Tudományos és szimbolikus számítások

Vinkó Tamás

verzió: 2023. január 11.

Tartalomjegyzék

.

A tananyag az EFOP-3.5.1-16-2017-00004 pályázat támogatásával készült.

.

Bevezetés

Kezdjük egy egyszerűen tűnő kérdéssel, mit jelent a következő kifejezés:

$$42$$

A kérdés persze nem pontos, talán inkább az lenne a jobban idevágó változat, hogy milyen (programozási nyelvekből jól ismert) típusú a kifejezés. Az esetek többségében a válasz: 'egész szám'. Lehet azonban racionális szám, valós szám, egy egész szám modulo m (ahol m>42), stb.

Miért fontos ezt lerögzítenünk? Egy halmaz elemein végzett művelet függhet attól, hogy mi az alaphalmaz. Például $42/13$ nem definiált az egészek halmazán, viszont igen a valós számokon. Valamint számít a számítógépes reprezentáció is:

- a diszkrét halmazokat általában teljes pontossággal tudjuk ábrázolni,
- míg a folytonosakat csak közelítőleg vagy implicit módon (szimbolikusan).

A hagyományos numerikus kurzussal ellentétben, ahol lebegőpontos számokkal végeztünk közelítő számításokat, itt leginkább kifejezésekre, algebrai objektumokon végrehajtott eljárásokra koncentrálunk.

Tipikus algebrai számítások Bár a jegyzet az itt fölsorolt témakörök csak egy részét fedi le, a következők tartoznak ide: tetszőleges pontosságú aritmetika, nincs kerekítés, szimbolumokkal és változókkal vegyesen számolunk, (matematikai értelemben vett) függvényekkel számolunk, kifejezések átalakítása (lehetőleg automatikusan), és szimbolikus műveletvégzés.

Matematikai objektumok ábrázolása

A következő szinteket különböztetjük meg:

1. szint: az objektumok szintje, ahol ha matematikai értelemben azonosak a kifejezések, akkor nincs megkülönböztetés
   - $2 + 2$ és a $3 \cdot 3 - 5$ ugyanaz az objektum (számokon)
   - $(x + 1)^2(x - 1)$ és $x^3 + x^2 - x - 1$ ugyanaz (polinomokon)
2. szint: forma szintje, ahol eltérő ábrázolást megkülönböztetjük
3. szint: adatstruktúra szintje, ahol pedig a memória-tárolásban eltérő ábrázolásokat tekintjük különbözőknek
   - például az $x^3 + x^2 - x - 1$ polinom esetén ez történhet az együtthatók tömbjével, vagy láncolt listájával

Alapműveletek végrehajtása

Az aritmetikai algoritmusok implementációja és végrehajtása alapvetően két, egymástól különböző kontextusban történhet: numerikusan vagy szimbolikusan. Az alábbi táblázat néhány érdekes aspektust mutat arra vonatkozóan, hogy ez a két megvalósítás valóban különbözik egymástól:

	numerikusan	szimbolikusan
költség	olcsó, általában pontatlan	drága, mindig pontos
válasz	gyakran ad is valamilyen választ	gyakran nem ad választ
zérusok	a zérusok nem speciálisak	a zérusok gyakran egyszerűsítenek
zérus detektálás	elég pici a szám?	a kifejezés ekvivalens a 0-val?

Számítási tárrobbanás

Az algebrai algoritmusoknál fontos a tárigény vizsgálata is. Számítási tárrobbanás *expression swell* Ennek illusztrációjaként számítsuk ki egy 9 dimenziós Hankel mátrix sajátértékeit.

$$H_{i,j} = H_{i-1,j+1}$$

(a numerikus kurzuson megismert Hilbert mátrix ennek egy speciális esete)

A mátrix 9 dimenzióban így néz ki:

$$\begin{pmatrix} -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 \end{pmatrix}$$

A karakterisztikus polinomja, amelynek gyökei a mátrix sajátértékei:

$$\lambda^9 + \lambda^8 - 40\lambda^7 - 24\lambda^6 + 240\lambda^5 + 144\lambda^4,$$

ami átírható szorzat alakban:

$$\lambda^4(\lambda + 6)(\lambda^4 - 5\lambda^3 - 10\lambda^2 + 36\lambda + 24)$$

ami alapján megállapíthatjuk, hogy a 0 négyszeres gyöke, -6 egyszeres gyöke, valamint van még egy negyedfokú tag, ami további 4 sajátértéket eredményez. A negyedfokú polinomok gyökeit meg tudjuk határozni, hiszen van rá megoldóképlet. Az egyik gyök a négy közül:

$$5/4 + 1/12\,\sqrt{3}\sqrt{\frac{8\left(3142+18\,i\sqrt{8071}\right)^{2/3}+155\,\sqrt[3]{3142+18\,i\sqrt{8071}}-\ +1856}{\sqrt[3]{3142+18\,i\sqrt{8071}}}-\sqrt{3}\sqrt{\frac{8\left(3142+18\,i\sqrt{8071}\right)^{2/3}+155\,\sqrt[3]{3142+18\,i\sqrt{8071}}+1856}{\sqrt[3]{3142+18\,i\sqrt{8071}}}-666\,\sqrt{3}\sqrt[3]{3142+18\,i\sqrt{8071}}-930\,\sqrt[3]{3142+18\,i\sqrt{8071}}\sqrt{\frac{8\left(3142+18\,i\sqrt{8071}\right)^{2/3}+155\,\sqrt[3]{3142+18\,i\sqrt{8071}}+1856}{\sqrt[3]{3142+18\,i\sqrt{8071}}}}}}$$

ahol bosszantó módon még a legjobb igyekezetünk ellenére sem tudjuk úgy megjeleníteni a képletet, hogy az teljes egészében látszódjon. Nyilvánvalóan ha numerikus megoldást keresünk, akkor nincs ilyen probléma.

Heurisztikák, valószínűségi módszerek

A jegyzetben több olyan algoritmust is bemutatunk majd, amelyek valószínűségi alapon adnak választ. Ennek motivációja a szokásos, a tárigényt és a futási időt szeretnénk csökkenteni. Az ilyen típusú eljárásokat alapvetően két csoportba osztjuk:

1. Monte Carlo módszerek:
    - a rossz válasz pozitív valószínűségű
2. Las Vegas módszerek:
    - soha nincs rossz válasz, de lehet, hogy nempolinomiális ideig fut

1. fejezet

Aritmetika

1.1. Adatábrázolás

Egész számok Az egy gépi szóban ábrázolható számokat egyszeres pontosságú egészeknek nevezzük. Ezek helyiértékes ábrázolása:

$$n = \sum_{i=0}^{k-1} d_i B^i \qquad B \in \mathbb{Z}, B > 1, d_i \in \mathbb{Z}$$

alakú, amelyet praktikusan így tároljuk: $[d_0 d_1 \dots d_{k-1}]$. Ez a tárolás történhet például láncolt listában vagy csak simán egy tömbben.

Az egész számok másik lehetséges ábrázolása a moduláris ábrázolás, ahol egyszeres pontosságú, páronként relatív prímek moduláris képeként ugyancsak láncolt listában vagy tömbben tárolunk. Erről részletesebben a 2. fejezetben lesz szó.

Az adatstruktúra megválasztása befolyásolja az aritmetikai algoritmusok sebességét:

- az összeadás, kivonás és szorzás gyorsan megy a moduláris aritmetikában – ahogyan ezt majd látni fogjuk a 3. fejezetben;
- az oszthatósági kérdések azonban a helyiértékes ábrázolás esetén végezhetőek el hatékonyabban.

Racionális számok A racionális számokat egész számok párjainak tekintjük: számláló és nevező formában tároljuk. Célszerű a legnagyobb közös osztóval (a legnagyobb közös osztóról a fejezet végén lesz szó) egyszerűsített ábrázolást választani. A szám előjele a számláló előjele.

Polinomok A fentiek alapján itt elsősorban racionális együtthatós polinomokról van szó, amelyek azonban lehetnek többváltozósak is. Kétféle ábrázolás használatos:

- faktorizált ábrázolás, pl $(x^2 + 1)(y - 1)$
- vagy kiterjesztett ábrázolás, pl $x^2y - x^2 + y - 1$

1.2. Aritmetikai algoritmusok

Ebben a szakaszban az aritmetikai műveletek algoritmikus végrehajtását tárgyaljuk. Részletesen csak az egész számokra vonatkozó műveleteket nézzük meg, majd röviden kitérünk a polinomokra vonatkozó algoritmusokra is.

Egész számokra Lényegében mintha kézzel számolnánk, csak a számrendszer alapja lehet más.

- Az összeadás műveletigénye $\mathcal{O}(n)$.
- A szorzás $\mathcal{O}(nm)$ műveletigényű. A 3. fejezetben látjuk majd, ezt be lehet gyorsítani. Megjegyezzük továbbá, hogy speciális esetekre is léteznek megoldások, mint például
  - a két szám közül az egyik sokkal kisebb, mint a másik
  - négyzetre emelés
  - konstanssal történő szorzás (amikor ugyanazzal a konstans taggal több alkalommal is szorzunk)
- Az osztás pedig $\mathcal{O}(n(m+1))$ műveletigényű. Megjegyezzük, hogy az eredmény a két egész hányadosa és a maradék együtt. Érdekes módon sok programozási nyelvben ez a két művelet két külön utasítás.

Érdemes még megjegyezni, hogy egy N egész számra a gyökvonást az $\lfloor\sqrt{N}\rfloor$ alakban értelmezzük, tehát az a legnagyobb olyan egész szám, amely kisebb vagy egyenlő, mint N négyzetgyöke.

Továbbá fontos művelet még a számrendszer átváltó (például tízesből kettesbe).

Polinomokra Polinomok esetében az összeadás és szorzás lényegében ugyanúgy megy, mint egész számok esetén, csak a helyiértékek szerepét itt a hatványtagok veszik át, ráadásul az átvitelre nem kell ügyelni.

Egy $p(x)$ polinom fokszámát a $\deg(p)$ jelöli. Két (racionális együtthatós) $p(x)$ és $q(x)$ polinomra, ahol $\deg(p)$ geq $\deg(q)$ definiáljuk a $\text{quot}(p,q)$ hányadost és $\text{rem}(a,b)$ maradékot úgy, hogy

$$p(x) = \text{quot}(p,q)\cdot q + \text{rem}(p,q),$$

ahol $0 \le \deg(\text{rem}(p,q)) < \deg(q)$.

Kapcsolat a polinomok és az egész számok szorzása között Tegyük fel, hogy össze akarunk szorozni a $p(x)$ és $q(x)$ nem-negatív együtthatós polinomokat, amelyek n-nél kisebb fokszámúak, és az együtthatóik felső korlátja K. Vegyük most $X = \beta^k > nK^2$ a β alapra és szorozzuk össze a $a = p(X)$ és $b = q(X)$ egészeket, amelyeket a p és r polinomok $x = X$ helyén vett kiértékeléssel kapunk. Ha most $r(x) = p(x)q(x) = \sum c_i x^i$, akkor világos, hogy $r(X) = \sum c_i X^i$. Mivel $c_i < nK^2 < X$, ezért a c_i együtthatók leolvashatóak az $r(X)$-ből. Példaként tekintsük a

$$(6x^5+6x^4+4x^3+9x^2+x+3)(7x^4+x^3+2x^2+x+7)$$

szorzást, ahol tehát a fokszám $n = 6$-nál kisebb és az együtthatók felső korlátja $K = 9$. Vehetjük most $X = 10^3 > nK^2$, és végrehajtjuk az egész szorzást

$$6\,006\,004\,009\,001\,003 \times 7\,001\,002\,001\,007 = 42\,048\,046\,085\,072\,086\,042\,070\,010\,021,$$

ahonnan leolvashatjuk, hogy

$$42x^9+48x^8+46x^7+85x^6+72x^5+86x^4+42x^3+70x^2+10x+21$$

a végeredmény. Megfordítva, tegyük fel, hogy össze akarunk szorozni két egészet $a = \sum_{0\le i<n} a_i\beta_i$ és $b = \sum_{0\le j<n} b_j\beta_j$. Végezzük el a polinom szorzást a $p(x) = \sum_{0\le i<n} a_i x_i$ és $q(x) = \sum_{0\le j<n} b_j x_j$ polinomokkal, amelynek az eredménye legyen $r(x)$. Ezután számoljuk ki az $r(\beta)$ helyettesítési értéket, hogy megkapjuk ab értékét. Megjegyezzük, hogy az $r(x)$ együtthatói nagyobbak is lehetnek, mint β, sőt akár $n\beta^2$ értékig is mehet. Például $a = 123, b = 456$, és $\beta = 10$, kapjuk, hogy $p(x) = x^2+2x+3, q(x) = 4x^2+5x+6$, és a szorzatuk $r(x) = 4x^4+13x^3+28x^2+27x+18$, és $r(10) = 56088$. Ezek a példák tehát jól rávilágítanak a polinomokkal és az egészekkel végzett műveletek analógiájára, valamint mutatják annak korlátait is.

1.3. Néhány érdekes megvalósítás

Ebben a szakaszban bemutatunk néhány, az alapműveletekre vonatkozó érdekes implementációt. A példák tekinthetők programozási gyakorlatnak is, nagyon tanulságosak.

Két előjel nélküli egész szám szorzása. A feladat természetesen úgy érdekes, hogy nincs megengedve az összeadás használata. Ahelyett: bitműveletek a megengedettek, és természetesen lehet ciklus utasításokat használni.

```
multiply_no_operator(x, y){
  sum = 0;
  while (x){
    if (x & 1) sum = add_no_operator(sum, y);
    x >>= 1; y <<= 1;
  }
  return sum;
}

add_no_operator(a, b){
  sum = 0;
  carryin = 0;
  k = 1;
  temp_a = a; temp_b = b;
  while (temp_a || temp_b){
    ak = a & k;
    bk = b & k;
    carryout = (ak & bk) | (ak & carryin) | (bk & carryin);
    sum |= (ak ^ bk ^ carryin);
    carryin = carryout << 1;
    k <<= 1;
    temp_a >>= 1;
    temp_b >>=1;
  }
  return sum | carryin;
}
```

Két egész szám osztása. Számítsuk ki az x/y értékét úgy, hogy csak összeadás, kivonás és eltolás műveleteket használhatunk.

Az ötlet, hogy használjuk az alábbi rekurziót:

$$\frac{x}{y} = \begin{cases} 0, & \text{ha } x < y, \\ 1 + \frac{x-y}{y} & \text{különben.} \end{cases}$$

Ez nagyon hasznos képlet, viszont rekurzív hívásként implementálva, amennyiben a két szám különbsége nagyon nagy, akkor túl sok lépést végez az algoritmus, és könnyen betelhet a verem. Ezért érdemesebb a

$$\frac{x}{y} = \begin{cases} 0, & \text{ha } x < y, \\ 2^k + \frac{x-2^k y}{y} & \text{különben} \end{cases}$$

képletet használni, ahol k az a legnagyobb érték, amelyre $2^k y \leq x$. Ilyenkor a $2^k y$ értéke számolható y szám k darab balra léptetésével.

Szorzás és kivonás csak összeadással. A feladatunk, hogy írjunk eljárást a szorzás és a kivonás műveletekre úgy, hogy csak az összeadás operátort használhatjuk.

A kivonás viszonylag egyszerű, ha kihasználjuk, hogy $a-b = a+(-1)b$, és alkalmazzuk a következő eljárást, amely egy a számot negál:

```
neg = 0
d = a < 0 ? 1 : -1
while (a != 0) {
  neg += d
  a += d
}
return neg
```

Vegyük észre, hogy itt arról van szó, hogy kihasználjuk a túlcsordulást, amikor egy pozitív számból negatív lesz.

A szorzáshoz pedig ismételt összeadást használhatunk, amely persze lassú, de működik.

1.4. Legnagyobb közös osztó

A négy alapművelet mellett talán az egyik legfontosabb művelet a legnagyobb közös osztó. A jegyzetben elsősorban egész számokra használjuk majd, de a fogalom természetesen értelmezhető (egyváltozós) polinomokra is: ebben az esetben a legmagasabb fokszámú közös osztót keressük.

Jelölése: $d = (a, b)$, vagy $d = \mathrm{lnko}(a, b)$, valamint használatos még a $d = \gcd(a, b)$ jelölés is (greatest common divisior).

Ha $d = 1$, akkor azt mondjuk, hogy a és b relatív prímek.

Kapcsolódó fogalom: legkisebb közös többszörös. Ennek jelölése és kiszámítása: $[a, b] = ab/(a, b)$.

1.4.1. Algoritmus egész számokra

Lássuk a több, mint 2000 éves algoritmust. Legyen az input két egész szám: $a_0 > a_1$. Számítsuk ki a következő maradékos osztás sorozatot:

$$\begin{aligned} a_0 &= q_1 a_1 + a_2 \quad (a_2 < a_1) \\ a_1 &= q_2 a_2 + a_3 \quad (a_3 < a_2) \\ &\vdots \\ a_{k-1} &= q_k a_k + a_{k+1} \quad (a_{k+1} < a_k). \end{aligned}$$

Amennyiben $a_{k+1} = 0$ teljesül, akkor álljunk meg.

Fontos megjegyzés, hogy a $(q_1, \ldots, q_k, a_k)$ sorozatot az a_0, a_1 *hányadosvektorának* nevezzük.

