuba-féle algoritmus futásideje körülbelül $O(n^{1,59})$.

2.2. Toom–Cook algoritmus

A Karatsuba-algoritmus alapgondolatának egy további általánosítása a Toom–Cook algoritmus. [4] [5]
Az algoritmus "rendjét" jelöljük r-rel.

Az összeszorzandó (legfeljebb n-bites) számokat írjuk fel $a_0+a_1x+...+a_{r-1}x^{r-1}$ és $b_0+b_1x+...+b_{r-1}x^{r-1}$ alakban, ahol $x = 2^k$ és így $r = \lceil \frac{n}{k} \rceil$. Karatsuba algoritmusánál $r = 2$, hiszen $k = \frac{n}{2}$ volt a választásunk.
Ekkor az $a \cdot b$ szorzat felírható egy legfeljebb $2r-2$-fokú polinomként. Ezt $2r-1$ pontban interpolálva megkapjuk a szorzat-polinomot, amivel kiszámítható a szorzat.

Az r-rendű Toom–Cook algoritmus során az interpoláció után egy n-bites szorzást sikerült $2r-1$ darab körülbelül $\frac{n}{r}$-bites szorzásra cserélni. Így A felírható a következő rekurzió ($T(n)$ az n-bites szorzás lépésszáma): $T(n) = (2r-1)T(\frac{n}{r})$.
$T(n)$-t keressük $T(n) = n^c$ alakban. Ekkor a rekurzió: $n^c = (2r-1)(\frac{n}{r})^c$, ebből logaritmus véve, és c-re rendezve kapjuk, hogy $c = \frac{\log 2r-1}{\log r}$.

Természetesen az O-ban levő konstans nagyban függ az interpoláció hatékonyságától. Optimális r választás mellett az algoritmus lépésszáma $n^{1+O(\frac{1}{\log n})}$. Ennél a fejlettebb algoritmusok jobb lépésszámot adnak, azonban ez az algoritmus jól párhuzamosítható, ami tovább csökkenti a futásidőt.

A 3-rendű Toom–Cook algoritmus (az interpolációs pontok $-2, -1, 0, 1, 2$, az interpoláló algoritmust ismert szubrutinként hívjuk Interpolate$((x_0 : f(x_0)), ...)$ módon):

3. algoritmus Toom–Cook-3

Input: $a, b \leq 2^n$ egészek

Output: $ab = c_0 + c_1 2^k + c_2 2^{2k} + c_3 2^{3k} + c_4 2^{4k} = C(2^k)$, ahol $k = \lceil \frac{n}{3} \rceil$

1: **if** $n \leq 3$ **then** Karatsuba(a, b)
2: **end if**
3: $a = a_0 + a_1 x + a_2 x^2$, $b = b_0 + b_1 x + b_2 x^2$, ahol $x = 2^k$
4: $v_0 \leftarrow$ Toom–Cook-3(a_0, b_0)
5: $v_1 \leftarrow$ Toom–Cook-3$(a_0 + a_1 + a_2, b_0 + b_1 + b_2)$
6: $v_{-1} \leftarrow$ Toom–Cook-3$(a_0 - a_1 + a_2, b_0 - b_1 + b_2)$
7: $v_2 \leftarrow$ Toom–Cook-3$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
8: $v_{-2} \leftarrow$ Toom–Cook-3$(a_0 - 2a_1 + 4a_2, b_0 - 2b_1 + 4b_2)$
9: $c_0, c_1, c_2, c_3, c_4 \leftarrow$ Interpolate$((0 : v_0), (1 : v_1), (-1 : v_{-1}), (2 : v_2), (-2 : v_{-2}))$
10: **return** $C(x)$

2.3. Diszkrét Fourier-transzformált

2.3.1. Alapgondolat

Ismét induljunk onnan, hogy a és b legfeljebb n-bites számokat írjuk fel polinomként: $P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$, $Q(x) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1}$. Ekkor $a \cdot b$ szorzat kiszámításához meg kell határoznunk az $R(x) = c_0 + c_1 x + c_2 x^2 + ... + c_{2n-2} x^{2n-2}$ polinom együtthatóit.
$R(x)$ együtthatói kiszámíthatóak $P(x)$ és $Q(x)$ együtthatói alapján a következő módon: $c_i = a_0 b_i + a_1 b_{i-1} + ... + a_i b_0$. Ha így akarnánk meghatározni az $R(x)$ együtthatóit, n^2 darab szorzást kéne elvégeznünk.

Jobb módszert kapunk, ha a $P(x)$ és $Q(x)$ polinomokba behelyettesítünk ($x = 0, 1, ..., 2n - 2$ értékeket), majd kiszámoljuk $R(x) = P(x)Q(x)$ szorzatot a megfelelő helyettesítési értékkel, így kapunk egy lineáris egyenletrendszert $R(x)$ együtthatóira:

$$
\begin{aligned}
c_0 &= R(0) \\
c_0 + c_1 + \ldots + c_{2n-2} &= R(1) \\
c_0 + 2c_1 + \ldots + 2^{2n-2} c_{2n-2} &= R(2) \\
&\ldots \\
c_0 + (2n-2)c_1 + \ldots + (2n-2)^{2n-2} c_{2n-2} &= R(2n-2)
\end{aligned}
$$

Mivel itt a változók mindig valamilyen konstans számmal vannak megszorozva, így ez valóban hatékonyabb megoldás, hiszen a konstanssal való szorzás már lineáris idejű. Így egyedül a $P(x)Q(x)$ szorzatok kiszámítása nem lineáris idejű, azonban ezekből csak $2n-1$ darab van. Emellett lineáris idejű szorzást végzünk kb. $4n^2$-szer ($P(x)$ és $Q(x)$ kiértékelése $2n-2$ helyen), majd nagyjából ugyanennyi összeadást, illetve az egyenletrendszert megoldani is $O(n^3)$ lépésben tudjuk például Gauss-eliminációval.

2.3.2. A Fourier-transzformált

A legfőbb észrevétel az, hogy akármilyen komplex számot is használhatunk behelyettesítési értékként. A leghatékonyabb módszert a komplex egységgyökök behelyettesítése fogja adni.

3. Definíció. Az $\varepsilon \in \mathbb{C}$ számot n-edik komplex egységgyöknek nevezzük, ha $\varepsilon^n = 1$.

Az n-edik egységgyökök száma pontosan n.

4. Definíció. Egy komplex szám különböző egész kitevős hatványainak számát a szám rendjének nevezzük.

5. Definíció. Az n rendű komplex számokat primitív n-edik egységgyököknek nevezzük.

Egy komplex szám pontosan akkor primitív n-edik egységgyök, ha hatványai pontosan az n-edik egységgyökök (vagyis az $\varepsilon^n = 1$ összes megoldása).
Tegyük fel, hogy $n = 2^k$ egy 2-hatvány. A $P(x)$ polinomba helyettesítsük be egy primitív n-edik egységgyök, azaz ε hatványait.

$$\hat{a}_j = P(\varepsilon^j) = a_0 + a_1\varepsilon^j + a_2\varepsilon^{2j} + \ldots + a_{n-1}\varepsilon^{(n-1)j}, \tag{2.2}$$

ahol $j = 0, 1, ..., n-1$.
Ekkor az $(a_0, a_1, ..., a_{n-1})$ sorozat n-edrendű diszkrét Fourier-transzformáltja az $(\hat{a}_0, \hat{a}_1, ..., \hat{a}_{n-1})$ komplex számsorozat.