Tömörebb pszeudokód változat (itt is föltesszük, hogy $a > b$):

```
while (b != 0) {
  r = rem(a, b)
  a = b
  b = r
}
return a
```

A fenti kódban használt `rem(a,b)` eljárás az a-nak a b-vel vett osztásának maradékát adja vissza.

1.4.2. Algoritmus polinomokra

Az euklideszi algoritmus természetesen működik polinomok esetében is. Ugyanúgy, mint az egész számokra, itt is teljesül, hogy

$$\text{lnko}(p, q) = \text{lnko}(q, \text{rem}(p, q))$$

A pszeudokód egyébként pontosan megegyezik az egész számokra megadott kóddal azzal a kiegészítéssel, hogy a feltétel ebben az esetben a fokszámra vonatkozik: deg(a) >deg(b).

Példa. Határozzuk meg a

$$p(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x + 5$$

és

$$q(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

polinomok legnagyobb közös osztóját. A következőkben csak az r maradékpolinomokat adjuk meg, amelyek tehát

$$\begin{aligned} r_1 &= -\frac{5}{9}x^4 + \frac{1}{9}x^2 - \frac{1}{3} \\ r_2 &= -\frac{117}{25}x^2 - 9x + \frac{441}{25} \\ r_3 &= \frac{233150}{6591}x - \frac{102500}{2197} \\ r_4 &= \frac{1288744821}{543589225} \end{aligned}$$

A két megadott polinom tehát relatív prím.

1.4.3. Kiterjesztett euklideszi algoritmus

A fentiekben bevezetett hányadosvektor felhasználható arra, hogy a kiterjesztett euklideszi algoritmust definiáljuk, amelynek végrehajtásával az

$$\text{lnko}(a_0, a_1) = v_0 \cdot a_0 + v_1 \cdot a_1$$

alakot tudjuk előállítani. Vegyük észre, hogy itt a legnagyobb közös osztót, mint a két szám lineáris kombinációját írjuk fel. Megjegyezzük továbbá, hogy a v_0 és v_1 értékeket a moduláris algoritmusoknál (a 2. fejezetben) majd még felhasználjuk. Az algoritmus megadásához definiáljuk a következő mátrixot:

$$M_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix}.$$

Ekkor felírhatjuk a következő összefüggést:

$$\begin{pmatrix} a_i \\ a_{i+1} \end{pmatrix} = M_i \cdot \begin{pmatrix} a_{i-1} \\ a_i \end{pmatrix}.$$

Innen kapjuk, hogy

$$\begin{pmatrix} a_i \\ a_{i+1} \end{pmatrix} = M_i \cdot M_{i-1} \cdot \ldots \cdot M_1 \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.$$

Az $M_i \cdot M_{i-1} \cdot \ldots \cdot M_1$ mátrix szorzat a $q_1, \ldots, q_i$ sorozat meghatározásával együtt számolható. Legyen most

$$M_i \cdot M_{i-1} \cdot \ldots \cdot M_1 = \begin{pmatrix} u_{11}^{(i)} & u_{12}^{(i)} \\ u_{21}^{(i)} & u_{22}^{(i)} \end{pmatrix}.$$

Ekkor

$$a_k = u_{11}^{(k)} a_0 + u_{12}^{(k)} a_1,$$

tehát lényegében a v_0 és v_1 értékek egyfajta melléktermékek, amelyekből tehát előáll az

$$\text{lnko}(a_0, a_1) = v_0 \cdot a_0 + v_1 \cdot a_1$$

alak, amit kerestünk.

1.4.4. Euklideszi algoritmus költsége

A futási idő elemzéséhez használjuk fel Lamé tételét 1845-ből. Ehhez elevenítsük fel a Fibonacci számok rekurzív definícióját:

$$F_0 = 0, \;\; F_1 = 1 \quad \text{és} \quad F_n = F_{n-1} + F_{n-2}.$$

1.4.1. Tétel. *Legyenek $n \geq 1$, és $u > v > 0$ olyan pozitív egészek, hogy az u-ra és v-re elvégzett Euklidészi algoritmus pontosan n osztási lépést tartalmaz, továbbá u ezen belül a lehető legkisebb. Ekkor $u = F_{n+2}, v = F_{n+1}$, amely tehát nem más, mint két egymást követő Fibonacci szám.*

Ismeretes, hogy az F_n Fibonacci szám számjegyeinek száma éppen $n \cdot \log(\phi) - \log(5)/2 + 1$ egészrésze, ha $n > 1$ és $\phi = 1,618...$ az aranymetszés értéke. Tekinthetjük tehát úgy, hogy F_n számjegyeinek száma $\mathcal{O}(n)$. Hagyományosan egy osztás $\mathcal{O}(n^2)$-be kerül. Ezért a teljes költség $\mathcal{O}(n^3)$ lesz. Ugyanakkor lehet gyorsabban is osztani (ahogy azt tanuljuk is majd később), akkor $\mathcal{O}(n^2 \log n)$ a költség.

Végezetül megjegyezzük, hogy amennyiben ismerjük a prímfelbontásokat, akkor könnyű a legnagyobb közös osztó kiszámítása.

1.5. További megjegyzések

Több szám legnagyobb közös osztója. Ha az $u_1, \ldots, u_n$ egész számokra akarjuk a legnagyobb közös osztót kiszámítani, akkor a következő rekurziót kell alkalmaznunk:

$$\text{lnko}(u_1, \ldots, u_n) = \text{lnko}(u_1, \text{lnko}(u_2, \ldots, u_n))$$

Relatív prímek gyakorisága. Dirichlet tétele ad egy benyomást a relatív prímek előfordulásának gyakoriságáról. Ha u és v véletlen egészek, akkor annak a valószínűsége, hogy $\text{lnko}(u, v) = 1$ az

$$\frac{6}{\pi^2} \approx 0.60793.$$

Ez az eredmény azért releváns, mert amikor több szám legnagyobb közös osztóját számítjuk, akkor ezen eredmény szerint gyakran gyorsan végezhetünk (azzal az eredménnyel, hogy 1 a keresett érték).

Bináris lnko. Az algoritmusnak létezik egy változata, amelyben nincs szükség osztás műveletre, hanem csak kivonásra, páros/páratlan ellenőrzésre, és shift (páros szám bináris előállításának jobbra tolása, azaz felezése) műveletre.

Az algoritmus pszeudokódja az alábbi.

```
Bemenet: a, b pozitív egészek
Kimenet: g és d egészek úgy, hogy g páratlan és lnko(a, b) = g * 2d
d = 0
while (a és b egyaránt páros){
```

```
  a = a/2
  b = b/2
  d++
}
while (a <> b) {
  if (a páros) a = a/2
  else if (b páros) = b/2
  else if (a > b) a = (a - b)/2
  else b = (b - a)/2
}
g = a
return g, d
```

Lánctörtek. Az euklideszi algoritmus kapcsolatban van a lánctörtekkel is. Ez valójában az algoritmus kiterjesztése valós számokra. Tekintsük az lnko(a, b) meghatározásakor kapott együtthatókat. Meglepő módon az alábbi összefüggést kapjuk:

$$\frac{a}{b} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ldots q_{n-2} + \cfrac{1}{q_{n-1}}}}} = [q_0, q_1, \ldots, q_{n-1}]$$

Vegyük példaként az következőt:

$$\frac{180}{146} = [1, 4, 3, 2, 2].$$

Ha most kitöröljük az utolsó elemet, akkor kapjuk:

$$[1, 4, 3, 2] = \frac{37}{30} \Rightarrow -30 \cdot 180 + 37 \cdot 146 = 2 = \text{lnko}(180, 146).$$

Tudjuk, hogy minden valós számnak egyértelmű lánctört felírása van. A lánctört alakot előállító algoritmus akkor és csak akkor áll meg véges lépésben, ha az inputja egy racionális szám.

1.6. Feladatok

1. Írjunk egy eljárást, amely $\mathcal{O}(n)$ művelettel kiszámolja az x^y értékét, ahol y egy dupla pontosságú lebegő szám, x pedig egész.

2. Teszteljük le, hogy egy adott programozási nyelven (környezetben) az a^2 (négyzetre emelés) vagy pedig az $a \cdot a$ (önmagával történő szorzás) a gyorsabban elvégzett művelet. Terjesszük ki vizsgálatainkat magasabb hatványozásra is.

3. Írjunk osztás eljárást, amely csak összeadás műveletet használ.

4. Mutassuk meg, hogy lnko(x, y, z) az a legkisebb egész, amely kifejezhető $ax + by + cz$ alakban, ahol $a, b, c \in \mathbb{Z}$.

5. Implementáljuk a bináris legnagyobb közös osztó eljárás, és végezzünk részletes tesztelést a futási idejére, összevetve a hagyományos változattal.

6. Vajon lehetséges bárhogyan is letesztelni programozással azt, hogy egy adott számítógépen az összeadás művelet ugyanannyi ideig tart, mint a szorzás?

1.7. Kapcsolódó gyakorlat

Ezen a gyakorlaton a MATLAB Symbolic Toolbox csomagját használjuk.

Az első feladat, hogy implementáljuk az Euklidészi algoritmust. Természetesen a kihívás ebben az, hogy általánosan működjön, tehát nem csak egész számokra, hanem akár polinomokra is.

A MATLAB beépített eljárása erre: `[g,c,d] = gcd`, amely így ebben a formában a *kiterjesztett* Euklidészi algoritmust valósítja meg, amelynek hasznosságát a későbbiekben látjuk majd.

Rátérve a Symbolic Toolbox-ra: a `syms x` utasítással tudjuk a MATLAB-bal közölni, hogy az `x` egy szimbolum.

A `[q,r] = quorem(a,b)` utasítás az `a = q·b + r` műveletet végzi el.

Végső soron azzal is foglalkozunk, hogy vajon mi a különbség a `rem` és a `mod` utasítások között? Erre választ a rövid dokumentációjuk ad.

2. fejezet

Nagypontosságú aritmetika

Nagyméretű vagy nagypontosságú számolásnál a hagyományos (egyszeres pontosságú) aritmetika nem használható. Itt elsősorban a számolások közbeni részeredmények pontossága a kritikus fontosságú. Ebben a fejezetben tárgyalt moduláris aritmetika egy lehetséges megoldást ad erre a feladatra.

Az alábbiakban föltesszük, hogy a, b, m egész számok bár a tárgyalt fogalmak és algoritmusok (racionális együtthatós) polinomok esetén is működnek.

2.1. Kongruenciák

Diszkrét matematika tantárgyból már tanultuk, itt most csak gyorsan átvesszük a legfontosabb fogalmakat. A

$$b \equiv c \pmod{m}$$

jelentése[1]: a b és c számok m-mel osztva ugyanazt a maradékot adják. Másképpen is fogalmazhatunk:

- a különbségük osztható m-mel,
- van olyan k egész szám, hogy $b - c = km$.

Az egymással kongruens számok m darab osztályba: az m szerinti maradékosztályokba sorolhatók.

Aritmetikai műveletek kongruenciákkal. Tetszőleges m-re, ha

$$a \equiv b \pmod{m} \text{ és } c \equiv d \pmod{m}$$

akkor

$$a + c \equiv b + d \pmod{m} \text{ és } ac \equiv bd \pmod{m}.$$

Láthatjuk tehát, hogy az összeadás (a kivonás) és a szorzás műveletek könnyen elvégezhetők modulo m.

Az előzőek alapján nézzük most, hogyan boldogulunk az egyenletekkel. Például:

$$\begin{aligned} 5x + 3 &\equiv 6x - 8 \pmod{11} \\ 5x - 6x &\equiv -3 - 8 \pmod{11} \\ (6 - 5)x &\equiv 3 + 8 \pmod{11} \\ x &\equiv 11 \equiv 0 \pmod{11} \end{aligned}$$

Ez a speciális feladat könnyen megoldható volt, de ebből már sejtjük, hogy mi hiányzik: vajon hogyan tudunk osztani modulo m? Ebben segít a következő tétel.

[1] az egyenlőséget úgy olvassuk, hogy *b kongruens c-vel modulo m*

2.1.1. Tétel. *Ha b és m relatív prímek, akkor a $0, 1, \ldots, m-1$ maradékok között van pontosan egy olyan c, amelyre*

$$bc \equiv 1 \pmod{m}.$$

Ezt c számot a b *reciprokának* tekintjük modulo m.

A tétel ugyan definiálja a reciprokot — de vajon hogyan tudjuk meghatározni? Erre alkalmas az 1.4.3. szakaszban tárgyalt kiterjesztett euklideszi algoritmus, amelynek eredménye:

$$\text{lnko}(b, m) = ub + vm.$$

Amennyiben b és m relatív prímek, tehát $\text{lnko}(b, m) = ub + vm = 1$, akkor innen

$$1 \equiv ub \pmod{m}$$

adódik, tehát az u szám lesz b reciproka modulo m.

Lássuk most, hogyan tudjuk ezt alkalmazni az alábbi példára:

$$3x \equiv 5 \pmod{11}$$

A kiterjesztett Euklidészi algoritmust elvégezve kapjuk, hogy

$$\text{lnko}(3, 11) = 4 \cdot 3 + (-1) \cdot 11 = 1,$$

tehát a 4 lesz a 3 inverze modulo 11, ezért beszorozhatunk 4-gyel és kapjuk:

$$x \equiv 4 \cdot 3x \equiv 4 \cdot 5 \equiv 9 \pmod{11}$$

tehát a megoldás 9 (ami az 5/3 megfelelője modulo 11).

Alkalmazás: nagy számok moduláris szorzása. Ha m prímszám, akkor minden b számhoz relatív prím, amely nem osztható m-mel. Tehát minden olyan b számnak, amely nem kongruens 0-val mod m van reciproka mod m. Alkalmazzuk ezt a megfigyelést! Ha például az a feladtunk, hogy számítsuk ki a

$$88 \cdot 76 - 65 \cdot 92 \pmod{3}$$

értékét, akkor ehhez először számítsuk ki:

$$88 \equiv 1, \quad 76 \equiv 1, \quad 65 \equiv 2, \quad 92 \equiv 2 \pmod{3},$$

majd pedig innen:

$$88 \cdot 76 - 65 \cdot 92 \equiv 1 \cdot 1 - 2 \cdot 2 \equiv -3 \equiv 0 \pmod{3}.$$

Megjegyezzük, hogy esetleg hasznos lehet több modulusra is előre kiszámolni a fentieket, így adott esetben rendkívül gyorsan elvégezhetjük a moduláris szorzást.

2.2. Kínai maradéktétel

A fentiekben tárgyaltak egyik legfontosabb hasznosítását a következő híres tétel adja meg.

2.2.1. Tétel (Kínai maradéktétel)**.** *Tegyük fel, hogy $m_1, \ldots, m_k$ páronként relatív prímek. Ekkor*

A) az $m_1, \ldots, m_k$ szerinti maradékok meghatározzák az $M = m_1 \cdot \ldots \cdot m_k$ szerinti maradékokat is, azaz ha

$$\begin{aligned} b &\equiv c \pmod{m_1} \\ b &\equiv c \pmod{m_2} \\ &\vdots \\ b &\equiv c \pmod{m_k} \end{aligned}$$

teljesülnek, akkor $b \equiv c \pmod{M}$; valamint

B) minden $b_1, \ldots, b_k$ számsorozathoz van pontosan egy olyan M-nél kisebb szám, amelyre

$$\begin{aligned} b &\equiv b_1 \pmod{m_1} \\ b &\equiv b_2 \pmod{m_2} \\ &\vdots \\ b &\equiv b_k \pmod{m_k} \end{aligned}$$

és ezt a b számot ki is lehet számítani.

A tétel lehetővé teszi, hogy M-nél kisebb számokat az $m_1, \ldots, m_k$ szerinti maradékaikkal egyértelműen reprezentáljuk.

Az ábrázolásnak van egy rendkívüli előnye, ugyanis a maradékokon végezhetünk egyszerű (tehát egyszeres pontosságú, és emiatt hardver szinten támogatott sebességű) műveleteket.

Példa. Legyen $m = 3$ és $n = 5$. Ekkor $M = 3 \cdot 5 = 15$. A kapott hozzárendelések:

$$\begin{aligned} 0 &\to [0, 0] \\ 1 &\to [1, 1] \\ &\vdots \\ 8 &\to [2, 3] \\ 9 &\to [0, 4] \\ &\vdots \\ 14 &\to [2, 4], \end{aligned}$$

azaz a $0, 1, \ldots M - 1$ számokat reprezentáljuk egy számpárral:

$$x \to [x \mod m, x \mod n].$$

Természetesen ez nem csak két értékkel érvényes. Ha például a b számot a $b_1, \ldots, b_{100}$ sorozat, a c számot pedig a $c_1, \ldots, c_{100}$ sorozat reprezentál, akkor

- a $b + c$ számot a $b_1 + c_1, \ldots, b_{100} + c_{100}$ sorozat,
- a bc számot a $b_1c_1, \ldots, b_{100}c_{100}$ sorozat reprezentálja.

Ilyenkor egyébként a b_ic_i számot el lehet osztani maradékosan m_i-vel és a maradékot írni a helyére. Legyünk azonban óvatosak: a bc szorzatot csak addig határozzák meg $m_1, \ldots, m_k$ szerinti maradékai, amíg maga is kisebb, mint M.

Gyakorlati megvalósítás. Legyenek tehát $m_1, m_2, \ldots, m_k$ páronként relatív prím pozitív egészek. Legyen továbbá $M = m_1 \cdot \ldots \cdot m_k$ ezek szorzata, valamint $b_1, \ldots, b_k$ tetszőleges egészek.

- Legyen $M_j = M/m_j$, ahol $j = 1, \ldots k$.
- Tudjuk, hogy lnko$(m_j, M_j) = 1$, hiszen $m_1, m_2, \ldots, m_k$ páronként relatív prímek.
- Így érvényes a 2.1.1. Tétel: tudjuk, hogy van olyan y_j egész, amely az M_j inverze modulo m_j. Másképpen:
$$M_j \cdot y_j \equiv 1 \pmod{m_j} \quad j = 1, \ldots, k. \tag{2.1}$$
- Legyen most
$$x = b_1 M_1 y_1 + b_2 M_2 y_2 + \ldots + b_k M_k y_k.$$
Mivel (2.1) teljesül minden j-re, ezért
$$x \equiv b_j M_j y_j \equiv a_j \pmod{m_j} \quad j = 1, \ldots, k. \tag{2.2}$$
Így tehát x ennek a k darabos kongruencia-rendszernek a megoldása.

Példa. Legyen ismét $m = 3$ és $n = 5$. Tudjuk, hogy lnko$(3, 5) = 1 = 2 \cdot 3 + (-1) \cdot 5$. Ha most a $[2, 3]$ pár eredeti számát akarjuk visszakapni, akkor

$$\begin{aligned} x &= 2 \cdot (-1) \cdot 5 + 3 \cdot 2 \cdot 3 \mod 15 \\ x &= -10 + 18 \mod 15 \\ x &= 8 \mod 15 \end{aligned}$$

vagyis a keresett szám az $x = 8$.

Példa. Legyen most $m_1 = 3, m_2 = 5, m_3 = 7$. Keressük a $[2, 3, 2]$ hármas eredeti számát. Ez azt jelenti, hogy keressük a

$$\begin{aligned} x &\equiv 2 \pmod 3 \\ x &\equiv 3 \pmod 5 \\ x &\equiv 2 \pmod 7 \end{aligned}$$

rendszer megoldását. Itt tehát $b_1 = 2, b_2 = 3, b_3 = 2$. A (2.2) alapján tudjuk, hogy x-et milyen alakban keressük. Világos, hogy $M = 3 \cdot 5 \cdot 7 = 105$, továbbá $M_1 = M/m_1 = 35, M_2 = M/m_2 = 21, M_3 = M/m_3 = 15$. Ezért

$$\begin{aligned} 35y_1 &\equiv 1 \pmod 3 \\ 21y_2 &\equiv 1 \pmod 5 \\ 15y_3 &\equiv 1 \pmod 7 \end{aligned}$$

Ezen kongruenciák megoldásához a modulo reciprokokat kell kiszámítani a kiterjesztett Euklidészi algoritmus futtatásával:

$$\begin{aligned} \text{lnko}(35, 3) = 1 &= \underline{-1} \cdot 35 + 12 \cdot 3 \\ \text{lnko}(21, 5) = 1 &= \underline{1} \cdot 21 + (-4) \cdot 5 \\ \text{lnko}(15, 7) = 1 &= \underline{1} \cdot 15 + (-2) \cdot 7 \end{aligned}$$

A fenti eredményekben aláhúzással jelöltük a keresett reciprok értékeket. Innen tehát kapjuk, hogy $y_1 = 2, y_2 = 1, y_3 = 1$. Összesítve az eredményeinket:

$$x = 2 \cdot 35 \cdot (-1) + 3 \cdot 21 \cdot 1 + 2 \cdot 15 \cdot 1 = 233 \equiv 23 \pmod{105}.$$

A keresett szám tehát a 23.

Példa. Legyen most $m_1 = 95, m_2 = 97, m_3 = 98, m_4 = 99$, ezek páronként relatív prímek. Végezzük el az $123684 + 413456$ összeadást.

- 123684 reprezentációja: $(123684 \mod 95, 123684 \mod 97, 123684 \mod 98, 123684 \mod 99) = (89, 9, 8, 33)$.
- 413456 reprezentációja: $(413456 \mod 95, 413456 \mod 97, 413456 \mod 98, 413456 \mod 99) = (16, 42, 92, 32)$.
- Számolhatunk akkor most moduláris aritmetikában az összeadással:

$$\begin{aligned} 123684 + 413456 &= (89, 9, 8, 33) + (16, 42, 92, 32) \\ &= (105 \mod 95, 51 \mod 97, 100 \mod 98, 65 \mod 99) \\ &= (10, 51, 2, 65). \end{aligned}$$

- Megoldandó tehát a

$$\begin{aligned} x &\equiv 10 \pmod{95} \\ x &\equiv 51 \pmod{97} \\ x &\equiv 2 \pmod{98} \\ x &\equiv 65 \pmod{99} \end{aligned}$$

  kongruencia-rendszer a szokásos módon (reciprok kereséssel, stb). A megoldás: $x = 537140$.

- Megjegyezzük, hogy az összeadás egyébként azért volt elvégezhető, mert mind a két szám, mind pedig az összegük kisebb, mint $95 \cdot 97 \cdot 98 \cdot = 89403930$.

Összefoglalva: ha egész számokon kell összeadást, kivonást és szorzást elvégezni, akkor megbecsüljük, hogy mekkorára nőhet az eredmény, majd választunk olyan $m_1, \ldots, m_k$ számokat, amelyek

- páronként relatív prímek,
- szorzatuk $M = m_1 \ldots m_k$ legalább kétszer akkora, mint az eredmény nagysága,
- egyenként nem hosszabbak egy (gépi) szónál.

Minden b bemenő adathoz kiszámoljuk a $b_1, \ldots, b_k$ számokat, amelyek rendre a b legkisebb maradékai az $m_1, \ldots, m_k$ maradékokra nézve. Ugyanazokat a műveleteket, amelyeket az eredeti számolás követel, most modulo m_1 hajtjuk végre. Tehát minden művelet végrehajtása után külön kiszámítjuk az eredmény legkisebb maradékát modulo m_1 és azzal számolunk tovább. Hasonlóképpen járunk el a modulo $m_2, \ldots$, modulo m_k maradékokkal is.

Ismerjük tehát a b' eredmény $b'_1, \ldots, b'_k$ maradékait. Kiszámoljuk a b'-höz tartozó $c'_1, \ldots, c'_k$ számokat. Ha szükséges, akkor előállítjuk a b' számot.

2.3. Nagyméretű lineáris egyenletrendszerek pontos megoldása

Tekintsük az

$$\mathbf{Ax} = \mathbf{b}$$

lineáris egyenletrendszert, ahol $\mathbf{A}$ egy $n \times n$-es mátrix, a $\mathbf{b}$ pedig egy n-dimenziós vektor.

A közelítő (lebegőpontos számításokkal elvégzett) megoldást már megtanultuk a Közelítő és szimbolikus számítások kurzuson. Ott láttuk is, hogy amennyiben az $\mathbf{A}$ mátrix rosszul kondicionált, akkor a közelítő megoldás reménytelenül rossz is lehet a kerekítési hibák miatt. Bár általában a bemenő adatok pontatlanságán már nem tudunk változtatni, célunk most, hogy kerekítési hibák nélkül számoljuk.

Ehhez tegyük fel, hogy $\mathbf{A}$ és $\mathbf{b}$ egészeket tartalmaznak (ha racionális számokkal adott, akkor felszorzunk, ha pedig lebegőpontos számokkal, akkor azokat racionális számoknak tekintjük és felszorzunk).

Felhasználjuk a jól ismert Cramer-szabályt, amihez definiáljuk a következő mennyiségeket:

$$u_1 = \begin{vmatrix} b_1 & a_{12} & \dots & a_{1n} \\ \dots & & & \\ b_n & a_{n2} & \dots & a_{nn} \end{vmatrix}, \quad u_2 = \begin{vmatrix} a_{11} & b_1 & \dots & a_{1n} \\ \dots & & & \\ a_{n1} & b_n & \dots & a_{nn} \end{vmatrix}, \quad \dots, \quad u_n = \begin{vmatrix} a_{11} & \dots & a_{1,n-1} & b_1 \\ \dots & & & \\ a_{n1} & \dots & a_{n,n-1} & b_n \end{vmatrix},$$

továbbá legyen K az $\mathbf{A}$ mátrix determinánsa: $K = \det(\mathbf{A})$. Ekkor a Cramer szabály:

$$\mathbf{x}_i = \frac{u_i}{K},$$

amiből az $\mathbf{Ax} = \mathbf{b}$ átalakításával kapjuk, hogy

$$\mathbf{Au} = K\mathbf{b},$$

és ha $K \neq 0$, akkor a megoldás egyértelmű.

Most becsülnünk kell a K értékét, amihez felhasználjuk a Hadamard tételét, amely szerint

$$\det(\mathbf{A}) = K \leq (a_{11}^2 + \ldots + a_{n1}^2)^{1/2} \cdot \ldots \cdot (a_{1n}^2 + \ldots + a_{nn}^2)^{1/2}.$$

Ha például minden i, j-re $a_{ij} < M$, akkor $K \leq (\sqrt{n}M)^n$.

A tétel következményeként kapjuk, hogy amennyiben $a_{ij} \leq M$ és $b_j \leq M$, akkor

$$u_i \leq (\sqrt{n}M)^n.$$

Mivel az $\mathbf{Au} = K\mathbf{b}$ egyenlet megoldásai az $u_1/K, \ldots, u_n/K$ számok, ezért ezeket a $K, u_1, \ldots, u_n$ számokat kell meghatároznunk.

Ehhez hajtsuk végre a következő lépéseket:

1. Szükségünk van egy prímszám-táblázatra, amely sok, M-nél kisebb, de ezen belül minél nagyobb prímszámokat tartalmaz. Válasszuk ki ezek közül p_1-et. Számoljuk ki a $K \mod p_1$ értékét. Ha ez a szám 0, akkor p_1 úgynevezett szerencsétlen prím, dobjuk el, és válasszunk helyette másikat, és számoljuk ki a maradékos osztást[2]. Legyen tehát

   $$K^{(1)} := K \mod p_1$$

   (a maradékok közül a legkisebb).

[2]Megjegyezzük, hogy annak a valószínűsége, hogy p_1 szerencsétlen meglehetősen kicsi, ugyanis $1/p_1$ a valószínűsége, hogy p_1 a K osztója és tudjuk, hogy p_1 jó nagy

2. Most $p_2, \ldots, p_s$-re ismételjük meg a fentieket. Ekkor kapjuk a $K^{(1)}, K^{(2)}, \ldots, K^{(s)}$ nemzérus sorozatot, valamint teljesül, hogy $p_1 p_2 \ldots p_s > (\sqrt{n}M)^n$.

   Ha például $n = 100$, akkor $M = 10^7$, a prímek közel vannak M-hez, akkor $s \approx 120$.

3. Oldjuk meg az
$$\mathbf{A}\mathbf{x} \equiv K^{(i)}\mathbf{b} \pmod{p_i} \quad (i = 1, \ldots, s)$$
   egyenletrendszereket. Minden ilyen rendszert úgy oldunk meg, mint a hagyományos lineáris egyenletrendszereket, csak minden lépés után érdemes redukálni a legkisebb maradékra, valamint az osztás helyett a modulo reciprokkal való szorzás műveletét alkalmazzuk (amelynek értékét a kiterjesztett euklidészi algoritmussal kapunk meg). Ezek eredménye:
$$\begin{aligned} \mathbf{x}^{(1)} &= (x_1^{(1)}, \ldots, x_n^{(1)}) \\ &\vdots \\ \mathbf{x}^{(s)} &= (x_1^{(s)}, \ldots, x_n^{(s)}). \end{aligned}$$

4. Végül használjuk a kínai maradéktételt: számítsuk ki azokat a legkisebb abszolút értékű $K, u_1, \ldots, u_n$ számokat, amelyekre
$$\begin{aligned} L \equiv K^{(1)}, u_1 \equiv x_1^{(1)}, \ldots, u_n \equiv x_n^{(1)} \pmod{p_1} \\ \vdots \\ L \equiv K^{(s)}, u_1 \equiv x_1^{(s)}, \ldots, u_n \equiv x_n^{(s)} \pmod{p_s} \end{aligned}$$
   Belátható, hogy pontosan a $K, u_1, \ldots, u_n$ számok adják az eredeti rendszerünk megoldását.

Vegyük észre, hogy a fenti módszer rendkívül jól párhuzamosítható: a kongruencia-rendszereket párhuzamosan is megoldhatjuk, ha van erre alkalmas hardverünk.

2.4. Hatványozás: ismételt négyzetre emeléssel

Először tekintsük azt az általános feladatot, hogy ki kell számolnunk az a^n hatványt, ahol $a > 0$ és $n > 0$ egész számok.

- Nyilván a naiv eljárás az $a^n = a \cdot a \cdot \ldots \cdot a$ alakot használja, ami $\mathcal{O}(n)$ műveletigényű.
- Vegyük észre, hogy amennyiben feltesszük, hogy n páros, akkor az $a^n = a^{(n/2)} \cdot a^{(n/2)}$ azonosságot kihasználva máris megspórolhatjuk a szorzások felét. Ezt a sémát azonban rekurzívan is használhatjuk, kiegészítve azzal az esettel, amikor n páratlan:
$$a^n = \begin{cases} 1 & \text{ha} n = 0, \\ a \cdot a^{n-1} & \text{ha } n \text{ páratlan}, \\ a^{(n/2)} \cdot a^{(n/2)} & \text{ha } n \text{ páros}. \end{cases}$$

Ebből egy rekurzív eljárást írni bármelyik erre alkalmas programozási nyelvben egyszerű gyakorlat.

Mi a helyzet, ha a fenti ötletet iteratívan akarjuk megvalósítani? Ehhez használjuk ki, hogy a hatvány bináris számrendszerbeli alakja
$$n = 2^k + n_{k-1}2^{k-1} + \ldots + n_1 2 + n_0,$$

ahol $n_i \in \{0, 1\}$. Ekkor ugyanis

$$a^n = a^{2^k} \cdot a^{n_{k-1}2^{k-1}} \cdot \ldots \cdot a^{n_1 2} \cdot a^{n_0},$$

Érdekes módon két különböző hatékonyságú algoritmust kapunk attól függően, hogy a hatványokon jobbról balra vagy balról jobbra haladunk végig.

Balról jobbra eljárás:

1. Legyen $B := a$.
2. Minden $i = k - 1, \ldots 0$-ra hajtsuk végre a következő lépéseket:
   (a) $B := B \cdot B$ `//itt van tehát a négyzetre emelés`
   (b) ha $n_i = 1$, akkor $B := B \cdot a$
3. A végeredmény: B.

Az algoritmus például az a^{13}-on értékét az $((a^2 \cdot a)^2)^2 \cdot a$ alakban számolja ki.
Számoljuk csak végig...

Hatványozás moduláris aritmetikában

A dolog szépsége, hogy az iménti, balról jobbra eljárást moduláris aritmetikában is használhatjuk. Ehhez azt kell látni, hogy

$$a^n \mod m = (a^{(n/2)} \mod m) \cdot (a^{(n/2)} \mod m) \mod m.$$

Legyen $c = a^{(n/2)}$. Tudjuk, hogy $c = m \cdot q + r$ alakú (ahol $r < m$), így $c \mod m = r$.
Továbbá $(a^{(n/2)} \mod m) \cdot (a^{(n/2)} \mod m) \mod m = r^2 \mod m$.
Ugyanakkor $a^n = c^2$, így $c^2 \mod m = (m \cdot q + r)^2 \mod m = (m^2 \cdot q^2 + 2mqr + r^2) \mod m = r^2 \mod m$.

A fenti, balról jobbra eljáráson tehát csak egy picit kell igazítanunk: minden lépésben $\mod m$ kell számolnunk.

1. Legyen $B := a$.
2. Minden $i = k - 1, \ldots 0$-ra hajtsuk végre a következő lépéseket:
   (a) $B := B \cdot B \mod m$
   (b) ha $n_i = 1$, akkor $B := B \cdot a \mod m$
3. A végeredmény: B.

Példa. Moduláris aritmetikában, például, a $8^{13} \mod 17$-et a következőképpen számoljuk:

$$\begin{aligned} 8^{13} &\equiv (((8^2 \cdot 8)^2)^2 \cdot 8 \equiv (((13 * 8)^2)^2) \cdot 8 \\ &\equiv (2^2)^2 \cdot 8 = 4^2 \cdot 8 \equiv 16 \cdot 8 \equiv 9 \mod 17. \end{aligned}$$

Ez természetesen sokkal gyorsabb, mint direktbe kiszámolni a 8^{13} értékét és maradékosan elosztani 17-tel.

2.5. Néhány érdekes probléma

Nagy pontosságú 'csalás'? Vizsgáljuk a következő állításokat:

$$\sum_{n=1}^{\infty} \frac{\lfloor n \tanh(\pi) \rfloor}{10^n} = \frac{1}{81}.$$

Ez a kiértékelés rossz, bár 268 jegyig pontos. Továbbá

$$\sum_{n=1}^{\infty} \frac{\lfloor n \tanh(\pi/2) \rfloor}{10^n} = \frac{1}{81}.$$

szintén rossz, bár az első 12 jegyig pontos. Mindkét végtelen sor értéke egyébként transzcendens szám.