A visszatranszformáláshoz használjuk ki, hogy az összes n-edik egységgyök összege pontosan 0, ha $n > 1$. Emiatt ha minden $j = 0, 1, ..., n-1$-re összeadjuk a az $\hat{a}_j$-ket definiáló összegek ε^{-jk}-szorosát, vagyis:

$$\hat{a}_j \varepsilon^{-kj} = a_0 \varepsilon^{-kj} + a_1 \varepsilon^{(1-k)j} + ... + a_k + ... + a_{n-1} \varepsilon^{(n-1-k)j}, \tag{2.3}$$

tehát egyedül pont az a_k együtthatók lesznek 1-gyel szorozva, az összes többi k' esetén pedig minden j-re egy különböző n-edik egységgyökkel van szorozva az $a_{k'}$ együttható. Emiatt összeadva ezt az $n-1$ egyenletet, azt kapjuk, hogy

$$na_k = \hat{a}_0 + \hat{a}_1 \varepsilon^{-k} + ... + \hat{a}_{n-1} \varepsilon^{-(n-1)k}, \tag{2.4}$$

tehát a visszatranszformálás nagyon hasonlít az eredeti transzformáláshoz.

2.3.3. A gyors Fourier-transzformálás

Innentől feltesszük, hogy $P(x)$ és $Q(x)$ legfeljebb $\frac{n-1}{2}$ fokú, így garantáljuk, hogy $R(x)$ legfeljebb $n-1$-fokú. Ahogy eddig is, most is legyen a $Q(x)$ polinom együtthatói $(b_0, b_1, ..., b_{n-1})$ (az $\frac{n-1}{2}$ feletti együtthatók mind 0-k), és az $R(x) = P(x)Q(x)$ polinom együtthatói, pedig $(c_0, c_1, ..., c_{n-1})$. Ezen számsorozatok n-edrendű diszkrét Fourier-transzformáltjai legyenek $(\hat{b}_0, \hat{b}_1, ..., \hat{b}_{n-1})$ és $(\hat{c}_0, \hat{c}_1, ..., \hat{c}_{n-1})$.
Mivel a Fourier-transzformálás egy polinomba behelyettesítés, és $P(x)Q(x) = R(x)$, ezért

$$\hat{a}_j \hat{b}_j = \hat{c}_j. \tag{2.5}$$

A diszkrét Fourier-transzformálás alkalmazását az indokolja, hogy gyorsan kiszámítható a következő algoritmussal (FFT: gyors Fourier-transzformálás):

4. algoritmus FFT

Input: k, $A[0, 1, ..., 2^k - 1]$

Output: $Y[0, 1, ..., 2^k - 1]$, az A számsorozat Fourier-transzformáltja

1: **if** $k = 0$ **then** return $A[0]$ ▷ leállási feltétel
2: **end if**
3: $\varepsilon := e^{\frac{2\pi i}{2^k}}$ ▷ primitív 2^k-adik egységgyök
4: $E := 1$ ▷ Ez mindig ε^j lesz, j 0-tól indul, ezért először 1-re állítjuk
5: $A_0 := (A(0), A(2), ..., A(2^k - 2))$ ▷ A páros indexű elemek
6: $A_1 := (A(1), A(3), ..., A(2^k - 1))$ ▷ A páratlan indexű elemek
7: Y_0 :=FFT($k - 1$,A_0) ▷ Rekurzív hívás
8: Y_1 :=FFT($k - 1$,A_1) ▷ Rekurzív hívás
9: **for** $j = 1, ..., 2^k - 1$ **do**
10: $s := Y_1(j)E$
11: $Y(j) := Y_0(j) + s$
12: $Y(j + 2^{k-1}) := Y_0(j) - s$
13: $E := \varepsilon \cdot E$ ▷ $E = \varepsilon^j$ biztosítása
14: **end for**
15: **return** Y

Állítás: Az algoritmus valóban az A számsorozat Fourier-transzformáltját adja vissza.

Bizonyítás: A leállási feltétel triviálisan jól számolja ki, ezért elegendő a 11. és 12. sor helyességét látni. A 11. sor bizonyítása egy lépéssel egyszerűbb, ezért a 12. sort bizonyítjuk.

Az Y_0 2^{k-1}-rendű Fourier-transzformált j-edik értéke a következőképp számítható ki:

$$Y_0(j) = \sum_{i=0}^{2^{k-1}-1} a_{2i}(\varepsilon^2)^{ij}, \tag{2.6}$$

hiszen most az $\varepsilon' = e^{\frac{2\pi i}{2^{k-1}}} = \varepsilon^2$ számot helyettesítjük be. Az Y_1 Fourier-transzformált hasonlóan számítható a_{2i+1} indexeléssel.

Az A számsorozat Y 2^k-rendű Fourier-transzformáltjának $j + 2^k$-adik tagja (egy kicsit másfajta indexeléssel):

$$Y(j) = \sum_{i=0}^{2^{k-1}-1} a_{2i}\varepsilon^{2i(j+2^{k-1})} + a_{2i+1}\varepsilon^{(2i+1)(j+2^{k-1})} \tag{2.7}$$

Ezt kicsit átírva kapjuk, hogy

$$Y(j) = \sum_{i=0}^{2^{k-1}-1} a_{2i}(\varepsilon^2)^{ij}\varepsilon^{i2^k} + \sum_{i=0}^{2^{k-1}-1} a_{2i+1}(\varepsilon^2)^{ij}\varepsilon^{i2^k}\varepsilon^j\varepsilon^{2^{k-1}} \tag{2.8}$$

Mivel ε egy 2^k-adik egységgyök, ezért $\varepsilon^{2^k} = 1$, és $\varepsilon^{2^{k-1}} = -1$. Ekkor láthatjuk, hogy pont a (2.6.) egyenlet jobb oldala lesz az első szummából, illetve az $Y_1(j)$ $-\varepsilon^j$-szerese lesz a másodikból. Ezzel megkaptuk a bizonyítandó

$$Y(j) = Y_0(j) - \varepsilon^j Y_1(j) \tag{2.9}$$

kifejezést.

A 11. sor ugyanígy igaz, csak ott nem kell a $+2^{k-1}$-gyel ügyeskedni.

Az aritmetikai műveletek darabszámát jelölje $T(k)$. Ekkor a rekurzív hívásokkor $2T(k-1)$ művelet fut le, emellett a ciklusban pedig 2^k művelet szerepel, így $T(k) = 2T(k-1) + 2^k$, amiből $T(k) = (k+1)2^k = n(\log n + 1)$. Ez a módszer használható többek közt polinomok összeszorzására is.

Az iménti számolás az aritmetikai műveleteket egy lépésnek vette, ami nem teljesen pontos. A következőkben egy bitműveletet fogunk egy lépésnek tekinteni.

Legyen a két összeszorzandó (legfeljebb n-bites) szám 2-es számrendszerbeli alakja $a = \overline{a_{n-1}...a_1a_0}$ és $b = \overline{b_{n-1}...b_1b_0}$. Most is polinomokkal végezzük el a szorzást először. A számjegyekből képzett polinomok legyenek $A(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1}$ és $B(x)$ hasonlóan. Így nekünk a $C(x) = A(x)B(x)$ polinom $C(2)$ helyettesítési értékét kell kiszámolnunk.

Mivel $A(x)$ és $B(x)$ minden együtthatója 0 vagy 1, ezért a $C(x)$ polinom összes együtthatója legfeljebb n lehet, ezért a behelyettesítés – ami néhány 2-vel hatványozás, majd legfeljebb $n \log n$-bites számok összeadása – futásideje $O(n \log n)$.

A Fourier-transzformáció során komplex, irracionális számokkal számoltunk, ezért ezzel foglalkozni kell még. A komplex számokkal egyszerű műveleteket végezni, hiszen két valós számmal lehet reprezentálni, és a komplex műveletvégzés is megoldható néhány valós művelettel.

A valós számok pontosságát és hibáját kell még megbecsülni. Legyen $2n-2 = 2^k$, vagyis a $C(x)$ polinom együtthatóinak száma legyen 2-hatvány.

Állítás: Ha a 2^k-adik egységgyök $8 \log n$ bit pontos, akkor a számítások végén legfeljebb $\log n$ bit kerekítési hiba lesz, de ezt egészre kerekítve pontos értéket kapunk.

Bizonyítás:

(1.) A $8\log n$ bit hiba egy legfeljebb $\frac{1}{n^8}$ nagyságú hibát jelent.

(2.) Ha $|x| \leq 1$ hibája maximum d és $|y| \leq 1$ hibája is maximum d, akkor xy hibája legfeljebb $3d$ lehet.

Ezek alapján a rekurzió hívásakor, amikor ε^2 értékét számoljuk, akkor a hiba legfeljebb 3-szorozódik. Így az FFT$(k-l, A_{...})$ hívásakor a hiba legfeljebb $\frac{3^l}{n^8}$. Az E érték kiszámításakor a j. ciklusig legfeljebb $2j$-szeresére nőtt a hiba (indukcióval látható).

Ezeket összerakva a hiba becslése: $\frac{2j3^l}{n^8} \leq \frac{2^{k-l}3^l}{n^8} \leq \frac{4^k}{n^8} \leq \frac{1}{n^6}$.

Ezt egy $O(n)$ nagyságrendű együtthatóval szorozva, majd $O(n)$ így kapott számot összeadva a hiba legfeljebb $O(\frac{1}{n^4})$ lesz, tehát $A(\varepsilon^j)$ hibája ilyen nagyságrendű lesz. $|A(\varepsilon^j)| \leq n$, hiszen $A(x)$-nek n darab 0 vagy 1 értékű együtthatója van, emiatt pedig az $A(\varepsilon^j)B(\varepsilon^j)$ szorzat hibája $O(\frac{1}{n^3})$ nagyságrendű lesz. A visszatranszformáláskor a hiba $O(\frac{1}{n})$ lesz, tehát a számítások végén ekkora nagyságrendű hiba lesz, amit egészre kerekítve a megfelelő értéket kapjuk.

Ezzel láttuk, hogy $O(\log n)$ bit pontos számábrázolással megfelelően tudunk számolni. Ennyi bites számok között az aritmetikai műveletek $O(\log^2 n)$ időben végezhetők, tehát összesen a futásidő $O(n \log^3 n)$.

2.4. Schönhage – Strassen algoritmus

2.4.1. Fourier-transzformáció gyűrűben

Most a Fourier-transzformálást egy R gyűrűben fogjuk elvégezni, ahol ω egy K-adik egységgyök, ha $\omega^K = 1$ R-ben. Ekkor az $(a_0, a_1, ..., a_{K-1})$ sorozat K-rendű Fourier-transzformáltja az $(\hat{a}_0, \hat{a}_1, ..., \hat{a}_{K-1})$ sorozat, ahol

$$\hat{a}_j = a_0 + a_1\omega^j + a_2\omega^{2j} + ... + a_{K-1}\omega^{(K-1)j}. \tag{2.10}$$

Az eddig tárgyalt Fourier-transzformálthoz ez nagyon hasonlít, az ezt kiszámító (oda- és visszatranszformáló) algoritmusok is ugyanazon az alapon működnek, vagyis azon, hogy kettészedjük az $(a_0, ...a_{K-1})$ számsorozatot páros és páratlan indexű részsorozatokra, és rekurzívan hívjuk az algortimust.

1. Tétel. *Egy R gyűrűben az $(a_0, a_1, ..., a_{K-1})$ sorozat K-rendű Fourier-transzformáltat $O(K \log K)$ lépésben helyesen ki lehet számolni.*

Ezt nem bizonyítjuk, sok hasonló gondolat szerepel, mint az előző algoritmus bizonyításánál.

2.4.2. A Schönhage – Strassen algoritmus

A következő részben a Schönhage–Strassen algoritmussal foglalkozunk. [6]
Az algoritmus két n-bites szám (a és b) szorzatát számolja ki modulo 2^n+1. Legyen R_n gyűrű a modulo 2^n+1 maradékosztályok gyűrűje. Az algoritmus során a Fourier-transzformációt $R_{n'}$ gyűrűn fogjuk végezni. Az input két egész szám, a és b, amelyekre $a, b < 2^n + 1$, és egy $K = 2^k$ szám, amire $n = MK$ valamilyen M-re. A működésének alapja a következő lépések:

- Felbontjuk a számokat úgy, hogy $A = \sum_{j=0}^{K-1} a_j 2^{jM}$, ahol minden $0 \leq a_j < 2^M$, kivéve $0 \leq a_{K-1} \leq 2^M$. B-t felbontjuk ugyanígy.

- Ezután választunk egy olyan n'-t, ami többszöröse K-nak, és $n' \geq \frac{2n}{K} + k$. Ekkor legyen $\theta = 2^{\frac{n'}{K}}$ és $\omega = \theta^2$.

- Most minden $j = 0, ..., K-1$-re a_j-t és b_j-t átszámoljuk $\theta^j a_j$-re és $\theta^j b_j$ modulo $(2^{n'} + 1)$. Nyilván $\theta^K = -1$ modulo $(2^{n'} + 1)$, és így ω egy primitív K-adik egységgyök lesz $R_{n'}$-ben.

- Az így kapott $(a_0, ..., a_{K-1})$ és $(b_0, ..., b_{K-1})$ sorozatokat (előre) Fourier-transzformáljuk K renddel és ω egységgyökkel (így kapjuk a kalapos $\hat{a}$ és $\hat{b}$ értékeket), és kiszámítjuk a
  $\hat{c}_j = \hat{a}_j \hat{b}_j$ modulo $(2^{n'} + 1)$
  szorzatokat. Itt vagy rekurzívan hívjuk az algoritmust a szorzás elvégzésére, vagy ha n' már kicsi, akkor egy másik szorzási algoritmust, például a Karatsuba-algoritmust. A $\hat{c}_0, ..., \hat{c}_{K-1}$ sorozatot visszafelé transzformálva kapjuk a $c_0, ..., c_{K-1}$ sorozatot.

- Végül a c_j értékeket végigosztjuk $K\theta^j$-vel modulo $(2^{n'}+1)$. Legvégül, ha $c_j \geq (j+1)2^{2M}$, akkor c_j-t csökkentjük $(2^{n'} - 1)$-gyel, hogy (2.12) teljesüljön. Így megkapjuk az $AB = C = \sum_{j=0}^{K-1} c_j 2^{jM}$ szorzatot modulo $(2^n + 1)$, ami az algoritmus output-ja.

6. Definíció. Azt mondjuk, hogy $f(n)$ függvény $g(n)$ nagyságrendű, ha $f(n) = O(g(n))$ és $g(n) = O(f(n))$.

2. Tétel. *Az algoritmus kiszámolja a* $0 \leq A, B < 2^n + 1$ *számok szorzatát modulo* $(2^n + 1)$, *és futásideje* $O(n \log n \log\log n)$, *ha* K $\sqrt{n}$ *nagyságrendű.*

Bizonyítás. A helyesség bizonyítása n szerinti indukcióval megy, a kezdőlépés egyszerű, hiszen ha n' már kicsi, akkor egy másik szorzási technikát alkalmazunk, aminek már láttuk a helyességét.