Miért érdekes egyáltalán ez a kérdés? A $\tanh(\pi)$ és $\tanh(\pi/2)$ értéke közel van 1-hez, ezért az lehet az érzésünk, hogy $\lfloor n \tanh(\pi) \rfloor = n - 1$ nagyon sok n-re. Továbbá

$$\sum_{n=1}^{\infty} \frac{n-1}{10^n} = \frac{1}{81}.$$

ezért volt a fenti sejtésünk. Nézzük most a lánctört alakot:

$$\tanh(\pi) = [0, 1, \underline{267}, 4, 14, 1, 2, 1, 2, 2, 1, 2, 3, 8, 3, 1, \cdots]$$

és

$$\tanh(\pi/2) = [0, 1, \underline{11}, 14, 4, 1, 1, 1, 3, 1, 295, 4, 4, 1, 5, 17, 7, \cdots]$$

Nem lehet véletlen, hogy mindkét lánctörtben a harmadik szám az, amely egyel kisebb csak a fentebb mutatott pontosságnál! A magyarázat túlmutat a jegyzet keretein, a részleteket a [**?**] könyvben megtalálhatjuk.

Elég pontos? A

$$\sum_{n=1}^{\infty} \frac{\lfloor n e^{\pi \sqrt{163/9}} \rfloor}{2^n} = 1280640$$

kiértékelés pontos az első kb *félmilliárd* jegyre.

A π értéke nem 22/7. A π értékére a $22/7$ racionális közelítést még Arkhimédész adta. Amennyiben a MATLAB Symbolic Toolbox-át használjuk, akkor

```
>> syms x
>> f=(x^4*(1-x)^4)/(1+x^2);
>> int(f,[0,1])
ans =
  22/7 - pi
```

Amit itt történik, az tulajdonképpen azon alapszik, hogy

$$\int_0^t \frac{x^4(1-x)^4}{1+x^2} dx = \frac{1}{7}t^7 - \frac{2}{3}t^6 + t^5 - \frac{4}{3}t^3 + 4t - 4\arctan(t).$$

Innen vegyük észre, hogy

$$\int_0^1 x^4(1-x)^4 dx = \frac{1}{630},$$

valamint

$$\frac{1}{2}\int_0^1 x^4(1-x)^4dx < \int_0^t \frac{x^4(1-x)^4}{1+x^2}dx < \int_0^1 x^4(1-x)^4dx.$$

Innen azt kapjuk, hogy

$$223/71 < 22/7 - 1/630 < \pi < 22/7 - 1/1260 < 22/7$$

amely már elvezet a

$$3\frac{10}{71} < \pi < 3\frac{10}{70}$$

becsléshez.

2.6. Feladatok

1. Végezzünk numerikus kísérelteket arra vonatkozóan, hogy kiderítsük: vajon mekkora méretű számokkal végzett műveletek esetében éri meg használni a kínai maradéktételt?

2. Adott két egyenes, döntsük el, hogy metszik-e egymást (a feladat megoldásához használjuk nagypontosságú aritmetikát).

3. Maradék fa (remainder tree). Az $n \mod p_1, n \mod p_1, n \mod p_3, n \mod p_4$ kiszámításához használhatjuk a maradék fa elvét. Számítsuk ki $n \mod p_1p_2p_3p_4$ értékét, majd ezt redukáljuk modulo p_1p_2-vel, hogy megkapjuk $n \mod p_1p_2$ értékét, majd ezt redukáljuk modulo p_1-gyel, hoyg megkapjuk $n \mod p_1$-et, és így tovább. A feladat, hogy implementáljuk ezt az eljárás.

4. Írjunk faktoriális számolót, teszteljük a határait a használt programozási nyelv beépített típusaival. Hogyan változik a végrehajtási idő és a számítási kapacitás ha nagypontosságú aritmetikát használunk?

5. Kedvenc programozási nyelvünkön számítsuk ki az $5^{4^{3^2}}$ értékét. Győződjünk meg róla, hogy az első és az utolsó 20-20 számjegy:

$$62060698786608744707 \ldots 92256259918212890625.$$

3. fejezet

Gyors aritmetika

3.1. Nagy számok szorzása

Legyenek adottak u és v egész számok (10-es számrendszerben):

$$\begin{aligned} u &= u_{m-1}u_{m-2}\dots u_1u_0 = u_0 + 10u_1 + \dots + 10^{m-1}u_{m-1} \\ v &= v_{n-1}v_{n-2}\dots v_1v_0 = v_0 + 10v_1 + \dots + 10^{n-1}v_{n-1}. \end{aligned}$$

Ha összeszorozzuk őket, akkor a következő alakú eredményt kapjuk:

$$\begin{aligned} uv = w &= w_{m+n-1}w_{m+n-2}\dots w_1w_0 \\ &= w_0 + 10w_1 + \dots + 10^{m+n-1}w_{m+n-1}. \end{aligned}$$

Amennyiben ezt a 1.2 szakaszban tárgyalt, naiv szorzási algoritmussal számoljuk ki, akkor mn darab elemi műveletet végzünk el. Karacuba 1962-ben publikált észrevétele alapján azonban lehet gyorsabban is!

Az általánosság megszorítása nélkül tegyük fel, hogy u és v $2n$ jegyű egész számok, tehát

$$\begin{aligned} u &= u_{2n-1}u_{2n-2}\dots u_1u_0 \\ v &= v_{2n-1}v_{2n-2}\dots v_1v_0. \end{aligned}$$

Írjuk most fel őket a következő alakban:

$$u = 10^nU_1 + U_0 \qquad v = 10^nV_1 + V_0$$

ahol

$$\begin{aligned} U_1 &= u_{2n-1}\dots u_n \quad \text{(bal fele)} \\ U_0 &= u_{n-1}\dots u_0 \quad \text{(jobb fele)} \end{aligned}$$

valamint V_0 és V_1 hasonlóan. Amennyiben ezt a felírást használjuk, akkor a szorzatra a következő alakot kapjuk:

$$uv = 10^{2n}U_1V_1 + 10^n(U_0V_1 + U_1V_0) + U_0V_0,$$

ahol

$$U_0V_1 + U_1V_0 = (U_1 - U_0)(V_0 - V_1) + U_0V_0 + U_1V_1.$$

Innen pedig átrendezéssel:

$$uv = (10^{2n} + 10^n)U_1V_1 + 10^n(U_1 - U_0)(V_0 - V_1) + (10^n + 1)U_0V_0.$$

A fentiek szerint tehát 4 helyett 3 darab n-jegyű szorzás is elég. Megmutatható továbbá: az összeadások száma nem több, mint $9n$.

A formálisabb műveletigény vizsgálathoz legyen $T(n)$ két n jegyű szám *szorzásához* szükséges elemi műveletek száma. Teljesül, hogy

$$T(2n) \leq 3T(n) + 9n. \tag{3.1}$$

Teljesül továbbá, hogy $T(1) = 1$. Nézzük meg néhány értékre:

$$T(4n) \leq 3^2T(n) + 3 \cdot 9n + 2 \cdot 9n = 3^2T(n) + 9n(3^2 - 2^2)$$
$$T(8n) \leq 3^2T(n) + 9n(3^2 - 2^2) + 9 \cdot 2^2n = 3^3T(n) + 9n(3^3 - 2^3)$$

Az általános képlet

$$T(2^kn) \leq 3^kT(n) + 9n(3^k - 2^k), \tag{3.2}$$

amelynek bizonyításához teljes indukciót használunk. Tegyük fel, hogy (3.2) teljesül minden k-nál kisebb számra. Alkalmazzuk (3.1) képletet:

$$T(2 \cdot 2^{k-1}n) \leq 3T(2^{k-1}n) + 9 \cdot 2^{k-1}n.$$

Most alkalmazva az indukciós feltevést kapjuk, hogy

$$T(2^{k-1}n) \leq 3^{k-1}T(n) + 9n(3^{k-1} - 2^{k-1}).$$

Helyettesítsünk be az előbbi egyenlőtlenségbe:

$$\begin{aligned} T(2 \cdot 2^{k-1}n) &\leq 3 \cdot 3^{k-1}T(n) + 9n \cdot 3 \cdot (3^{k-1} - 2^{k-1}) + 9 \cdot 2^{k-1}n \\ &= 3^kT(n) + 9n(3^k - 2^k), \end{aligned}$$

amely tehát bizonyítja a (3.2) helyességét.

Ha most $n = 10$ és $k = 2$, akkor $T(40) \leq 9T(10) + 450$. Tudjuk, hogy $T(10) \leq 100$, ezért $T(40) \leq 1350$; hagyományos szorzással ez $40 \cdot 40 = 1600$ lenne.

Minél nagyobb n, annál nagyobb a nyereség. Az általános képlet két n jegyű szám összeszorzásának műveletigényére:

$$27n^{\log_2 3} \approx 27n^{1.85} = \mathcal{O}(n^{1.85}).$$

3.1.1. Implementáció

A következőkben megmutatjuk a Karacuba algoritmus egy lehetséges implementációját MATLAB-ban. A kódban a `cut` paraméter meghatározza, hogy mekkora számjegyű számok esetében használjuk a hagyományos szorzást.

```
function uv = Kara(u,v,cut)
  m=floor(log10(u))+1; # u számjegyeinek száma
  n=floor(log10(v))+1; # v számjegyeinek száma
  N=max(n,m);
  if (N<cut)
      return u*v;
  end
  k=floor(N/2);
  pow10k = power(10,k);
  U0=rem(u,pow10k);  U1=floor(u./pow10k);
  V0=rem(v,pow10k);  V1=floor(v./pow10k);
```

```
  T0=Kara(U1,V1,cut);
  T1=Kara(U1-U0,V0-V1,cut);
  T2=Kara(U0,V0,cut);
  uv = (power(10,2*k) + pow10k)*T0 + pow10k*T1 + (pow10k+1)*T2;
end;
```

3.1.2. Karacuba a gyakorlatban

Amennyiben n jegyű számokat kell összeszoroznunk, akkor az általános szabály, hogy alkalmazzuk a hagyományos $\mathcal{O}(n^2)$ algoritmust $n < 16$ esetben, a Karacuba módszert $16 \leq n < 4096$ között és a később tárgyalandó FFT módszert (lásd 4.5.1. szakasz[1]) nagyobb n-ekre.

3.2. Nagy mátrixok szorzása

Legyenek $\mathbf{A}$ és $\mathbf{B}$ $n \times n$-es mátrixok. Az $\mathbf{AB} = \mathbf{C}$ szorzatmátrix elemeinek kiszámítása a

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

képlettel történik. Mivel a k, i és j indexeket 1-től n-ig kell futtatni, ezért adódik, hogy a műveletigény n^3 szorzás és $n^2(n-1)$ összeadás.

Az alábbiakban bemutatjuk Strassen módszerét, amely kihasználja, hogy bizonyos számolt részeredmények többször is felhasználhatóak a C mátrix elemeinek kiszámolásához.

A Strassen-képleteket 2×2-es $\mathbf{A}$ és $\mathbf{B}$ mátrixokra mutatjuk meg. A mátrix elemeire a szokásos jelöléseket használva definiáljuk:

$$\begin{aligned}
I &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
II &= (a_{21} + a_{22})b_{11} \\
III &= a_{11}(b_{12} + b_{22}) \\
IV &= a_{22}(b_{21} + b_{11}) \\
V &= (a_{11} + a_{12})b_{22} \\
VI &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
VII &= (a_{12} + a_{22})(b_{21} + b_{22})
\end{aligned}$$

Ellenőrizhető, hogy

$$\begin{aligned}
c_{11} &= I + IV - V - VII, & c_{12} &= III + V, \\
c_{21} &= II + IV, & c_{22} &= I + III - II + VI.
\end{aligned}$$

Azt kaptuk tehát, hogy az eredeti 8 szorzás és 4 összeadás helyett 7 szorzás és 18 összeadás elvégzése szükséges.

Mivel a fenti képletek nem használják fel a kommutativitást, ezért azok 4×4-es mátrixokra is használhatók, ahol az a_{ij} és b_{ij} elemek 2×2-es blokkokat jelölnek. Ezt rekurzívan folytatva használhatjuk őket 2^k rendű mátrixok összeszorzására.

Nézzük meg most a műveletigényt formálisan is. Legyen $n \times n$-es mátrixok szorzásának műveletigényében $S(n)$ az összeadások és kivonások száma, és $M(n)$ a szorzások száma. Strassen képleteket

[1] bár a 4.5.1 szakaszban a gyors Fourier transzformációt polinomok konvolúciójának kiszámítására tárgyaljuk, a módszer természetesen (nagyméretű) számok szorzására is alkalmazható

alkalmazva két négyzetes 2^k rendű mátrix összeszorzásához $2^{k-1} \times 2^{k-1}$-es blokkok 7 szorzatát kell kiszámolnunk. Ez $7M(k-1)$ szorzást jelent, így

$$M(k) = 7M(k-1).$$

Mivel $M(0) = 1$, azt kapjuk, hogy

$$M(k) = 7^k = 2^{k \log_2 7} = n^{\log_2 7}$$

ahol $n = 2^k$ a mátrixok mérete volt.

Az összeadást és kivonást illetően $2^{k-1} \times 2^{k-1}$-es blokkok 18 összeadását és kivonását, valamint 7 szorzását kell elvégeznünk. Az utóbbi $7S(k-1)$, az előbbi pedig $18(2^{k-1})^2 = 9 \cdot 2^{2k-1}$ összeadást jelent. Így

$$S(k) \leq 9 \cdot 2^{2k-1} + 7S(k-1).$$

Vezessük be a $B(k) = 7^{-k}S(k)$ jelölést, amivel a rekurzió átírható

$$B(k) = \frac{9}{2}\left(\frac{4}{7}\right)^k + B(k-1)$$

alakba. k szerint összeadva, és kihasználva, hogy $B(0) = 0$ kapjuk, hogy

$$B(k) = \frac{9}{2}\sum_{j=1}^{k}\left(\frac{4}{7}\right)^j < \frac{9}{2} \cdot \frac{4}{3} = 6.$$

Ebből tehát

$$S(k) \leq 6 \cdot 7^k = 6 \cdot 2^{k \log_2 7} = 6n^{\log_2 7} = 6n^{2,807}$$

Megállapíthatjuk tehát, hogy Strassen módszere $\mathcal{O}(n^{2,81})$ műveletigényű. Érdekességként megjegyezzük, hogy Strassen publikációjának címe: *Gaussian elimination is not optimal*.

Amennyiben a szorzandó mátrixok rendje nem 2 hatvány, akkor megfelelő mennyiségű 1 hozzá vételével a főátlóba 2-hatvány rendűvé tehetőek.

Az a sejtés, hogy minden $\varepsilon > 0$-ra létezik olyan algoritmus amely két $n \times n$-es mátrixok szorzatát $cn^{2+\varepsilon}$ művelettel számol ki, ahol a c konstans az ε függvénye. A jelenlegi csúcs: $c \cdot n^{2,3728639}$ (a gyakorlatban használhatatlanul nagy c értékkel).

3.3. Mátrixok invertálása

Bár mátrixok invertálása a gyakorlatban egy olyan művelet, amelyet lehetőség szerint kerülni kell, Frobenius módszere az invertálásra némi tanulsággal szolgál.

Tegyük fel, hogy A egy olyan mátrix, amely blokk-formában fölírható

$$\begin{pmatrix} P & Q \\ R & S \end{pmatrix}$$

ahol P négyzetes és nemszinguláris, továbbá $U = S - R(P^{-1}Q)$ szintén nemszinguláris. Akkor

$$A^{-1} = \begin{pmatrix} P^{-1} + (P^{-1}Q)(U^{-1}RP^{-1}) & -(P^{-1}Q)U^{-1} \\ -(U^{-1}RP^{-1}) & U^{-1} \end{pmatrix}$$

Az állítás helyessége egyszerű ellenőrzéssel bizonyítható. Vegyük észre, hogy a számítás két mátrixinvertálást (P és U) és hat darab mátrixszorzást (rendre $P^{-1}Q, R(P^{-1}Q), RP^{-1}, U(RP^{-1}), (P^{-1}Q)(U^{-1}RP^{-1})$ és $(P^{-1}Q)U^{-1}$) igényel.

3.4. Számok osztása

Olyan algoritmust keresünk, amely két n jegyű szám hányadosának n jegyig pontos meghatározását $\mathcal{O}(n^2)$ elemi művelettel megadja. Meglepő módon mindezt a Newton-módszer biztosítja[2].

Amennyiben adott az $f : \mathcal{R} \to \mathcal{R}$ differenciálható függvény, akkor a Newton-iteráció képlete:

$$x_{m+1} = x_m - \frac{f(x_m)}{f'(x_m)},$$

feltéve, hogy $f'(x_m) \neq 0$.

Most az $1/u$ kiszámításához az $1/x - u = 0$ egyenletet kell megoldanunk. Alkalmazzuk erre a Newton képletet, ahol tehát az $f(x) = 1/x - u$ függvényt használjuk:

$$x_{m+1} = x_m - \frac{1/x_m - u}{-1/x_m^2} = 2x_m - ux_m^2 = x_m(2 - ux_m) \tag{3.3}$$

Megjegyezzük, hogy szokás még az

$$x_{m+1} = x_m + x_m(1 - ux_m)$$

alakot is használni. Ez matematikailag ekvivalens az előző képlettel, ugyanakkor a számítógépes implementációban előnyösebb lehet. Ahhoz, hogy az eredményt $2n$ bit pontosságra megkapjuk az (3.3) képlet használatával, ahhoz ki kell számolni a szorzatot x_m és $(2 - ux_m)$ között ahol x_i pontossága adott (n bit). Ezzel szemben a szorzatot az x_i és $(1 - ux_i)$ között csak n bitre pontosan kell, hogy kiszámítsuk, hiszen az első n bit (a bináris pont után) az $(1 - ux_i)$-ben mind nullák.