Az indukciós lépés bizonyításához fejezzük ki $AB = C = \sum_{j=0}^{K-1} c_j 2^{jM}$ modulo $(2^n + 1)$ együtthatóit: c_j-t azok az $a_l b_m$ szorzatok befolyásolják, ahol $l, m = 0, 1, ..., K-1$ és $l+m = j$ vagy $l+m = j+K$ feltételek teljesülnek. Az $l+m = j+K$ esetben $a_l 2^{lM} b_m^{2mM} = a_l b_m 2^{jM+KM}$, ami kongruens $-a_l b_m 2^{jM}$-mel modulo $(2^n + 1)$, hiszen $KM = n$. Emiatt:

$$c_j = \sum_{l,m=0 \ \& \ l+m=j}^{K-1} a_l b_m - \sum_{l,m=0 \ \& \ l+m=j+K}^{K-1} a_l b_m, \tag{2.11}$$

ezt szeretnénk valahogy kiszámolni.
Az első szumma $j + 1$ tagból áll, a második pedig $K - 1 - j$-ből, és minden tag legfeljebb 2^{2M}, az első szumma tagjaira pedig ez szigorúan kisebb, mint 2^{2M}. Így kapjuk, hogy

$$(j - K + 1)2^{2M} \leq c_j < (j + 1)2^{2M}. \tag{2.12}$$

Legyen $a'_j = \theta^j a_j$. Az előre és hátra Fourier-transzformálás után kapjuk, hogy

$$c'_i = K \cdot \sum_{l,m=0 \ \& \ l+m=i}^{K-1} a'_l b'_m + K \cdot \sum_{l,m=0 \ \& \ l+m=i+K}^{K-1} a'_l b'_m \tag{2.13}$$

Visszaírva az $a'_j = \theta^j a_j$-t, és b-re is ugyanzet azt kapjuk, hogy

$$c'_i = \theta^i K \cdot \sum_{l,m=0 \ \& \ l+m=i}^{K-1} a_l b_m - \theta^i K \cdot \sum_{l,m=0 \ \& \ l+m=i+K}^{K-1} a_l b_m, \tag{2.14}$$

hiszen $\theta^{K+i} = -\theta^i$ modulo $(2^{n'} + 1)$. Végül az algoritmus a c'_i értékeket leosztja $\theta^i K$-val, és korrigál, ha (2.12) sérül. Így pont a (2.11) egyenletben kapott c_j értékeket kapjuk meg, tehát az algoritmus valóban a helyes végeredményt adja vissza.

A tételben feltettük, hogy K $\sqrt{n}$ nagyságrendű, emiatt n' is $\sqrt{n}$ nagyságrendű. A lépések közül a rekurzívan hívott szorzáshoz kell a legtöbb lépés. Az indukciós lépés alkalmazásával kapjuk, hogy ennek a lépésnek a komplexitása $O(Kn' \log n' \log\log n') = O(n \log n \log\log n)$, mivel n' $\sqrt{n}$ nagyságrendű. A többi lépés ennél kevesebb lépést tesz. □

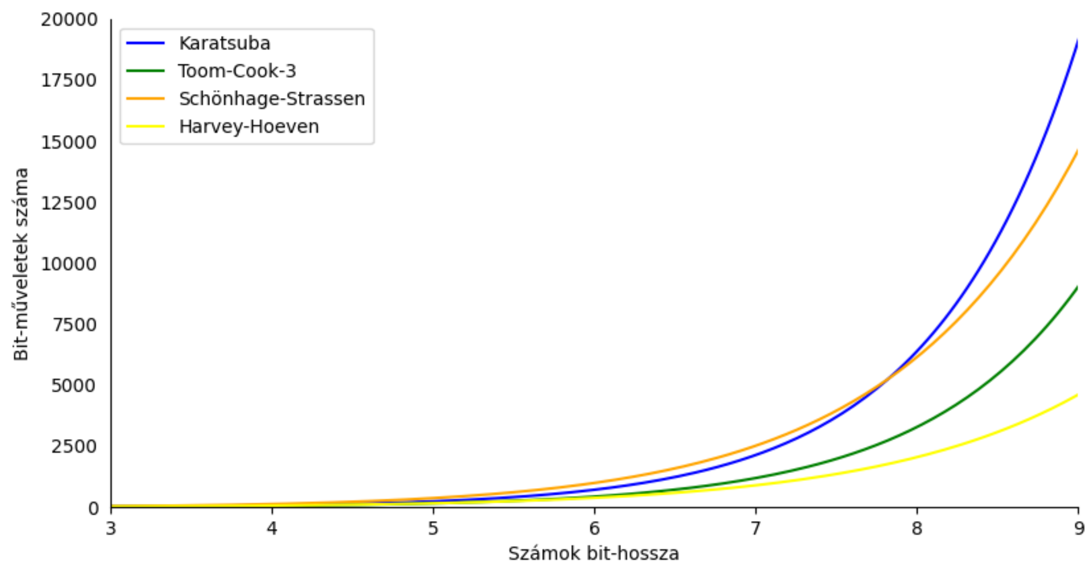
2.5. Összegzés

Ebben a fejezetben láthattunk négy különböző módot arra, hogy két egész szám szorzatát hatékonyan kiszámoljuk. A jelenleg ismert leghatékonyabb módszer David Harvey és Joris van der Hoeven nevéhez fűződik, akik 2019-ben egy $O(n \log n)$ idejű algoritmust publikáltak. [7]

A Schönhage–Strassen algoritmus esetén is láthattuk, hogy ha már kis bit-méretű számokat kell összeszorozni, akkor célszerűbb például Karatsuba algoritmusát használni. Azt, hogy melyik eljárást érdemes használni, a bit-méret határozza meg, illetve az, hogy hogyan vannak implementálva az egyes módszerek. Minden algoritmusnál a lépésszámot csak becsültük az $O()$ jelöléssel, azonban nagyon jelentős, hogy milyen konstansok teljesítik a lépésszámok megfelelő felső becslését (ld. 1. fejezet, Definíció 2).

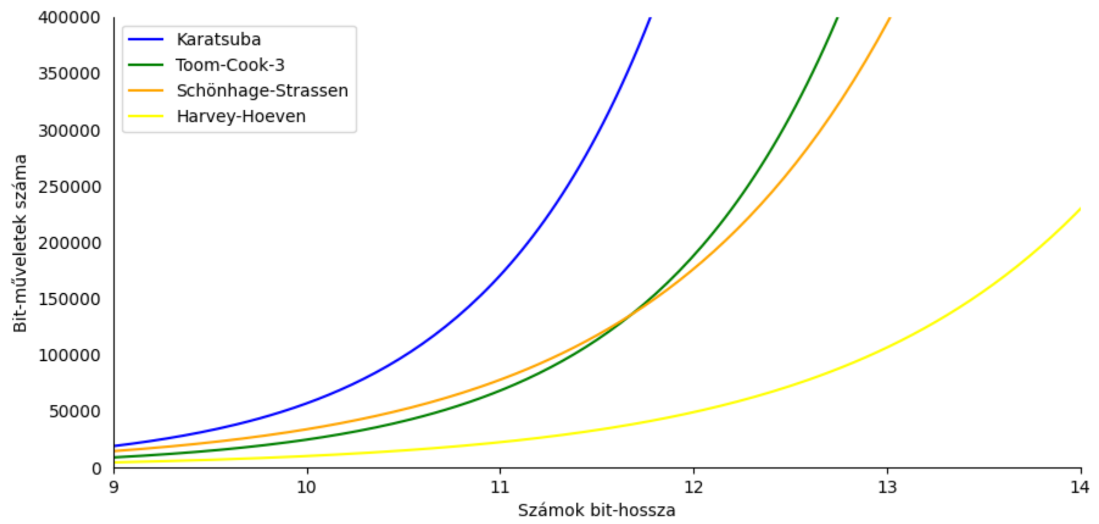
A következő ábrák azt szándékoznak szemléltetni, hogy valóban érdemes meggondolni, hogy mikor melyik algoritmust hatékony használni. Az ábrázolásban pusztán az $O()$ jelölés belsejét használtam, tehát az imént kiemelt konstansokkal nem foglalkozik a grafikon. Így tehát ezek egyáltalán nem pontosak, viszont az látható belőle, hogy kis méretű számok esetén jól használhatók az egyszerűbb algoritmusok, viszont ha a bit-méret már nagy, akkor előbb-utóbb a fejlettebb algoritmusok egyértelműen jobban teljesítenek, és ez a konstansokat beleszámítva is igaz lesz.

Az ábrákon az x tengelyen az összeszorzandó számok bitméretének 2-es alapú logaritmusa szerepel, tehát például az x-tengelyen az $x = 5$ helyen a grafikon a 2^5 helyen felvett függvényértéken halad át. A grafikonokat definiáló függvények tehát az $O()$ jelölésben előforduló függvények, vagyis például a Toom–Cook-3 algoritmus esetén az $f(x) = x^{1.46}$ függvényt ábrázoltam.



Ezen az ábrán látszódik, hogy kis számok esetén (és a konstansok beszámítása nélkül) a Karatsuba-algoritmus és a 3-rendű Toom–Cook algoritmus is jobb futásidőt mutat, mint a Schönhage–Strassen-algoritmus, ami körülbelül 2^8 bit-hossznál éri utol Karatsuba algoritmusát.

A következő ábrán már nagyobb bit-méreteket vizsgálunk:



Most azt láthatjuk, hogy ha a bit-méret 2^{12}-t eléri, akkor már valóban a fejlettebb algoritmusok futásideje lesz alacsonyabb a konstansokat figyelmen kívül hagyva.

3. fejezet

Moduláris számítások

Ebben a fejezetben a moduláris osztásra, hatványozásra és inverz-képzésre látunk algoritmusokat. A számításokat modulo m végezzük, a számítás során az inputok és a végeredmény is egy maradékosztály reprezentása lesz, amire igaz, hogy nemnegatív és m-nél kisebb.

3.1. Barrett algoritmusa

Ebben a részben Barrett algoritmusát ismertetjük. [8]
Az algoritmus egy moduláris osztást elvégző módszer, azonban több megkötés is van az inputokra. Legyen β a használt számábrázolás alapja (például n-bites számok esetén 2^n). Legyen A és B olyan, hogy $0 \leq A < \beta^2$ és $\frac{\beta}{2} < B < \beta$. Az algoritmus ekkor kiszámít két olyan Q és R számot, hogy $A = QB + R$, ahol $0 \leq R < B$, vagyis $Q = A \bmod B$.

Az algoritmus:

5. algoritmus Barrett

Input: A,B az iménti feltételekkel
Output: Q hányados, R maradék

1: $I := \lfloor \frac{\beta^2}{B} \rfloor$ ▷ Előfeldolgozás
2: $A = A_1B + A_0$, ahol $0 \leq A_0 < \beta$ ▷ A felbontása
3: $Q := \lfloor \frac{A_1 I}{B} \rfloor$ ▷ Q becslése
4: $R := A - QB$
5: **while** $R \geq B$ **do**
6: $\quad Q = Q + 1$
7: $\quad R = R - B$
8: **end while**
9: **return** Q, R

3. Tétel. *Az algoritmus helyesen számol, és a ciklust legfeljebb háromszor futtatja le (ez a becslés éles).*

Bizonyítás. Az, hogy $A = QB + R$ a 4. sorból adódik, tehát csak azt kell látni, hogy a végén B valóban egy reprezentáns. Mivel $\frac{\beta}{2} < B < \beta$, ezért $\beta \leq \lfloor \frac{\beta}{B} \rfloor = I < 2\beta$. Mivel $I \leq \frac{\beta^2}{B}$, vagyis $IB \leq \beta^2$, így $Q \leq \frac{A_1 I}{\beta} \leq \frac{A_1 \beta}{B} \leq \frac{A}{B}$, amiből következik, hogy R nemnegatív.
I definíciójából következik, hogy $I > \frac{\beta^2}{B} - 1$, amiből $IB > \beta^2 - B$. Q definíciójából hasonlóan $\beta Q > A_1 I - \beta$.
Induljunk ki $BQ\beta$ -ból. $BQ\beta > A_1 IB - \beta B > A_1(\beta^2 - B) - \beta B$ az előzőek miatt. $A_1\beta^2 - A_1 B - \beta B = \beta(A - A_0) - (A_1 + \beta)B > \beta A - 4\beta B$, hiszen $A_0 < \beta < 2B$, és $A_1 < \beta$. β-val való osztással kapjuk, hogy $BQ > A - 4B$, vagyis $A < QB + 4B$, tehát valóban legfeljebb háromszor kell elvégezni a korrigáló ciklust. □

Az előfeldolgozás nélkül a futás során a legköltségesebb művelet kettő darab szorzás (3. és 4. lépés), így a futásidőt a szorzás elvégzésének gyorsasága határozza meg.

Az algoritmus például akkor alkalmazható, ha kiszámítottuk két szám szorzatát, és redukálni szeretnénk a szorzatot modulo B. Két olyan szám esetén, ami kisebb, mint β, a szorzat kisebb lesz β^2-nél, ezért az algoritmus alkalmazható.

Az elején végzett előfeldolgozás miatt – ami csak a bázistól és B-től függ – az algoritmus akkor igazán hasznos, ha ugyanazzal az osztóval több műveletet is el kell végeznünk.

3.2. Euklideszi algoritmus

Legyen A és B legfeljebb n-bites számok. A kiterjesztett euklideszi algoritmus a két szám legnagyobb közös osztóját keresi meg, és előállítja $(A, B) = uA + vB$ alakban (u, v egészek).

6. algoritmus GCD

Input: A,B
Output: $(A,B),u,v$

1: **if** $A > B$ **then** $A,B = B,A$ ▷ Csere
2: **end if**
3: **lnko**$(A,B,1,0,0,1)$ ▷ Rekurzív algoritmus hívása
4: **lnko**(a,b,u,v,w,z): ▷ $a = uA + vB$ és $b = wA + zB$
5: **if** $A = 0$ **then return** (b,w,z)
6: **end if**
7: $b = ca + r$, ahol $0 \leq r < a$ ▷ $b : a$ maradékos osztás elvégzése
8: **lnko**$(r,a,(w-cu),(z-cv),u,v)$ ▷ rekurzív hívás

Az iménti az úgynevezett kiterjesztett euklideszi algoritmus. A hagyományos euklideszi algoritmus pusztán a legnagyobb közös osztót határozza meg. Az algoritmus nagyon hasonlít a kiterjesztetthez, annyiban különbözik, hogy nem tároljuk az **lnko** algoritmus u,v,w,z paramétereit, így a rekurzív híváskor csak r,a paraméterekkel hívjuk az algoritmust.

4. Tétel. *A kiterjesztett euklideszi algoritmus valóban a legnagyobb közös osztó helyes felbontását számolja ki, és a futás során legfeljebb $2n$ darab maradékos osztást végez, így az algoritmus lépésszáma $O(n^3)$.*

Bizonyítás. A legnagyobb közös osztó a végeredmény:
Az algoritmus során számolt egyenlőségek:

$$\begin{aligned} b &= c_1 a + r_1 \\ a &= c_2 r_1 + r_2 \\ &\dots \\ r_{n-2} &= c_n r_{n-1} + r_n \\ r_{n-1} &= c_{n+1} r_n \end{aligned}$$

Az utolsó egyenletből következik, hogy $r_n | r_{n-1}$, emiatt a felette lévőből az jön, hogy $r_n | r_{n-2}$, és ez így végigvihető $r_n | a$ és $r_n | b$-ig, ezért r_n valóban egy közös osztó. Legyen r' egy másik közös osztó. Mivel $r' | b$ és $r' | a$, így $r' | r_1$. Ugyanígy látható, hogy $r' | r_2$, ..., $r' | r_n$, tehát r_n a közös osztók közül a legnagyobb.

A felbontás helyessége:
A 3. sorban az **lnko** algoritmus úgy van hívva, hogy teljesíti az $a = uA + vB$ és $b = wA + zB$ feltételeket. Így elegendő annyit belátni, hogy $r = (w-cu)A + (z-cv)B$.

$r = b - ca$-ba beírva, hogy $a = uA + vB$ és $b = wA + zB$, épp az igazolandó egyenlőséget kapjuk.