Ha az $1/u$ számot n értékes jegy pontossággal akarjuk tudni, akkor elég x_k-t kiszámítani, ahol

$$k = \lceil \log n \rceil + 1.$$

Amennyiben a hiba adott úgy, hogy $\epsilon_i = 1 - ux_i$, akkor

$$\begin{aligned} \epsilon_{i+1} &= 1 - ux_{i+1} \\ &= 1 - u(x_i(2 - ux_i)) \\ &= 1 - 2ux_i + u^2x_i^2 \\ &= (1 - ux_i)^2 \\ &= {\epsilon_i}^2. \end{aligned}$$

Ez itt nem más, mint a jól ismert kvadratikus konvergencia.

A kezdeti érték megválasztása egy érdekes probléma lehet. Érdemes az u értékét átskálázni úgy, hogy $0,5 \leq D \leq 1$ teljesüljön, természetesen úgy, hogy a számlálót is ugyanekkor értékkel eltoljuk. Ezután a következő lineáris közelítő formát használhatjuk:

$$x_0 = T_1 + T_2u \approx \frac{1}{u}$$

a kezdeti érték választásához. Ahhoz, hogy a hiba abszolút értékének maximumát minimalizáljuk a $[0.5, 1]$ intervallumon, használjuk a

$$x_0 = \frac{48}{17} - \frac{32}{17}u$$

képletet.

[2]ezt a módszert a Közelítő és szimbolikus számítások kurzusokon nemlineáris egyenletek gyökeinek megkeresésére tanuljuk

Pszeudokód A következő pszeudokód a fentiekben tárgyalt Newton-módszer egy lehetséges megvalósítását írja le.

```
Vegyük u lebegőpontos alakját (kettes számrendszerben):
         M * pow(2,e) (ahol tehát 1<= M <2)
u' = u / pow(2, e+1)   // skálázzunk 0.5 és 1 közé
N' = N / pow(2, e+1)   // a számlálót is
x = 48/17 - 32/17 * u'
repeat ceil(log2((P+1)/log2(17))-szer
    x = x + x * (1 - u' * x)
end
return u' * X
```

3.5. Barrett redukció

A Barrett redukció a

$$c = a \bmod n.$$

naiv kiszámítását (amely persze az iménti, 3.4. szakaszban tárgyalt gyors osztás eljárást használná) optimalizálja. Természetesen csak bizonyos feltételek mellett érdemes használni, mégpedig akkor, ha n konstans és $a < n^2$ teljesül. Ekkor az osztást érdemes szorzásra cserélni.

Az általános elv a következő. Legyen $s = 1/n$ az n inverze lebegőpontos aritmetikával számolva. Akkor

$$a \bmod n = a - \lfloor as \rfloor n$$

ahol szokás szerint $\lfloor x \rfloor$ az alsó-egészrészt jelöli. Az eredmény pontos mindaddig, amíg az s értékét megfelelő pontossággal ki tudjuk számítani.

Barrett azt az esetet vizsgálta, amikor a számok elférnek egy gépi szóban. Tekintsük a következő `reduce` nevű függvényt:

```
function reduce(a)
  q = a / n  // egész osztás (alsó-egészrész az eredmény)
  return a - q * n
```

Barrett ötlete az volt, hogy az $1/n$ értékét közelítsük $m/2^k$ értékkel, hiszen a 2^k értékkel történő osztás elvégezhető jobbra léptetéssel. Az m értékének kiszámításához adott 2^k esetén használjuk az

$$\frac{1}{n} = \frac{2^k/n}{n(\frac{2^k}{n})} = \frac{2^k/n}{2^k} = \frac{m}{2^k}$$

összefüggést. Ahhoz, hogy m egész legyen a $2^k/n$ értékét kell valahogyan kerekítenünk. A legközelebbi egészre kerekítés a legjobb közelítést adhatja, azonban kaphatjuk hogy $m/2^k$ nagyobb lesz, mint $1/n$, ami alulcsordulást eredményezhet. Ezért általában az $m = \lfloor 2^k/n \rfloor$ számítási módot használjuk. Kapjuk tehát a módosított változatot a `reduce` függvényre:

```
function reduce(a)
  q = (a * m) >> k // a ">> k" jelentése: k-szoros bitléptetés
  return a - q * n
```

Mivel azonban $m/2^k \leq 1/n$ ezért a q értéke nagyon pici is lehet, emiatt aztán az a értékére csak a $[0, 2n)$ intervallumot tudjuk biztosítani a $[0, n)$ helyett. Egy megfelelő kivonás művelet azonban ezt is korrigálja, amivel megkapjuk a végleges változatot:

```
function reduce(a)
  q = (a * m) >> k
  a -= q * n
  if n <= a {
    a -= n
  }
  return a
```

3.6. Feladatok

1. Vessük össze a 3.1 szakaszban ismertetett Karacuba módszert a MATLAB beépített szorzás műveletével végrehajtási idő tekintetében. Ehhez próbáljuk meg a kódot minél jobban optimalizálni.

2. A számtani-mértani közép definíciója a következő: Legyen x és y két egész szám. Először számítsuk ki a számtani közepüket, amely legyen a_1, majd számoljuk ki a mértani közepüket, ezt jelölje g_1:

$$a_1 = \tfrac{1}{2}(x + y)$$
$$g_1 = \sqrt{xy}$$

   A kapott két számnak újra kiszámoljuk a számtani és a mértani közepét, és ezt iteráljuk minden a_n és g_n párra:

$$a_{n+1} = \tfrac{1}{2}(a_n + g_n)$$
$$g_{n+1} = \sqrt{a_n g_n}$$

   Ekkor az a_n és a g_n sorozatok ugyanahhoz a számhoz tartanak, ami x és y számtani-mértani közepe.

   A számtani-mértani közép meghatározása felhasználható, többek között a Gamma függvény kiszámításához. Derítsük ki, hogy hogyan, és implementáljuk az algoritmusokat.

3. A Gauss-Legendre algoritmus a π számjegyeinek gyors kiszámítására használható. Meglepő módon mindössze 25 lépésből 45 millió jegyre pontosan meghatározza a π értékét. A lépések a következők:

   (a) Kezdeti értékek:

$$a_0 = 1 \qquad b_0 = \frac{1}{\sqrt{2}} \qquad t_0 = \frac{1}{4} \qquad p_0 = 1.$$

   (b) Ismételjük az alábbiakat mindaddig, amíg az a_n és b_n különbsége a kívánt pontosságon belül van:

$$a_{n+1} = \frac{a_n + b_n}{2},$$
$$b_{n+1} = \sqrt{a_n b_n},$$
$$t_{n+1} = t_n - p_n(a_n - a_{n+1})^2,$$
$$p_{n+1} = 2p_n.$$

   (c) A közelítés értéke:

$$\pi \approx \frac{(a_{n+1} + b_{n+1})^2}{4t_{n+1}}.$$

   Implementáljuk az eljárást nagypontosságú aritmetika fölhasználásával.

4. Hogyan módosítanánk a Karacuba eljárást komplex számok szorzására? Naivan 4 darab szorzás és 2 összeadásra van szükség, de ez módosítható 3 szorzásra és 5 összeadásra.

5. Strassen algoritmusa polinomok gyors szorzására is használható, amelyet az alábbiakban vázolunk. A naiv eljárás $\mathcal{O}(mn)$ műveletigényű, ahol m és n a két polinom tagjainak a száma. Tegyük fel, hogy $n = m = 2k + 1$. A $P(x)$ és $Q(x)$ összeszorzásához tekintsük a következő alakot:

$$\begin{aligned} P(x) &= P_1(x) + x^{2^k} \cdot P_2(x) \\ Q(x) &= Q_1(x) + x^{2^k} \cdot Q2(x), \end{aligned}$$

ahonnan

$$P(x) \cdot Q(x) = P_1(x)Q_1(x) + x^{2^k} \cdot (P_1(x)Q_2(x) + P_2(x)Q_1(x)) + x^{2^{k+1}} \cdot P_2(x)Q_2(x).$$

Definiáljuk tehát a következő 3 szorzatot:

$$\begin{aligned} A(x) &= P_1(x)Q_1(x) \\ B(x) &= P_2(x)Q_2(x) \\ C(x) &= (P_1(x) + P_2(x)) \cdot (Q_1(x) + Q_2(x)) \end{aligned}$$

Akkor

$$P(x)Q(x) = A(x) + x^{2^k} \cdot (C(x) - A(x) - B(x)) + x^{2^{k+1}} \cdot B(x).$$

Mutassuk meg, hogy a futási idő $\mathcal{O}(3^k) = \mathcal{O}(n \log_2 3) = \mathcal{O}(n^{1,585})$.
Implementáljuk az eljárást, és végezzünk numerikus teszteket a végrehajtási időre (összehasonlítva a naiv módszerrel).

6. Olvasnivaló:
`https://randomascii.wordpress.com/2014/10/09/intel-underestimates-error-bour`

4. fejezet

Műveletek polinomokkal

4.1. Alapfogalmak

Ebben a fejezetben elsősorban egyváltozós polinomokra vonatkozó műveletek elvégzését tárgyaljuk. Az $x_1, \ldots, x_k$ változók polinomjának nevezünk minden olyan kifejezést, amely ezekből a változókból és valós számokból a szorzás, összeadás és kivonás segítségével építhető fel. Példa egy háromváltozós polinomra: $P(x_1, x_2, x_3) = 1/3 + x_1^3 - 5x_2x_3$

Mint azt korábban tárgyaltuk, a $P(x)$ egyváltozós $(n-1)$-ed fokú polinomot az együtthatóival szokás megadni.

Látjuk majd, hogy az algoritmusok műveletigényének meghatározásánál különbséget teszünk a lineáris és a nemlineáris műveletek között.

lineáris művelet: összeadás és olyan szorzás, ahol az egyik tényező ismert szám. Egy polinom helyettesítési értékének kiszámítása lineáris műveletek sorozatával elvégezhető.

nemlineáris művelet: olyan szorzás, amelynél mindkét tényező paramétert tartalmazó kifejezés vagy olyan osztás, amelyben a nevező paramétert tartalmaz.

4.2. Polinomok kiértékelése - Horner módszer

Tekintsük a

$$P(x) = a_0 + a_1x + \ldots + a_nx^n$$

egyváltozós polinomot. Ha a műveleteket úgy végezzük el, ahogyan azok ki vannak jelölve, akkor $n-2$ darab összeadást és $2n-1$ szorzást kell elvégeznünk.

Átalakítással azonban a szorzások száma n-re csökkenthető. Ezt a

$$P(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n)\ldots)$$

alakkal tudjuk elérni. Megjegyezzük, hogy se kevesebb szorzás, se kevesebb összeadás nem elegendő.

4.3. Polinomok kiértékelése - prekondicionálással

Tekintsük most azt az esetet, amikor a $P(x)$ polinomot kell kiértékelnünk **különböző** x helyeken. Ilyenkor bizonyos dolgokat megéri előre kiszámolni, mert azokat többször is felhasználjuk a számolás során. Természetesen a prekondicionáláshoz csak az $a_0, \ldots, a_n$ együtthatókat használhatjuk fel.

Érdekes módon itt arról van szó, hogy egy

$$Q(x) = b_0 + b_1 x + \ldots + b_{m-1}x^{m-1}$$

polinomot felfoghatjuk úgy, mint két vektor,

$$\mathbf{b} = (b_0, b_1, \ldots, b_{m-1}) \text{ és } \mathbf{x} = (1, x, \ldots, x^{m-1})$$

skalárszorzatát: $\mathbf{b}\mathbf{x}^T$. Tegyük fel, hogy m páros. Ekkor $Q(x)$ tehát felírható a következő alakban:

$$\begin{aligned} Q(x) &= (b_0 + b_{m-1}x^{m-1}) + (b_1 x + b_{m-2}x^{m-2}) + (b_2 x^2 + b_{m-3}x^{m-3}) + \ldots \\ &= \Big((b_0 + x^{m-1})(1 + b_{m-1}) + (b_1 + x^{m-2})(x + b_{m-2}) + \ldots - \frac{m}{2}x^{m-1}\Big) - \\ &\quad -(b_0 b_{m-1} + b_1 b_{m-2} + \ldots + b_{m/2-1}b_{m/2}). \end{aligned}$$

Nézzük meg ennek az alaknak a műveletigényét:

(i) az $x^2, x^3, \ldots, x^{m-1}$ előállításához $m-2$ darab szorzás (ezt vehetjük úgy, hogy nagyságrendileg m),

(ii) a $b_0 b_{m-1} + \ldots + b_{m/2-1}b_{m/2}$ előállításához $m/2$ darab szorzás,

(iii) míg a nagyobb zárójelben lévő kifejezés kiszámításához $m/2$ szorzás szükséges.

A (ii) alatti műveletek *előre* elvégezhetők, azok nem függnek az x értékétől, hiszen csak a $Q(x)$ polinom együtthatói szerepelnek benne. Az (i) alatti műveletekben a b_i együtthatók nem fordulnak elő[1].

Számolt műveletek száma: $m + m/2$. Ha most úgy gondolkodunk, hogy k darab prekondicionált n-ed fokú polinomot kellene *ugyanezen* a helyen kiértékelni, akkor ez a szám $km/2 + m$ lenne[2] Ezt a tulajdonságot fogjuk felhasználni a

$$P(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$$

polinom kiértékeléséhez, ahol $n \leq km$. Osszuk a tagokat m-es csoportokba:

$$P(x) = Q_0(x) + Q_1(x)x^m + Q_2(x)x^{2m} + \ldots + Q_{k-1}x^{(k-1)m}, \tag{4.1}$$

ahol

$$\begin{aligned} Q_0(x) &= a_0 + \ldots + a_{m-1}x^{m-1}, \\ Q_1(x) &= a_m + \ldots + a_{2m-1}x^{m-1}, \\ &\ldots \\ Q_{k-1}(x) &= a_{(k-1)m} + \ldots + a_{km-1}x^{m-1}, \end{aligned}$$

Mint láttuk, a $Q_i(x)$ polinomok kiszámítása összesen legfeljebb $km/2 + m$ műveletbe kerül, hiszen $i = 0, \ldots, k-1$, tehát pontosan k darab van belőlük. Ha x^m már megvan, akkor $P(x)$ kiszámítása a $Q_0(x), \ldots Q_{k-1}(x)$ együtthatókból[3] további $k-1$ művelettel elvégezhető. Az egész számolás így legfeljebb $km/2 + m + k$ műveletet követel, ahol $km \geq n$.

[1] Ennek akkor van jelentősége, ha több polinomot is ugyanazon az x helyen kell kiértékelni. De most azzal az esettel foglalkozunk, amikor *különböző* helyeken történik a kiértékelés.

[2] Ennek magyarázata: ilyenkor a fenti (i) alatti műveletekből (kihasználva azt, hogy mindegyik polinomot ugyanazon helyen kell kiértékelni) csak m darab szorzás kell, míg marad a (iii) alatti műveletek sora, amiből pedig $km/2$ kell, hiszen a polinomok együtthatói különbözőek.

[3] az (4.1) alapján a $P(x)$ együtthatói a $Q_0(x), \ldots, Q_{k-1}(x)$

Ha például $k = m = \lceil \sqrt{n} \rceil$, akkor a műveletigény $n/2 + 2\lceil \sqrt{n} \rceil$.

A polinom prekondicionálása tehát abból áll, hogy előre ki kell számolni és eltárolni k darab összeget, amelyek

$$s_j = a_{j \cdot m} a_{(j+1) \cdot m - 1} + a_{j \cdot m + 1} a_{(j+1) \cdot m - 2} + \dots$$

alakúak és $j = 0, \dots k-1$. A számítógépek programozási könyvtáraiban az elemi függvényeket rendszerint közelítő polinommal számolják. A polinomok gyakran ilyen prekondicionált formában vannak jelen.

4.4. Tabellázás

Azzal a feladattal folytatjuk, amikor a $P(x) = a_0 + a_1 x + \ldots + a_n x^n$ polinomhoz szeretnénk elkészíteni a

$$P(u_1), \dots, P(u_k)$$

táblázatot, ahol $u_1, \dots, u_k$ adott számok.

Amennyiben a 4.2. szakaszban tárgyalt Horner elrendezést használjuk, akkor ez kn darab nemlineáris művelet. Az iménti, 4.3. szakaszban ismertetett prekondicionálással ez $\approx kn/2$.

Lehet-e azzal nyerni, ha az egész táblázatot *egyszerre* számítjuk ki? Feltéve, hogy $k \geq n$ az alábbi eljárással $k \log^2 n$ művelettel ez megoldható, ahol a nemlineáris műveletek száma $k \log n$.

Az eljárást úgy fogjuk bemutatni, hogy a lépéseket egy nagyon egyszerű példán keresztül demonstráljuk. Feladatunk tehát az, hogy a $P(u_1), \dots, P(u_n)$ értékeket kell kiszámolni.

Példa: Számítsuk ki a $P(x) = x^2 + 3x + 4$ polinomot az $u_1 = 5$ és $u_2 = 6$ helyeken.

Ehhez kiszámoljuk az

$$I(x, u_1, \dots, u_n) = (x - u_1)(x - u_2) \dots (x - u_n)$$

polinom együtthatóit is, amelynek gyökei az $u_1, \dots, u_n$ számok.