A maradékos osztások száma:
Az 1. sor miatt az elején is garantálva van, hogy $A \leq B$, tehát **lnko** hívása során az a, b paraméterekre végig igaz, hogy $a \leq b$, tehát a maradékos osztás c hányadosa legalább 1. Így $b \geq a + r > 2r$, tehát $r \leq \frac{b}{2}$.
Így a mindenkori ab szorzat legalább a felére csökken minden egyes rekurzív hívás során, vagyis legalább egy bittel rövidebb lesz. A, B legfeljebb n-bitesek, így AB legfeljebb $2n$-bites, ezért az algoritmus maximum $2n$ hívást csinál, tehát legfeljebb $2n$ maradékos osztást végez el. □

Ha elvégezhető a moduláris osztás, akkor az mindenképp visszavezethető egy olyan $a : b$ mod m osztásra, ahol b és m relatív prímek. Ez az osztás elvégezhető úgy, hogy először ellenőrizzük, hogy valóban relatív prím-e b és m, majd a kiterjesztett euklideszi algoritmussal keresünk olyan u, v számokat, hogy $1 = ub + vm$. Ekkor $a : b$ mod m eredménye ua mod m, hiszen ekkor $uab = a$ mod m. Így tehát a kiterjesztett euklideszi algoritmus segítségével a moduláris osztás kiszámítható $O(n^3)$ időben.

3.3. Moduláris inverz

Egy a szám inverze modulo m egy olyan b szám, hogy $ab = 1$ mod m. Ez csak akkor létezik, ha a és m relatív prímek. Ennek kiszámítására alkalmazható a kiegészített euklideszi algoritmus, és az **lnko** algoritmusnak elég csak az egyik felírást eltárolnia.

Még a fejlettebb technikák esetén is az inverz számítása költségesebb a szorzásnál, így ezt érdemes lenne minél kevesebbszer elvégezni. A következő algoritmus erre ad egy megoldást.

7. algoritmus Több inverz

Input: $0 < x_i < N$, ahol $i = 1, ..., k$

Output: $y_i = 1/x_i \bmod m$, ahol $i = 1, ..., k$

```
 1: z_1 := x_1
 2: for i = 2, ..., k do
 3:     z_i := z_{i-1} x_i mod m
 4: end for
 5: q := 1/z_k mod m
 6: for i = k, ..., 2 (-1) do
 7:     y_i = q z_{i-1} mod m
 8:     q = q x_i mod m
 9: end for
10: y_1 = q
11: return y_1, ..., y_k
```

Mivel a második for ciklus előtt $q = 1/(x_1 x_2 ... x_k)$, és $z_i = x_1 x_2 ... x_i$, ezért y_k-nak éppen $1/x_k$-t fogjuk értékül adni, majd $q = 1/x_1 ... x_{k-1}$ fog maradni. Ekkor $i = k-1$-gyel folytatva hasonlóan látható az algoritmus helyessége, és a végén q-ból $1/x_1$ fog maradni.

Így tehát az inverz kiszámítását csak egyszer kell elvégezni, emellett pedig $3(k-1)$ darab modulo m szorzást végzünk.

3.4. Moduláris hatványozás

A kriptográfia egyik legismertebb eljárása, az RSA-titkosítás, amelynek alapját a moduláris hatványozás jelenti. Egy $a^b \bmod m$ hatványt könnyen ki tudunk számolni, azonban az a szám meghatározása a többi ismeretében nagyon nehéz, ha m legalább két nagy prímtényezőből áll, ezek miatt RSA-eljárás alkalmas titkosításra.

3.4.1. A kitevő csökkentése

Ha a kitevő nagy, akkor megpróbálhatjuk redukálni az Euler-féle φ-függvény segítségével.

7. Definíció. Egy n pozitív egész szám esetén $\varphi(n)$ jelenti a az $1, 2, ..., n$ számok közül n-nel relatív prímek számát.

Ez könnyen kiszámítható, hiszen ha n prímtényezős felbontása $n = p_1^{\alpha_1} p_2^{\alpha_2} ... p_k^{\alpha_k}$, akkor $\varphi(n) = \prod_{i=1}^{k}(p_i^{\alpha_i} - p_i^{\alpha_i - 1}) = \frac{1}{n}\prod_{i=1}^{k}(1 - \frac{1}{p_i})$.

5. Tétel. ***(Kis Fermat-tétel)*** *Ha a és m relatív prímek, akkor $a^{\varphi(m)} = 1$ mod m.*

Ezen tétel miatt az a^b mod m hatványozás kitevője redukálható b mod m szerinti maradékára, ha a és m relatív prímek. Ezt akkor célszerű használni, ha ismerjük m prímtényezős felbontását.

3.4.2. Bináris hatványozás

Ez az algoritmus a kitevő bináris alakján halad végig balról jobbra. Egy lépés azon alapszik, hogy $a^{(e_0 ... e_k 1)_2} = a^{(e_0 ... e_k 0)_2} \cdot a = a^{2 \cdot (e_0 ... e_k)_2} \cdot a = (a^{(e_0 ... e_k)_2})^2 \cdot a$, ahol az $(e_0 ... e_k)_2$ jelölés egy kettes számrendszerbeli alakot jelent. Ezzel a módszerrel balról jobbra haladva bitenkénti bontásban el tudjuk végezni a hatványozást.

8. algoritmus Bináris hatványozás

Input: a, b, m pozitív egészek, $b = (e_k ... e_0)_2$ alakú ($e_k = 1$ feltehető)
Output: $x = a^b$ mod m

1: $x := a$
2: **for** $i = k - 1, ..., 0$ (-1) **do**
3: $\quad x = x^2$ mod m
4: $\quad$ **if** $e_i = 1$ **then** $x = ax$ mod m
5: $\quad$ **end if**
6: **end for**
7: **return** x

Ha b kettes számrendszerbeli alakja $k+1$ bitből áll, és az első bit kivételével (ami bizotsan 1-es) a többi $\frac{1}{2}$-$\frac{1}{2}$ valószínűséggel 1-es és 0-s, akkor várhatóan $\frac{3}{2}k$ szorzást végez az algoritmus.

3.4.3. Nagy kitevővel hatványozás

Ha a kitevő elég nagy, akkor egy előfeldolgozás segítségével csökkenthető a szorzások száma. A szorzási technikák esetén többször is előfordult, hogy egy szám bináris alakját k-bites blokkokra bontjuk. Legyenek egy felbontott blokk e_i, ahol $0 \leq e_i < 2^k$. Az előző algoritmus annyiban módosítjuk, hogy most nem egyesével haladunk a biteken, hanem a k hosszú bitblokkokon lépkedünk. Emellett előre kiszámítjuk az összes lehetsége a^{e_i} mod m hatványt, ahol $e_i = 1, ..., 2^k - 1$.

9. algoritmus K-Bináris hatványozás

Input: a, b, m pozitív egészek, $b = \sum_{i=0}^{l} e_i 2^{ik}$
Output: $x = a^b \bmod m$

```
1: pre[i] := a^i mod m, ahol i = 1, ..., 2^k − 1      ▷ előfeldolgozás
2: x := pre[e_l]
3: for i = l − 1, ..., 0 (-1) do
4:     x = x^(2^k) mod m
5:     if e_i > 0 then x = pre[e_i]x mod m
6:     end if
7: end for
8: return x
```

Az előfeldolgozáshoz $2^k - 2$ moduláris szorzás elvégzése szükséges. Ha ismét feltételezzük, hogy a k-bites e_i blokkok egyenletes valószínűséggel kerülnek ki a $0, 1, ..., 2^k - 1$ számok közül (kivéve e_l esetén), akkor $1 - \frac{1}{2^k}$ annak a valószínűsége, hogy az 5. lépésben kell szorzást végeznünk.

A szorzások összeszámolásánál a 4. sorban levő hatványozást kihagyhatjuk, hiszen az nem (nagyon) függ k választásától, mivel a művelet elvégzéséhez k darab négyzetre emelés, vagyis szorzás kell mod m, és ezt a programsort l-szer végezzük, és lk körülbelül n-nel egyenlő, ahol n a kitevő bitjeinek száma.

Így tehát az elvégzett mod m szorzások száma várhatóan $2^k - 2 + \frac{n}{k}(1 - \frac{1}{2^k})$ (ismét használtuk, hogy l nagyjából $\frac{n}{k}$). Ha $n > 16$, akkor már nem $k = 1$ lesz az optimális (ami pont az előző algoritmus) választás, hanem $k = 2$, és $n > 48$ esetén a $k = 3$ az optimális, tehát ez az algoritmus valóban hatékonyabb nagy kitevő esetén.

3.5. Kínai maradéktétel

6. Tétel. ***(Kínai maradéktétel)*** *Ha x ismeretlen, és ismerjük az $x_1, ..., x_k$ maradékait a páronként relatív prím $m_1, ..., m_k$ modulusokra nézve, akkor egyértelműen meghatározható x maradéka az $m_1 \cdot ... \cdot m_k$ szorzatra mint modulusra.*

Bizonyítás. Ha $k = 2$-re belátjuk, akkor utána egyszerűen alkalmazhatunk indukciót, hiszen ha x_1 és x_2 maradékok ismeretében kiszámolunk egy x' maradékot modulo $m_1 \cdot m_2$ (ez a tétel $k = 2$ esete), akkor ez x_3 mod m_3-mal együtt újra a $k = 2$ esetet adja.

A $k = 2$ eset: mivel m_1 és m_2 relatív prímek, ezért a kiterjesztett euklideszi algoritmus kiszámít olyan q_1 és q_2 számokat, amelykre $1 = q_1 m_1 + q_2 m_2$. Ekkor $x = x_1 + (x_2 - x_1) q_1 m_1 \bmod m_1 m_2$ épp a keresett x érték, hiszen m_1 modulusra x_1

maradékot ad (mivel a második tag osztható m_1-gyel), m_2-re pedig $x_1+x_2-x_1 = x_2$-t (mivel $q_1m_1 = 1 \bmod m_2$). □

A tétel alapján egy nagy modulus esetén lehet úgy számításokat végezni, hogy a modulust felbontjuk $m = m_1m_2...m_k$ módon, majd a számításokat elvégezzük a k darab kisebb modulussal – nevezzük ezt maradékrendszernek –, és ezek alapján kiszámoljuk az eredeti számítás eredményét.
Ehhez szükséges egy algoritmus, ami hatékonyan kiszámítja a maradékrendszer tagjaira vett maradékokat, illetve egy olyan, ami az ismert maradékrendszer alapján rekonstruálja az eredeti számot, ami a kínai maradéktétel alapján egyértelműen meghatározható.

3.5.1. A maradékrendszer előállítása

A következő rekurzív algoritmus az x egész szám esetén meghatározza az $x_i = x \bmod m_i$ maradékokat $i = 1, ..., k$-ra. Az algoritmus az egyszerre több inverz számítás alapötletéhez hasonló módon működik, hiszen az elején adott x, és egy lépésben kiszámol egy $x \bmod m_1...m_l$ maradékot és egy $x \bmod m_{l+1}...m_k$ maradékot, ahol l körülbelül $\frac{k}{2}$. Ezzel egy fát épít, aminek levelei a keresett maradékok, és egy szinttel lejjebb úgy kerülünk, hogy elvégzünk kettő maradékos osztást.

10. algoritmus IntToCode

Input: x n-bites szám, $m_1, m_2, ..., m_k$
Output: $x_i = x \bmod m_i$, $i = 1, 2, ..., k$

```
1: if k ≤ 2 then                                  ▷ Ekkor már levelekbe érkezünk
2:     return x_1 = x mod m_1,...,x_k = x mod m_k
3: end if
4: l := ⌊k/2⌋
5: M_1 := m_1...m_l
6: M_2 := m_{l+1}...m_k
7: x_1, ...x_l := IntToCode(x mod M_1, m_1, ..., m_l)
8: x_{l+1}, ...x_k := IntToCode(x mod M_2, m_{l+1}, ..., m_k)
```

Legyen x és $m = m_1...m_k$ egyaránt n-bites, valamint legyenek m_i modulusok egyforma méretűek. Ekkor egy M_1 és M_2-re bontás után a két szám bithossza fele lesz m bithosszának, így $x \bmod M_1$ és $x \bmod M_2$ is felezi a bithosszt.
Az algoritmus során végzett M_1 és M_2-re bontások előfeldolgozással is megadhatók, ha több műveletet is végzünk ugyanezzel a maradékrendszerrel. Ezen kívül a futásidőt a következő rekurzió határozza meg ($T(k)$ az algoritmus lépésszáma k-bites

inputra): $T(n) = D(\frac{n}{2}) + T(\frac{n}{2})$, hiszen minden rekurzív hívásnál feleződik az x paraméter bithossza, emellett végzünk 2 darab $\frac{n}{2}$-bites számokkal való osztást, aminek lépésszámát $D(\frac{n}{2})$-nel jelölünk (ez x mod M_1 és x mod M_2 kiszámítása). Ez a futásidőre $O(\log nD(n))$ nagyságrendet ad. $D(n)$-ről látható, hogy megegyezik a szorzás lépésszámának nagyságrendjével, amit $M(n)$-nel jelölünk.

3.5.2. Maradékrendszer dekódolása

A visszatranszformálás esetén az x_i mod m_i – ahol $i = 1, ..., k$ és a modulusok páronként relatív prímek – ismeretében kell meghatározni egy $0 \leq x < m_1...m_k$ számot.

11. algoritmus CodeToInt

Input: x_i maradék és m_i modulus, ahol $i = 1, ..., k$
Output: $0 \leq x < m_1...m_k$, amire $x_i = x$ mod m_i, ahol $i = 1, ..., k$

1: **if** $k = 1$ **then**
2:     **return** x_1
3: **end if**
4: $l := \lfloor \frac{x}{2} \rfloor$
5: $M_1 := m_1...m_l$
6: $M_2 := m_{l+1}...m_k$ ▷ M_1 és M_2 relatív prímek
7: $X_1 :=$**CodeToInt**$([x_1, ..., x_l], [m_1, ..., m_l])$
8: $X_2 :=$**CodeToInt**$([x_{l+1}, ..., x_k], [m_{l+1}, ..., m_k])$
9: $u, v =$ **GCD**(M_1, M_2) megfelelő outputjai, hogy $1 = uM_1 + vM_2$
10: $\lambda_1 := uX_2$ mod M_2
11: $\lambda_2 := uX_1$ mod M_1
12: $x := \lambda_1 M_1 + \lambda_2 M_2$
13: **return** x
14: **if** $x \geq M_1 M_2$ **then**
15:     $x = x - M_1 M_2$
16: **end if**

Az algoritmus helyessége a következő állításon múlik:

Állítás: A $\lambda_1 M_1 + \lambda_2 M_2$ kifejezés maradéka X_1 modulo M_1 és X_2 modulo M_2.

Bizonyítás: x modulo M_1 maradéka: $x = \lambda_1 M_1 + \lambda_2 M_2 = \lambda_2 M_2$ mod M_1. $\lambda_2 M_2 = vX_1 M_2 = (vM_2)X_1 = X_1$ mod M_1, hiszen $vM_2 = 1$ mod M_1 az euklideszi algoritmus miatt. $x = X_2$ modulo M_2 ugyanígy.

Ekkor a rekurzív hívások során most is egy fa épül, aminek leveleiben x_i maradékok vannak. Egy szinttel feljebb a $\lambda_1 M_1 + \lambda_2 M_2$ kiszámításával lépünk. Kezdetben a levelekben az x_i maradékok szerepelnek. Az állításból következik, hogy az első szintlépés után kapott csúcsok is az elvárt maradékokat adják az m_i modulusokkal,

például: x_1 és x_2 közös szülője legyen $X_{1,2}$, ekkor $X_{1,2} = x_1$ modulo m_1 és $X_{1,2} = x_2$ modulo m_2.
Ezután indukcióval látható, hogy ez minden szintlépés után teljesülni fog. A fa csúcsában végül a megfelelő x maradékot kapjuk, amire teljesül, hogy $x = x_i$ mod m_i minden $i = 1, ..., k$-ra.