Példa: $I(x, 5, 6) = (x-5)(x-6) = x^2 - 11x + 30$.

Ismert, hogy

$$I(x, u_1, \dots, u_n) = x^n - \sigma_1 x^{n-1} + \sigma_2 x^{n-2} - \dots$$

alakban írható fel, ahol

$$\begin{aligned} \sigma_1 &= u_1 + \ldots + u_n, \\ \sigma_2 &= u_1 u_2 + u_1 u_3 + \ldots + u_{n-1} u_n, \\ &\dots \\ \sigma_n &= u_1 u_2 \dots u_n \end{aligned}$$

Ezeket az $u_1, \dots u_n$ változók elemi szimmetrikus polinomjainak nevezzük.

Egy rekurzív eljárást adunk meg. Tegyük fel, hogy az algoritmus már adott, ha $P(x)$ foka kisebb, mint m. Most definiáljuk az eljárást minden olyan $P(x)$ polinomra, amelynek fokszáma kisebb vagy egyenlő, mint $2m$. Feltehetjük, hogy $P(x)$ fokszáma $n = 2m$ (ha nem, akkor nulla együtthatójú tagokkal kiegészítjük):

$$P(x) = a_0 + a_1 x + \ldots + a_{2m} x^{2m}.$$

Ki akarjuk számítani $P(x)$ értékét az $u_1, \dots, u_{2m}$ helyeken. Osszuk ezeket két részre: $u_1, \dots, u_m$ és $u_{m+1}, \dots, u_{2m}$. Képezzük az

$$I(x, u_1, \dots, u_m), \qquad I(x, u_{m+1}, \dots, u_{2m})$$

polinomokat – ezeket meg tudjuk határozni, hiszen feltettük, hogy az eljárás m-re már ismert. Osszuk el a $P(x)$ polinomot az $I(x, u_1, \ldots, u_m)$ polinommal:

$$P(x) = I(x, u_1, \ldots, u_m)Q_0(x) + R_0(x),$$

valamint az $I(x, u_{m+1}, \ldots, u_{2m})$ polinommal:

$$P(x) = I(x, u_{m+1}, \ldots, u_{2m})Q_1(x) + R_1(x).$$

Tudjuk, hogy az $R_0(x)$ és $R_1(x)$ maradékpolinomok fokszáma kisebb, mint m. Mivel az I polinom eltűnik az $u_1, \ldots, u_m$ és $u_{m+1}, \ldots, u_{2m}$ helyeken, ezért

$$P(u_1) = R_0(u_1), \ldots, P(u_m) = R_0(u_m)$$
$$P(u_{m+1}) = R_1(u_{m+1}), \ldots, P(u_{2m}) = R_1(u_{2m})$$

ahelyett, hogy a $2m$-ed fokú $P(x)$ polinomot értékelnénk ki az $u_1, \ldots, u_{2m}$ helyeken, az m-nél kisebb fokú $R_0(x)$ és $R_1(x)$ polinomokat értékeljük ki az $u_1, \ldots, u_m$ és $u_{m+1}, \ldots, u_{2m}$ helyeken.

Az m-nél kisebb fokú polinomokra az eljárás definiálva van.

Hátravan még az $I(x, u_1, \ldots, u_{2m})$ kiszámítása, de ez könnyű, mert

$$I(x, u_1, \ldots, u_{2m}) = I(x, u_1, \ldots, u_m)I(x, u_{m+1}, \ldots, u_{2m}).$$

Példa. A rekurzív fölbontás szerint

$$I_1(x, 5) = x - 5, \quad \text{és} \quad I_2(x, 6) = x - 6$$

Ezeket fölhasználva kapjuk, hogy

$$P(x) = (x - 5)(x + 8) + 44 \quad \text{és} \quad P(x) = (x - 6)(x + 9) + 58,$$

ezért $P(5) = 44$ és $P(6) = 58$.

4.5. Polinomok szorzása

A

$$P(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1} \quad \text{és} \quad Q(x) = b_0 + b_1x + \ldots + b_{n-1}x^{n-1}$$

polinomok szorzata az az

$$R(x) = P(x)Q(x) = c_0 + c_1x + \ldots + c_{2n-2}x^{2n-2}$$

polinom, amelynek c_i együtthatóit az alábbi képletek határoznak meg:

$$c_i = a_0b_i + a_1b_{i-1} + \ldots + a_ib_0.$$

A $(c_0, \ldots, c_{2n-2})$ sorozatot az $(a_0, \ldots, a_{n-1})$ és $(b_0, \ldots, b_{n-1})$ sorozatok *konvolúciójának* nevezzük. Erre egyébként tekinthetünk úgy is, mint két n dimenziós vektor olyan szorzata, amelynek eredménye egy $2n$ dimenziós vektor, amely tehát különbözik a hagyományos belső- vagy külső szorzattól (ahol rendre számot, illetve mátrixot kapunk eredményül).

Ha a $(c_0, \ldots, c_{2n-2})$ sorozatot a fentebb felírt módon akarjuk meghatározni, akkor n^2 darab szorzást kell elvégezni az a_i és b_j együtthatókkal. Azonban gyorsabban is lehet: Toom eljárással, amelyet a következőkben ismertetünk.

Kiindulásnak vegyünk $2n-1$ darab számot, például a

$$0, 1, \ldots, 2n-2$$

számokat. Most ahelyett, hogy az $R(x) = P(x)Q(x)$ polinom együtthatóit számolnánk ki, először számoljuk az $R(x)$ helyettesítési értékeit a $k = 0, 1, \ldots, 2n-2$ helyeken:

$$\begin{aligned} R(k) &= P(k)Q(k) \\ &= c_0 + c_1 k + c_2 k^2 + \ldots \\ &= (a_0 + a_1 k + a_2 k^2 + \ldots)(b_0 + b_1 k + b_2 k^2 + \ldots). \end{aligned}$$

Ehhez tehát ki kell számolni a $P(x)$ és $Q(x)$ helyettesítési értékeit a $0, 1, \ldots 2n-2$ helyeken, majd összeszorozni őket. Ezután pedig meg kell határoznunk az $R(x)$ együtthatóit az $R(0), R(1), \ldots, R(2n-2)$ értékekből. Vegyük észre, hogy ez az interpolációs feladat valójában a $c_0, \ldots, c_{2n-2}$ ismeretlenek kiszámítását jelenti a

$$\begin{aligned} c_0 &= R(0) \\ c_0 + c_1 + c_2 + \ldots + c_{2n-2} &= R(1) \\ c_0 + 2c_1 + 2^2 c_2 + \ldots + 2^{2n-2} c_{2n-2} &= R(2) \\ &\ldots \\ c_0 + (2n-2)c_1 + \ldots + (2n-2)^{2n-2} c_{2n-2} &= R(2n-2) \end{aligned}$$

lineáris egyenletrendszerből.

Ebben a pillanatban még az a benyomásunk, hogy semmit nem nyertünk, sőt egy lineáris egyenletrendszert kell megoldanunk, amelynek műveletigénye $O(n^3)$. Amennyiben viszont megint különbséget teszünk a lineáris és a nemlineáris műveletek között, akkor gyorsabbak is lehetünk.

Bár a

$$P(0)Q(0), P(1)Q(1), \ldots, P(2n-2)Q(2n-2)$$

értékek kiszámítása $2n-1$ darab nemlineáris művelet, a fenti lineáris egyenletrendszer (amely ugyanarra az eredményre vezet) megoldható csak lineáris művelettel! Ehhez vegyük észre, hogy paraméterek csak az $R(0), R(1), \ldots R(2n-2)$-ben vannak, ezeket pedig a megoldás során nem kell egymással se szorozni, se osztani. A nemlineáris műveletek száma tehát $2n-1$. A $(c_0, \ldots, c_{2n-2})$ együtthatókat definiáló *eredeti* képletben n^2 darab nemlineáris művelet fordult elő. Ebből a szempontból tehát a Toom-féle eljárás egyszerűbb.

Tovább is léphetünk: a lineáris műveletekből sem kellene sokat elvégezni. Ez könnyen teljesíthető, csak a helyettesítési értékeket kell megfelelően megadni! A fentiekben azt az esetet tárgyaltuk, amikor az $1, 2, \ldots, 2n-2$ értékekkel számoltunk. Azonban nem vagyunk ezekhez a számokhoz kötve. Ha ezeket komplex egységgyököknek választjuk, akkor a konvolúció megvalósítható $\mathcal{O}(n \log n)$ művelettel. Ez lesz a véges Fourier-transzformált módszer.

4.5.1. Véges Fourier-transzformált

Legyen $P(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ polinom. Legyen $r \geq n$ és

$$\omega = \omega_r = e^{\frac{2\pi\imath}{r}}$$

az első komplex r-edik egységgyök, és $\imath^2 = -1$. Az $\omega^0 = 1, \omega, \omega^2, \ldots, \omega^{r-1}$ számok mind különbözők és $\omega^r = 1$.

Képezzük most a $P(1), P(\omega), \ldots, P(\omega^{r-1})$ értékeket:

$$\hat{a}_k = P(\omega^k) = a_0 + a_1\omega^k + a_2\omega^{2k} + \ldots + a_{n-1}\omega^{(n-1)k}$$

az $(\hat{a}_0, \ldots, \hat{a}_{r-1})$ komplex számsorozatot az $(a_0, \ldots, a_{n-1})$ számsorozat *véges Fourier-transzformáltjának* nevezzük.

A transzformációnak két, számunkra most fontos tulajdonsága van:

- a visszatranszformálás hasonló képlettel történik:

$$a_j = \frac{1}{r}(\hat{a}_0 + \hat{a}_1\omega^{-j} + \hat{a}_2\omega^{-2j} + \ldots + \hat{a}_{r-1}\omega^{-(r-1)j}),$$

- valamint ha $r \geq 2n-2$, és a $(c_0, \ldots, c_{2n-2})$ sorozat az $(a_0, \ldots, a_{n-1})$ és $(b_0, \ldots, b_{n-1})$ sorozatok konvolúciója, akkor

$$\hat{c}_0 = \hat{a}_0\hat{b}_0, \;\; \hat{c}_1 = \hat{a}_1\hat{b}_1, \ldots, \hat{c}_{r-1} = \hat{a}_{r-1}\hat{b}_{r-1}.$$

A konvolúció tehát a következő műveletsorozat: transzformáció, r darab szorzás, visszatranszformáció.

Legyen $r \geq n$ és

$$F_k^{(r)}(a_0, \ldots, a_{n-1}) = \hat{a}_k \quad (k = 0, \ldots, r-1).$$

A $2r$ elemű Fourier-transzformáció nem más mint 2 darab r elemű Fourier-transzformáció. Ez a tény egy *divide and conquer* típusú algoritmusra vezet el bennünket.

Legyen $a_0, \ldots, a_{2r-1}$ egy $2r$ elemű sorozat. Ennek a transzformálásához az $\omega_{2r} = e^{\frac{2\pi\imath}{2r}}$ egységgyököket használjuk. Ezzel a jelöléssel:

$$\begin{aligned}
\hat{a}_k &= a_0 + a_1\omega_{2r}^k + a_2\omega_{2r}^{2k} + \ldots + a_{2r-1}\omega_{2r}^{(2r-1)k} \\
&= a_0 + a_2\omega_{2r}^{2k} + a_4\omega_{2r}^{4k} + \ldots + a_1\omega_{2r}^k + a_3\omega_{2r}^{3k} + \ldots \\
&= a_0 + a_2(\omega_{2r}^2)^k + a_4(\omega_{2r}^2)^{2k} + \ldots + \omega_{2r}^k[a_1 + a_3(\omega_{2r}^2)^k + a_5(\omega_{2r}^2)^{2k} + \ldots],
\end{aligned}$$

ahol tehát megfelelően csoportosítottuk az együtthatókat. Ha $\omega_{2r} = e^{\frac{2\pi\imath}{2r}}$, akkor $\omega_{2r}^2 = \omega_r$, ezért

$$F_k^{(2r)}(a_0, \ldots, a_{2r-1}) = F_k^{(r)}(a_0, a_2, \ldots, a_{2r-2}) + \omega_{2r}^k F_k^{(r)}(a_1, a_3, \ldots, a_{2r-1})$$

Megmutatható:

$$K(2r) \leq 2K(r) + 6r$$

továbbá $K(1) = 0$, ezért a fenti képletet m-szer egymás után alkalmazva

$$K(2^m) \leq 3m \cdot 2^m$$

Ha r a 2-nek egész kitevőjű hatványa, akkor

$$K(r) \leq 3r \log_2 r.$$

Szokás szerint: ha r nem 2 hatvány, akkor a sorozatot nullákkal egészítjük ki.

Gyors Fourier transzformáció: Konkrét megvalósítás

Legyen

$$\mathbf{a} = [a_0, a_1, \dots a_{n-1}]^T \quad \mathbf{b} = [b_0, b_1, \dots b_{n-1}]^T$$

két n-dimenziós vektor, amelynek komponensei komplex számok. az $\mathbf{a} * \mathbf{b} = \mathbf{c}$ konvolúciót akarjuk kiszámolni

Legyenek

$$F^{-1}(\mathbf{a}) := F^{-1}[a_0, a_1, \dots, a_{n-1}, 0, \dots 0] := (f_0, f_1, \dots, f_{2n-1})^T$$

és

$$F^{-1}(\mathbf{b}) := F^{-1}[b_0, b_1, \dots, b_{n-1}, 0, \dots 0) := (g_0, g_1, \dots, g_{2n-1})^T$$

a $2n$ komponensre kiegészített $\mathbf{a}$ és $\mathbf{b}$ vektorok véges inverz Fourier transzformáltjaik. Legyen továbbá

$$F^{-1}(\mathbf{a}) \circ F^{-1}(\mathbf{b}) := [f_0 g_0, f_1 g_1, \dots, f_{2n-1} g_{2n-1}]^T$$

a komponensenkénti szorzásokkal kapott vektor. Ekkor

$$F^{-1}(\mathbf{a} * \mathbf{b}) = F^{-1}(\mathbf{a}) \circ F^{-1}(\mathbf{b}).$$

A fenti, utolsó képlettel ekvivalens alak:

$$\mathbf{a} * \mathbf{b} = F[F^{-1}(\mathbf{a}) \circ F^{-1}(\mathbf{b})],$$

amely lehetővé teszi, hogy két n-dimenziós vektor konvolúcióját az FFT segítségével is kiszámoljuk, amennyiben n kettőnek egész kitevőjű hatványa.

4.6. Feladatok

1. A fejezetben nem volt szó polinomok osztásáról. A szakirodalomból keressünk egy erre alkalmas (gyors) algoritmust és implementáljuk, teszteljük.

2. Teszteljük le a MATLAB Symbolic Toolbox rendszerében elérhető polinom szorzás végrehajtási idejét összevetve a (lebegőpontos) `conv` függvénnyel, ami polinomok/vektorok konvolúcióját számítja ki.

3. A Horner módszer használható az osztott differencia

   $$\frac{p(y) - p(x)}{y - x}$$

   kiszámítására. Legyen adott a

   $$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots + a_n x^n$$

   polinom a következő lépéseket kell végrehajtanunk:

   $$\begin{aligned} b_n &= a_n, & d_n &= b_n, \\ b_{n-1} &= a_{n-1} + b_n x, & d_{n-1} &= b_{n-1} + d_n y, \\ &\vdots & &\vdots \\ b_1 &= a_1 + b_2 x, & d_1 &= b_1 + d_2 y, \\ b_0 &= a_0 + b_1 x. \end{aligned}$$

Ennek végén kapjuk,hogy

$$\begin{aligned} p(x) &= b_0, \\ \frac{p(y)-p(x)}{y-x} &= d_1, \\ p(y) &= b_0 + (y-x)d_1. \end{aligned}$$

Teszteljük a módszer hatékonyságát a naív eljárással összevetve, különösen olyan példákon, amelyre x és y értéke közel azonos.

4. Mint azt tárgyaltuk, a Horner-módszer optimális abban az értelemben, hogy egy polinom helyettesítési értékének kiszámításához a legkevesebb szorzás és összeadás műveletet használja. Azonban ezt úgy éri el, hogy az egymást követő összeadás és szorzás műveletek függnek a korábban végrehajtott műveletek eredményeitől. Így a módszer nehezen, vagy egyáltalán nem párhuzamosítható. Az Estrin-módszer párhuzamos gépekre alkalmas eljárás, amelynek ötlete a következő példán mutatható be. Jelölje

$$P_n(x) = C_0 + C_1x + C_2x^2 + C_3x^3 + \ldots + C_nx^n$$

egy n-változós polinomot. Az Estrin alakot írva a következő résszámításokat kapjuk:

$$\begin{aligned} P_3(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 \\ P_4(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + C_4x^4 \\ P_5(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + (C_4 + C_5x)x^4 \\ P_6(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + ((C_4 + C_5x) + C_6x^2)x^4 \\ P_7(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + ((C_4 + C_5x) + (C_6 + C_7x)x^2)x^4 \\ P_8(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + ((C_4 + C_5x) + (C_6 + C_7x)x^2)x^4 + C_8x^8 \\ P_9(x) &= (C_0 + C_1x) + (C_2 + C_3x)x^2 + ((C_4 + C_5x) + (C_6 + C_7x)x^2)x^4 + (C_8 + C_9x)x^8 \\ &\ldots \end{aligned}$$

amelyből láthatjuk, hogy az x^{2^n} értékét egyszer kell csak számolnunk, és azt megtarthatjuk addig, amíg szükség van rá.
Feladatunk, hogy implementáljuk a módszert, és vessük össze a végrehajtási idejét a Horner módszerrel szekvenciális nyelven és olyan környezetben is, ahol párhuzamos végrehajtás is megengedett.

5. A Ruffini-algoritmus egy $P(x)$ polinom $x - r$ taggal történő osztásának hatékony kiszámolására alkalmas. Keressük meg a leírását és készítsük el az implementációját.

5. fejezet

Polinomok gyökeiről

Azt mondjuk, hogy c a $p(x)$ polinom gyöke, ha $p(c) = 0$. Amennyiben $p(x)$ valós együtthatós polinom, amelynek gyöke az α komplex szám, akkor gyöke ennek $\overline{\alpha}$ konjugált is.

Egy fontos és egyszerű észrevétel, hogy az $p(x)$ polinom $(x-c)$ első fokú polinommal való osztásának maradéka egyenlő az $p(x)$ polinom $p(c)$ értékével. Legyen ugyanis

$$p(x) = (x - c)q(x) + r.$$

Ha most vesszük mindkét oldal értékét $x = c$-nél, akkor $p(c) = r$.

Ennek a ténynek egy következménye, hogy a c szám akkor és csak akkor gyöke az $p(x)$ polinomnak, ha $p(x)$ osztható $(x - c)$-vel. Ha $p(x)$ osztható valamely $ax + b$ első fokú polinommal, akkor osztható az $x - \frac{-b}{a}$-val is, vagyis egy $x - c$ alakú polinommal.

Egy polinom gyökeinek a meghatározása tehát ekvivalens lineáris osztóinak felkutatásával. A Horner-elrendezés segítségével meghatározhatjuk a

$$p(x) = (x - c)q(x) + r$$

alakot.

Példa. Legyen $p(x) = x^4 - 8x^3 + x^2 + 4x - 9$, és ezt osszuk el $x + 1$-gyel:

	1	-8	1	4	-9
-1	1	$-1 \cdot 1 + (-8) = -9$	$-1 \cdot (-9) + 1 = 10$	$-1 \cdot 10 + 4 = -6$	$-1 \cdot (-6) + (-9) = -3$

Ezért a hányados: $x^3 - 9x^2 + 10x - 6$, a maradék pedig $p(-1) = -3$.

Többszörös gyök. Amennyiben található olyan k szám, amelyre $p(x)$ maradék nélkül osztható az $(x-c)^k$-val, de nem osztható $(x-c)^{k+1}$-gyel, akkor a k szám a $p(x)$ polinom c gyökének *multiplicitása*.

Az algebra alaptétele a következőt állítja: minden legalább első fokú komplex együtthatós polinomnak van legalább egy (általában komplex) gyöke. Az alaptétel szerint egy polinom tehát elsőfokú polinomok szorzatára lehet bontani – ezzel majd később foglalkozunk.

Minden n-edfokú komplex együtthatós polinomnak n gyöke van, ha minden gyökét annyiszor számítjuk, amennyi a multiplicitása. Ez utóbbi tényre építve állíthatjuk, hogy bármely, n-nél nem magasabb fokú polinomot egyértelműen meghatároznak azon értékei melyeket az ismeretlenek bármely, n-nél több különböző értéke mellett vesz fel. Ha ismerjük egy n-ed fokú polinom értékét $n+1$ különböző pontban, akkor bármely más pontban is kiszámíthatjuk[1].

[1] Ez a tény adódik a Lagrange-féle interpolációs képletből.

Megoldóképletek. A másodfokú polinomegyenlet megoldásának módszerét már a régi görögök is ismerték. A harmad- és negyedfokú polinomok gyökeire csak a XVI.században találtak képleteket. 300 évig folytak sikertelen kísérletek a magasabb fokú polinomok gyökeinek meghatározására szolgáló képlet megkeresésére.

Majd 1832-ben Abel és Ruffini bebizonyította, hogy ilyen képletek nem léteznek $n \geq 5$ esetén *tetszőleges* n-ed fokú polinomra.

Fontos azonban látni, hogy az Abel-Ruffini tétel nem zárja ki azt a lehetőséget, hogy talán minden *konkrét* komplex együtthatós polinom gyökei mégis kifejezhetők lennének valahogyan az együtthatókból gyökvonások valamilyen kombinációi segítségével[2]

1830-as évek eleje: Galois[3] megmutatta, hogy milyen feltételek mellett oldható meg radikálokkal valamely adott polinomegyenlet. Ezen eredmény alapján kiderült, hogy minden $n \geq 5$-re megadható olyan n-ed fokú polinom, amely nem oldható meg radikálokkal.

5.1. Sturm tétel

Valós együtthatós $f(x)$ polinom valós gyökeinek számának meghatározására mutatunk egy eljárást, amely az euklideszi algoritmustól csupán a fellépő maradékok előjelében különbözik.

Legyen

$$\begin{aligned} f_0(x) &:= f(x) \\ f_1(x) &:= f'(x) \\ &\dots \\ f_{j-1}(x) &:= q_{j-1}(x)f_j(x) - f_{j+1}(x) \\ &\dots \\ f_{m-1}(x) &:= q_{m-1}(x)f_m(x) \end{aligned}$$

ahol az $f_{j+1}(x)$ maradékpolinom fokszáma kisebb, mint az $f_j(x)$ osztó polinom fokszáma. $f_m(x)$ az összes $f_j(x)$ polinom legnagyobb közös osztója.

A fent definiált polinomsorozat alapvető tulajdonságai:

5.1.1. Segédtétel. *Ha az $f(x)$ polinomnak a véges $[a,b]$ intervallumba eső gyökei mind egyszeresek, akkor minden $x \in [a,b]$-re $f_m(x) \neq 0$ (sőt, valójában $f_m(x)$ állandó előjelű az $[a,b]$-ben), ha valamely $1 \leq j \leq m$-re és $x \in [a,b]$-re $f_j(x) = 0$, akkor*

$$f_{j-1}(x)f_{j+1}(x) < 0,$$

ha valamely $x \in [a,b]$-re $f_0(x) = 0$, akkor

$$f_0'(x)f_1(x) > 0.$$

Legyen $f_0(x), f_1(x), \dots, f_m(x)$ valós együtthatós polinomoknak olyan sorozata, amelyben a fokszámok monoton csökkennek. Ezt *Sturm sorozatnak* nevezzük valamely $[a,b]$ intervallumon, ha a fenti lemmában szereplő *1-3* tulajdonságok teljesülnek és ezen felül még az is igaz, hogy

$$f(a)f(b) \neq 0.$$

Jelöljük $S(x)$-szel az $f_0(x), f_1(x), \dots, f_m(x)$ Sturm sorozatban előforduló *előjelváltások* számát. Jelváltás akkor van, ha pozitív szám után negatív következik, vagy fordítva.

[2] tehát, hogy minden konkrét polinom gyöke megadható radikálokkal (n-edik gyökökkel: $\sqrt[n]{\cdot}$))

[3] *„Don't cry, Alfred! I need all my courage to die at twenty."*

5.1.2. Tétel (Sturm)**.** *Legyen $f(x)$ valós együtthatós polinom. Ha*

$$f(x) := f_0(x), f_1(x), \ldots, f_m(x)$$

Sturm sorozat valamely $[a, b]$ intervallumon, akkor $f(x)$-nek (a, b)-ben pontosan $S(a) - S(b)$ számú gyöke van.

Példa. Legyen $f(x) = x^3 + 3x^2 - 1$, határozzuk meg a valós gyökeinek a számát. A polinom Sturm rendszere:

$$\begin{aligned} f(x) &= x^3 + 3x^2 - 1, \\ f_1(x) &= 3x^2 + 6x, \\ f_2(x) &= 2x + 1, \\ f_3(x) &= 1. \end{aligned}$$

Most állapítsuk meg ennek a rendszernek az előjelváltásainak számát a $(-\infty, \infty)$ intervallumon:

	$f(x)$	$f_1(x)$	$f_2(x)$	$f_3(x)$	váltások száma
-∞	$-$	$+$	$-$	$+$	3
∞	$+$	$+$	$+$	$+$	0

Kapjuk tehát, hogy az $f(x)$ polinomnak 3 valós gyöke van.

Valós gyökök szétválasztása. A Sturm tétel alkalmazható valós gyökök szétválasztására is a következőképpen:

1. Számítsuk ki $S(x)$ értékét különböző x-ekre.
2. Ha találunk olyan x_1, x_2 párt, amelyre $S(x_1) \neq S(x_2)$, akkor biztosak lehetünk abban, hogy x_1 és x_2 közrefog egy gyököt.
3. Felezzük meg az intervallumot, amelyben a gyök található: $S((x_1 + x_2)/2)$.
4. Menjünk vissza az 1. lépésre.

5.2. További korlátok polinomokra

Legyen adott

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_n \prod_{i=1}^{n} (x - \alpha_i)$$

n-ed fokú polinom. Vezessük be a következő jelöléseket:

- $H(f)$ (vagy $\|f\|_\infty$), amelyet magasságnak nevezünk és $\max_{i=0,\ldots,n} |a_i|$
- $\|f\|$ a kettes norma $\sqrt{\sum_{i=0}^{n} |a_i|^2}$
- $L(f)$ (vagy $\|f\|_1$), amelyet hosszúságnak nevezünk és $\sum_{i=0}^{n} |a_i|$

Ekkor adódik, hogy

$$H(f) \leq \|f\| \leq L(f) \leq (n+1)H(f).$$

Igaz továbbá, hogy

$$|a_n| \prod_{\alpha_i > 1} |\alpha_i| < \|f\|.$$

Ezek következményeképpen adódik továbbá, hogy

$$|a_{n-i}| \leq \binom{n}{i} |a_n| \prod_{\alpha_i > 1} |\alpha_i| \leq \binom{n}{i} \|f\|.$$

Gyökökre vonatkozó korlátok A fentiekben bemutattuk Sturm módszerét *valós* gyökök számának meghatározására. Néha azonban szükségünk lehet a valós és a komplex gyökök megkülönböztetésére. Tekintsük ehhez a következő példát.

Legyen W_{20} az a polinom, amelynek gyökei a $-1, -2, \ldots, -20$ számok. Azaz $W_{20} = (x+1)(x+2)...(x+20) = x^{20} + 210x^{19} + \ldots + 20!$. Tekintsük most a

$$W_{20}(x) + 2^{-23}x^{19}$$

polinomot. Arra számítunk, hogy ennek is húsz darab valós gyöke van, az eredeti W_{20}-hoz közel. A helyzet azonban az, hogy ennek csak 10 valós gyöke van, közelítőleg a $-1, -2, \ldots -7, -8.007, -8.917, -20.847$ értékekkel, valamint 5 pár komplex konjugált gyöke, amelyek közelítő értéke rendre a $-10.095 \pm 0.6435i, -11.794 \pm 1.652i, -13.992 \pm 2.519i, -16.731 \pm 2.813i, -19.502 \pm 1.940i$ értékek.

Legyen most $f = \sum_{i=0}^{n} a_i x^i$, valamint az $f(x)$ gyökei az $\alpha_1, \ldots, \alpha_n$ számok. Vezessük be az alábbi jelölést:

- $\text{rb}(f) = \max_{1 \leq i \leq n} |\alpha_i|$,

A következő korlátok teljesülnek:

- $\text{rb}(f) \leq 1 + \max(|a_i|)/|a_n|$.
- $\text{rb}(f) \leq 2 \max \left\{ \frac{|a_{n-1}|}{a_n}, \sqrt{\frac{|a_{n-2}|}{a_n}}, \ldots, \sqrt[n-1]{\frac{|a_1|}{a_n}}, \sqrt[n]{\frac{|a_0|}{a_n}} \right\}$

Amennyiben ezeket a korlátokat a $W_{20}(x)$ polinomra alkalmazzuk, akkor rendre a 1.38×10^{19} és 420 értékeket kapjuk, amely mutatja a második korlát erősségét, legalábbis az elsővel szemben.

5.3. Laguerre módszer

Ebben a szakaszban bemutatunk egy módszert, amely Edmond Laguerre nevéhez fűződik. A Newton-módszerrel ellentétben ez polinomok gyökeinek megkeresésére van specializálva.

A módszer tárgyalása előtt ismertetünk egy gyökkorlátos eredményt is, amely szintén Laguerre-től származik.

Valós gyökök további korlátai A módszere olyan $p(x) = \sum_{k=0}^{n} a_k x^k$ polinomokra alkalmazható, amelyeknek csak valós gyökei vannak. Ezek a gyökök a

$$-\frac{a_{n-1}}{na_n} \pm \frac{n-1}{na_n} \sqrt{a_{n-1}^2 - \frac{2n}{n-1} a_n a_{n-2}}$$

intervallumban vannak.
Például az $x^4 + 5x^3 + 5x^2 - 5x - 6$ polinomnak négy valós gyöke van: $-3, -2, -1$ és 1. A képlet a

$$-\frac{5}{4} \pm \frac{3}{4}\sqrt{\frac{35}{3}};$$

intervallumot adja, amely lebegőpontosan a $[-3.8117, 1.3117]$ intervallum – ebben az esetben tehát egész jó befoglalást kaptunk a gyökökre.

Gyökök megkeresése A módszer egyik tulajdonsága, hogy a kezdeti értéktől *függetlenül* majdnem mindig konvergál *valamelyik* gyökhöz, legyen az akár komplex szám. Az input tehát egy n-ed fokú polinom. Az eljárás a következő:

1. Válasszunk egy x_0 kezdeti értéket, és legyen $a = \infty$.
2. Adott lépésszámig vagy $a < \epsilon$ teljesüléséig hajtsuk végre a következőket:
   - Ha $p(x_k)$ nagyon pici, akkor az algoritmus véget ér, x_k egy közelítő érték.
   - Legyen $G = \frac{p'(x_k)}{p(x_k)}$ és $H = G^2 - \frac{p''(x_k)}{p(x_k)}$.
   - Legyen $a = \frac{n}{G \pm \sqrt{(n-1)(nH-G^2)}}$, ahol az előjelet úgy választjuk, hogy a nevező a lehető legnagyobb abszolút értékű legyen (a lebegőpontos számolásoknál ez fontos).
   - Végül legyen $x_{k+1} = x_k - a$.

5.4. Rezultánsok

A rezultánsok fogalmát a következő kérdés megválaszolására vezették be: mi a szükséges és elégséges feltétele annak, hogy az $f(x)$ és $g(x)$ polinomoknak van közös gyöke? Legyen

$$\begin{aligned} f(x) &= f_0 + f_1 x + \ldots + f_m x^m \\ g(x) &= g_0 + g_1 x + \ldots + g_n x^n, \end{aligned}$$

$\alpha_1, \ldots, \alpha_m$ az f polinom gyökei, $\beta_1, \ldots, \beta_n$ pedig a g polinom gyökei. Ekkor

$$\operatorname{res}(f, g) = f_m^n g_n^m \prod_{j=1}^{n} \prod_{i=1}^{m} (\alpha_i - \beta_j)$$

sorozatot az f és g polinomok *rezultánsának* nevezzük.

Ez a definíció matematikai értelemben teljesen rendben van, azonban praktikusan használhatatlan, mivel a gyökök ismeretét tételezi föl. A *Sylvester-mátrix*ot előállítjuk az együtthatókból:

$$\begin{pmatrix} f_m & f_{m-1} & \ldots & f_1 & f_0 & 0 & 0 & \ldots & 0 \\ 0 & f_m & f_{m-1} & \ldots & f_1 & f_0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & f_m & f_{m-1} & \ldots & f_1 & f_0 & 0 \\ 0 & \ldots & 0 & 0 & f_m & f_{m-1} & \ldots & f_1 & f_0 \\ g_n & g_{n-1} & \ldots & g_1 & g_0 & 0 & 0 & \ldots & 0 \\ 0 & g_n & g_{n-1} & \ldots & g_1 & g_0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & 0 & g_n & g_{n-1} & \ldots & g_1 & g_0 & 0 \\ 0 & \ldots & 0 & 0 & g_n & g_{n-1} & \ldots & g_1 & g_0 \end{pmatrix}$$

A Sylvester-mátrix determinánsát kell meghatározni, és megvizsgálni, hogy mikor 0 az értéke. Ez megtehető $\mathcal{O}((n+m)^3)$ lépésben például a jól ismert Gauss-eliminációval.

Egy másik lehetséges megoldást a következő algoritmus ad, ahol `LeadCoeff` a megadott polinom főegyütthatóját adja, a `deg` a polinom fokszámát, a `mod` művelet pedig a két polinom osztásával keletkező maradékot határozza meg.

```
Input: F, G polinomok
F[1] = F;
F[2] = G;
i = 2;
R[1] = 1;
while deg(F[i])>0{
  F[i+1] = F[i-1] mod F[i]
  di = deg(F[i]);
  di1 = deg(F[i-1]);
  R[i] = power(-1, di*di1)*power(LeadCoeff(F[i]), di1-di)*R[i-1]
  i++;
}
if (F[i] <> 0) return R[i]*power(LeadCoeff(F[i]), deg(F[i-1]))
else return 0;
```

Szimbolikus eset. A rezultánsok legfontosabb előnyei, hogy a bemeneti polinomok szimbolikus együtthatókat is tartalmazhatnak. Legyen például

$$\begin{aligned} f(x) &:= 2x^3 - \xi x^2 + x + 3 \\ g(x) &:= x^2 - 5x + 6. \end{aligned}$$

Ekkor

$$\begin{aligned} \mathrm{res}(f,g) &= \begin{vmatrix} 2 & -\xi & 1 & 3 & 0 \\ 0 & 2 & -\xi & 1 & 3 \\ 1 & -5 & 6 & 0 & 0 \\ 0 & 1 & -5 & 6 & 0 \\ 0 & 0 & 1 & -5 & 6 \end{vmatrix} \\ &= 36\xi^2 - 429\xi + 1260 = 0. \end{aligned}$$

A másodfokú egyenlet gyökei: $\xi = 20/3$ és $\xi = 21/4$.