Az algoritmus 5,6 illetve 9-es lépése mind előre kiszámítható, ezt akkor érdemes, ha többször is számolunk ezekkel a modulusokkal. Ezeken kívül a lépésszám az előző algoritmuséval egyezik, tehát $O(\log n M(n)$ feltéve, hogy x és az $m_1...m_k$ szorzat is n-bites, és az $m_1, ..., m_k$ számok azonos méretűek.

4. fejezet

Implementáció kriptográfiai elemzése

Ebben a rövid fejezetben a kriptográfiai célokra felhasznált algoritmusok egy feltöréséről ejtünk szót. Az ún. *timing attack* egy olyanfajta támadás, amikor az egyes algoritmusok futásideje alapján próbálja a támadó kitalálni az inputot. A módszert Paul Kocher publikálta 1996-ban. [9]

Tekintsük például az RSA titkosítást, aminek részeként a felhasználó egy a^b mod m hatványozást végez el. Itt m nyilvános, a esetlegesen megtudható, azonban a b szám egy titkos kulcs. Ha az áldozatnak több különböző a hatványalappal is el kell végeznie a számítást, akkor a támadó ezek futásidejét ismerheti.

Vegyük például a Bináris hatványozás algoritmus jobbról balra haladó változatát:

12. algoritmus Bináris hatványozás2

Input: a, b, m pozitív egészek, b bitjei: $e_k ... e_0$
Output: $x = a^b \bmod m$

1: $x := 1$
2: **for** $i = 0, ..., k - 1$ **do**
3: **if** $e_i = 1$ **then** $x = ax \bmod m$
4: **end if**
5: $x = x^2 \bmod m$
6: **end for**
7: **return** x

A támadás alapját egy leegyszerűsített esettel szemléltetjük. Tegyük fel, hogy az ax mod m számítás a legtöbb esetben gyorsan elvégezhető, azonban néhány a és x párra a futásidő szignifikánsan lassabb a moduláris négyzetre emelésnél is. Az ilyen párok az algoritmus ismeretében meghatározhatók.

Ilyen feltevés mellett a támadó az első j darab bit ismeretében ki tudja számítani a következő bitet. Az ismert bitekkel végzett számítás futásidejét és végeredményét

tudja a támadó, tehát adott x értéke a j. iteráció után. Ekkor ha az a és x pár olyan, hogy az $ax \bmod m$ művelet szignifikánsan lassú lenne, viszont a következő iteráció nem lassú, akkor a $(j+1)$. bit csakis 0 lehet. Ha pedig a futás minfen y esetén lassú a $(j+1)$. iterációban, akkor a $(j+1)$. bit valószínűleg 1 lesz. Ezzel a módszerrel az elejétől kezdve minden bit kitalálható.

Természetesen a feltevés nem teljesül a való életben, ezért a lehetséges bitsorozatoknak csak egy valószínűségét lehet becsülni, és abból létrehozni egy tippet.

A timing attack elkerülésére a megoldás az, hogy az algoritmusok az inputtól függetlenül mindig fix lépést tegyenek. A kriptográfiai célokra írt algoritmusoknál tehát esetenként ezt is figyelembe kell venni. Azonban a különböző fordítókban lefuttatott optimalizálások az algoritmus fix futásideje ellenére is lehetőséget adhatnak a timing attack-nak, így az implementáció során a legvégén is szükséges a program tesztelése.

5. fejezet

Összegzés

A dolgozat során betekintést nyerhettünk az egész számok műveleteit elvégző algoritmusok tárházába, amelyek a kriptográfiai eljárások alapkövei. A néhány tárgyalt példán keresztül is látható, hogy általában több módszer közül is lehet választani egy konkrét feladat megoldásához, attól függően, hogy éppen mi a legoptimálisabb.

A bevezetés során említett Ethereum Virtual Machine-en futó algoritmusok költséghatékonysága a sikerességük fő kulcsa. Minél optimálisabban van egy "smart contract" felépítve, annál szívesebben használják, így a motiváció egyértelműen adott.

Egy hatékony algoritmus írása mellett a felhasználásából adódhatnak egyéb elvárások. A 4. fejezetben mutatott "timing attack" egy jó példa arra, hogy előfordulhat, hogy az implementáció során a fix futásidőt is megkövetelik. Ez "smart contract"-ok írásakor gyakran előfordul.

Összességében a dolgozat bevezetést adott a számelméleti függvények kiszámításához szükséges algoritmusok hatékony megvalósításához. Emellett láthattuk, hogy a konkrét implementálás során számos további megfontolandó tényező is szerepel.

Köszönyetnyilvánítás

Ezúton szeretném megköszönni a témavezetőim segítségét, amelyek lehetővé tették a szakdolgozat megírását. Az érdekes témaajánlás és a rengeteg hasznos forrás mellett egy kutatás folyamatába is betekinthetést nyerhettem. Továbbá a komoly tapasztalatuk is sokat jelentett a munka során.

Emellett hálával tartozom eddigi tanáraimnak, barátaimnak, akiktől rengeteg ismeretet szereztem az évek során.

Irodalomjegyzék

[1] Paul Zimmermann Richard P. Brent. *Modern Computer Arithmetic*. 2011. ISBN: 978-0-521-19469-3.

[2] Király Zoltán. *Algoritmuselmélet*. Typotex kiadó, 2014. ISBN: 978 963 279 241 5.

[3] Yuri Ofman Anatolii Karatsuba. „Multiplication of multi-digit numbers on automata". (1962).

[4] Andrei Leonovich Toom. „The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers". (1963).

[5] Stephen A. Cook. „On the minimum computation time of functions". (1966).

[6] Volker Strassen Arnold Schönhage. „Schnelle Multiplikation großer Zahlen". (1971).

[7] Joris van der Hoeven David Harvey. „Integer multiplication in time O(n log n)". (2021). URL: `"https://hal.science/hal-02070778/document"`.

[8] Paul Barrett. „Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor". (1987).

[9] Paul C. Kocher. „Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". (1996). URL: `"https://paulkocher.com/doc/TimingAttacks.pdf"`.

[10] Lovász László. *Algoritmusok bonyolultsága*. Typotex kiadó, 2014. ISBN: 978 963 279 253 8.

[11] Gyarmati Edit Freud Róbert. *Számelmélet*. Nemzeti Tankönyvkiadó, 2000. ISBN: 963 19 0784 8.

[12] Kumbakonam Govindarajan Subramanian Shahram Jahani Azman Samsudin. „Efficient Big Integer Multiplication and Squaring Algorithms for Cryptographic Applications”. (2014). URL: `"https://downloads.hindawi.com/journals/jam/2014/107109.pdf"`.

[13] Murat Cenk Murat Burhan Ilter. „Efficient Big Integer Multiplication in Cryptography”. (2017). URL: `"https://dergipark.org.tr/en/download/article-file/2160206"`.

[14] Gavin Wood. „Ethereum: A secure decentralised generalised transaction ledge”. (2022). URL: `"https://ethereum.github.io/yellowpaper/paper.pdf"`.

Algoritmusjegyzék

1. **Írásbeli összeadás** 4
2. **Írásbeli szorzás** 5
3. **Toom–Cook-3** 8
4. **FFT** 11
5. **Barrett** 19
6. **GCD** 21
7. **Több inverz** 23
8. **Bináris hatványozás** 24
9. **K-Bináris hatványozás** 25
10. **IntToCode** 26
11. **CodeToInt** 27
12. **Bináris hatványozás2** 29