5.5. Gröbner bázisok

Tekintsük a következő, m darab polinomból álló egyenletrendszert:

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ f_m(x_1, \ldots, x_n) &= 0 \end{aligned}$$

A Gröbner bázisok meghatározásával tulajdonképpen meghatározzuk a rendszer megoldását is, mert ezeknek ugyanazok a gyökei.

Példa. A következő 3 változós rendszer megoldását keressük:

$$\begin{aligned} f_1 &= xz - xy^2 - 4x^2 - \frac{1}{4} \\ f_2 &= y^2z + 2x + \frac{2}{1} \\ f_3 &= x^2z + y^2 + \frac{1}{2}x \end{aligned}$$

A rendszer Gröbner bázisa (a $x < y < z$ rendezéssel):

$$\begin{aligned} g_1 &= z + 64/65x^2 - 432/65x^3 + 168/65x^2 - 354/65x + 8/5 \\ g_2 &= y^2 - 8/13x^4 + 54/13x^3 - 8/13x^2 + 17/26x \\ g_3 &= x^5 - 27/4x^4 + 2x^3 - 21/16x^2 + x + 5/32 \end{aligned}$$

Érdekes módon ezt az alakot 'trianguláris' alaknak nevezzük, a Gauss-elmináció eredményéhez hasonlóan. Az analógia ott van, hogy a Gauss elimináció valójában a Gröbner bázisok módszerének egy speciális esete, amikor minden egyenlet lineáris.

A fenti alakból x meghatározható a 3. egyenletből, majd az behelyettesítjük a 2. egyenletbe megkapjuk y értékét, stb. A közelítő megoldás egyébként

$$(-0.128475, 0.321145, -2.356718)$$

Megjegyezzük, hogy egyváltozós polinomegyenlet megoldása lényegesen egyszerűbb, mint egy polinom rendszer megoldásának megkeresése. Itt most arról van szó, hogy úgy oldjuk meg a rendszert, hogy azt átalakítjuk, majd végül egyváltozós polinomok sorozatát kell megoldanunk.

A Gröbner bázisok kiszámítására szolgáló algoritmus, sőt, még a kapcsolódó alapfogalmak is túlmutatnak a jegyzet keretein, így azokat itt nem ismertetjük.

5.6. Feladatok

1. Implementáljuk a 5.2 szakaszban ismertetett korlátokat, és teszteljük az erősségüket véletlen együtthatós polinomokra.

2. A Horner-módszer a Newton-módszerrel kombinálva alkalmas polinomok össze valós gyökének megkeresésére. Legyen adott a $p_n(x)$ n-ed fokú polinom, amelynek gyökeire igaz, hogy $z_n < z_{n-1} < \cdots < z_1$. Legyen x_0 kezdeti érték úgy, hogy $x_0 > z_1$. Hajtsuk végre a következő iterációs lépéseket:

   (a) Használjuk a Newton-módszert a z_1 megkeresésére.

   (b) Használjuk a Horner-módszer a $(x - z_1)$ taggal történő leosztásra, amellyel megkapjuk a p_{n-1}-et. Térjünk vissza az 1. lépésre, ahol használjuk a p_{n-1} polinomot és a z_1-et, mint kezdeti értéket.

   Implementáljuk és teszteljük ezt az eljárást.

3. Végezzünk kísérleteket véletlen együtthatós polinomok gyökeinek eloszlásáról. Használjunk ehhez többfajta véletlenszám generátort különböző paraméterekkel.

4. Az $x^2 - S = 0$ egyenlet gyöke az $\sqrt{S}$. A négyzetgyök reciproka gyakran előfordul különféle numerikus módszerekben, ezért érdekes lehet a kiszámítására gyors eljárást kidolgozni. Természetesen csak közelítő eljárásról lehet szó. A módszer egyik legendás implementációja a Quake III nevű játék C forráskódjában található, amelyet *majdnem szó szerint* idézünk itt:

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;         // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 ); // what the f&*k?
  y  = * ( float * ) &i;
  y  = y*(threehalfs-(x2 * y * y)); // 1st iteration
//y  = y*(threehalfs-(x2 * y * y)); // 2nd iteration, this can be removed
  return y;
}
```

A kódban természetesen a `0x5F3759DF` konstans kelti fel elsősorban az olvasó figyelmét. Feladatunk, hogy derítsük ki, vajon mi szerepe van pont ennek az értéknek egy egyébként általános módszerben?

5. Bizonyítsuk be, hogy ha $p(x) \in \mathbb{Z}_p$ n-ed fokú polinom, akkor *legfeljebb* n gyöke van.

6. fejezet

Prímtesztelés

Ebben a fejezetben olyan módszerekről lesz szó, amelyek választ adnak arra a döntési kérdésre, hogy egy egész szám prímszám-e. Néhány valószínűségi algoritmust fogunk megismerni, valamint megemlítjük a 'PRIMES is in P' cikkről is. Láttuk a korábbi fejezetekben, hogy a prímeket használjuk a moduláris algoritmusoknál is.

6.1. Alapfogalmak

Egy N egész szám prímszám, ha $ab = N$ esetén $a = 1$ vagy $b = 1$.

Az *aritmetika alaptétele* szerint minden pozitív egész szám felírható prímszámok szorzataként.

Továbbá már Euklidész is tudta, hogy végtelen sok prímszám van. Bizonyításának egy picit módosított változata a következő. Tegyük fel, hogy véges sok prímszám van: $p_1 < p_2 < \ldots < p_r$. Legyen $N = p_1 \cdot p_2 \cdot \ldots \cdot p_r$. Az $N - 1$ egész szám, mivel szintén prímszámok szorzatára bontható, ezért szükségképpen van közös prímosztója az N-nel, ami legyen p_i. Ekkor p_i osztja a különbségüket, 1-et, ami lehetetlen.

6.2. Szita módszer

Eratoszthenész alapján egy lehetséges módszer N prímtesztelése a következő: kezdjük el osztogatni $3, \ldots, \sqrt{N}$-nel és nézzük meg, hogy van-e maradék.

Az algoritmus pszeudokódja a következő [8]:

```
Bemenet: egy n pozitív egész szám.
Kimenet: 1, ha az n szám prím, 0, ha összetett.

if (n == 2) return 1,
if (n mod 2 == 0) return 0,
t = 3; s = 4;
while (s <= n) {
   if (n mod t == 0) return 0;
   else {
     t = t + 2;
     s = s + 2 * t - 1;
     print(t, s);
   }
}
return 1;
```

6.3. Fermat kis tétele

6.3.1. Tétel (Fermat, 1640)**.** *Ha p prím és $a \in \mathbb{N}$ ehhez relatív prím, akkor*

$$a^{p-1} \equiv 1 \mod p.$$

Az a tehát nem osztható p-vel. A tétellel tehát azt tudjuk megmondani, hogy egy egész szám **nem** prímszám. A fenti fölírásban ez p.

A tétel egy másik változata a következő: bármely p prímszámra teljesül bármely $a \in \mathbb{N}$ egész szám esetén, hogy

$$a^p \equiv a \mod p.$$

Érdekességképpen megjegyezzük, hogy Fermat egyébként nem adott bizonyítást a tételre, csak egy levélben írta azt le. Leibniz bizonyította, de annak pontos dátum nem ismert.

6.3.1. Álprímek

Ha N összetett és

$$a^{N-1} \equiv 1 \mod N$$

valamely a-ra, akkor azt mondjuk, hogy N egy a-alapú **álprím**.

Sarrus 1820-ban az alábbi eredményre jutott: $N = 341$ az $a = 2$ alapra álprím (hiszen $341 = 11 \cdot 31$).

6.3.2. Algoritmus a Fermat kis tétele alapján

A Fermat kis tétele alapján a következő, valószínűségi algoritmus adódik:

Bemenet: N egész

Kimenet: 'összetett' vagy 'valószínűleg prím'

1) Legyen $2 \leq a \leq N - 2$ véletlen szám (egyenletes eloszlással választva).

2) Számítsuk ki

$$b = a^{N-1} \text{ rem } N,$$

ehhez használjuk az ismételt hatványozás algoritmust a 2.4 szakaszból. Itt a rem művelet az 'osztás maradéka'.

3) Ha $b \neq 1$, akkor a válasz '*összetett*', különben '*valószínűleg prím*'.

Tulajdonságok. Az algoritmusra a következő állítások teljesülnek:

- Ha a és N nem relatív prímek, akkor b és N sem azok, ezért az algoritmus ekkor korrekt választ ad.
  - Az 'a és N relatív prímek?' kérdés eldöntésére az euklidészi algoritmust használjuk, végrehajtva azt az 1) és 2) lépés között.
- Ha $\text{lnko}(a, N) = 1$, akkor a 'valószínűleg prím' vagy helyes vagy helytelen válasz.

Példa. Teszteljük le 5 próbálkozással, hogy a 97 prímszám-e. Tegyük fel, hogy a következő eredményeket kaptuk:

a	b	iteráció	jelentés
41	1	1	$41^{96} \equiv 1 \pmod{97}$
32	1	2	$32^{96} \equiv 1 \pmod{97}$
94	1	3	$94^{96} \equiv 1 \pmod{97}$
17	1	4	$17^{96} \equiv 1 \pmod{97}$
73	1	5	$73^{96} \equiv 1 \pmod{97}$

Az algoritmus az összes véletlenszerűen generált a érték esetében igaznak találja az $a \equiv 1 \pmod{n}$ kifejezést, ami azt jelzi, hogy a 97 prímszám, tehát 1 visszatérítési értékkel leáll.

Példa. Annak szemléletetésére, hogy az $a^{p-1} \equiv 1 \pmod{p}$ kiszámításához, ahol p egy nagy méretű prímszám nem kell kiszámolnunk az a^{p-1} értékét, hanem annak csak p-vel osztási maradékát, elevenítsük fel az ismételt négyzetre emelés módszerét.
Tegyük fel, hogy $2^{154} \mod 155$ értékét akarjuk meghatározni. A 154 bináris alakja: 10011010. A hatványok tehát bináris alakban rendre: $1, 10, 100, 1001, 10011, 100110, 1001101$, és 10011010 vagy 10-es számrendszerben: $1, 2, 4, 9, 19, 38, 77$, és 154. A következő számításokat kell elvégezünk:

$$\begin{aligned}
2^1 &\equiv 2 \cdot 1^2 \equiv 2 \mod 155 \\
2^2 &\equiv 2^2 \equiv 4 \mod 155 \\
2^4 &\equiv 4^2 \equiv 16 \mod 155 \\
2^9 &\equiv 2 \cdot 16^2 \equiv 47 \mod 155 \\
2^{19} &\equiv 2 \cdot 47^2 \equiv 78 \mod 155 \\
2^{38} &\equiv 78^2 \equiv 39 \mod 155 \\
2^{77} &\equiv 2 \cdot 39^2 \equiv 97 \mod 155 \\
2^{154} &\equiv 97^2 \equiv 109 \mod 155.
\end{aligned}$$

Először is az eredményből leolvashatjuk, hogy 155 nem prím szám, hiszen ha az lenne, akkor az 1 végeredményt kaptuk volna. Amennyiben a p számnak n bináris számjegye van, az $a^{p-1} \mod p$ kiszámítása $2n$ szorzással valamint n darab maradék művelettel kiszámítható. Ezek műveletigénye $\mathcal{O}(n^2)$, ezért a teljes műveletigény $\mathcal{O}(n^3)$ lesz.

6.3.3. Carmichael számok

Ha az

$$a^{N-1} \equiv 1 \mod N$$

kongruencia igaz minden N-hez relatív prím a-ra, akkor N **univerzális álprím**. Ezek a Carmichael számok. Végtelen sok van belőle, de sokkal ritkábbak, mint a prímszámok. A legkisebb ilyen: 561. Ezek tehát olyan inputok, amelyekre a Fermat teszt rossz választ ad, ezt minden esetben figyelembe kell vennünk az algoritmus használatakor.

Szerencsére a Carmichael számok karakterizációja megadható, amelyet a következő tételben foglaltunk össze.

6.3.2. Tétel. *N akkor és csak akkor Carmichael, ha négyzetmentes és $p-1$ osztja $N-1$-et, ahol p az N prímfaktora, továbbá: N páratlan és legalább 3 prímfaktora van.*

Az 561 utáni hat Carmichael szám a következő (az $a \mid b$ jelentése: a osztója b-nek):

$$
\begin{array}{lllll}
1105 = 5 \cdot 13 \cdot 17 & (4 \mid 1104; & 12 \mid 1104; & 16 \mid 1104) \\
1729 = 7 \cdot 13 \cdot 19 & (6 \mid 1728; & 12 \mid 1728; & 18 \mid 1728) \\
2465 = 5 \cdot 17 \cdot 29 & (4 \mid 2464; & 16 \mid 2464; & 28 \mid 2464) \\
2821 = 7 \cdot 13 \cdot 31 & (6 \mid 2820; & 12 \mid 2820; & 30 \mid 2820) \\
6601 = 7 \cdot 23 \cdot 41 & (6 \mid 6600; & 22 \mid 6600; & 40 \mid 6600) \\
8911 = 7 \cdot 19 \cdot 67 & (6 \mid 8910; & 18 \mid 8910; & 66 \mid 8910)
\end{array}
$$

Nagyobb számokra egyre kevesebb van belőlük: 20.318.200 darab van 1 és 10^{21} között. Ez 50 milliárd számból 1.

További információt a Carmichael számokról a `https://oeis.org/A002997` linken találhatunk az OEIS (Online Encyclopedia of Integer Sequences) oldalán.

6.4. Miller-Rabin teszt

Azokat az egész számokat, amelyek kielégítenek olyan prímteszteket, amelynek minden prímszám eleget tesz, de a legtöbb összetett szám nem, valószínű prímnek nevezzük. A valószínű prímek egy része összetett szám, ezek a 6.3.1. szakaszban tárgyalt álprímek. Az angol nyelvű szakirodalomban a PRP jelölést használják (*probable prime*).

Az ebben a szakaszban ismertetett algoritmus kihasználja, hogy ha p prímszám, akkor minden olyan a szám esetén, amelyre $\text{lnko}(a, p) = 1$ és $p - 1 = 2^s r$, ahol r páratlan szám, fennáll a következő összefüggések egyike:

- $a^r \equiv 1 \pmod{p}$, vagy
- létezik olyan $j, 0 \leq j \leq s - 1$ úgy, hogy $a^{2^j r} \equiv p - 1 \pmod{p}$.

Legyen $N - 1 = 2^s t$, ahol t páratlan és $s \geq 1$, hiszen a páros számokról könnyű eldönteni, hogy prímszámok-e. Válasszunk egy a véletlen számot, majd számítsuk ki a következőket:

1. $u_0 \equiv a^t \mod N$
2. $u_{i+1} \equiv u_i^2 \mod N$, ahol $i = 1, 2, \ldots s$.
3. Válasz: N *valószínűleg prím*, ha $u_0 = 1$, vagy valamely i-re $(0 \leq i < s)$ az $u_i = -1$

Megjegyezzük, hogy $u_s \equiv a^{2^s t} \equiv a^{N-1}$ ezért ennek értéke $1 \mod N$ ha N egy prímszám. Ez a feltétel az alapja az elfogadásnak, mert ha $u_i \equiv -1$, akkor $u_j \equiv 1$ minden $i + 1 \leq j \leq s$ indexre. Általában a tesztet k-szor egymástól függetlenül elvégezzük, hogy biztosabb választ kapjunk.

Tegyük fel, hogy N prímszám. Akkor mindig azt a választ adjuk, hogy '*valószínűleg prím*'. Mivel tudjuk, hogy $u_s \equiv 1 \mod N$, ezért vagy az van, hogy $u_0 \equiv 1 \mod N$ vagy pedig van olyan utolsó u_i, amelyre $u_i \equiv 1 \mod N$ NEM teljesül és $u_{i+1} \equiv u_i^2 \equiv 1 \mod N$. De ha N prím, akkor az $x^2 \equiv 1 \mod N$ egyenlet megoldása az $x \equiv \pm 1$, ezért szükségképpen $u_i \equiv -1 \mod N$, ami azt jelenti, hogy N-et *valószínűleg prím*-nek deklaráltuk.

Amennyiben N összetett szám, akkor az a véletlenszámok halmazának felével ezt az eredményt kapjuk. Ehhez tegyük fel, hogy $N = p_1 p_2 \ldots p_r$, és vegyünk egy $a \mod N$ értéket véletlenszerűen. A kínai maradéktétel szerint ez ugyanaz, mintha az $a_1 \mod p_1, a_2 \mod p_2, \ldots, a_r \mod p_r$ értékeket

választottuk volna véletlenszerűen. Ahhoz, hogy az $u_i \mod N$ értéke ne legyen egyenlő -1-gyel, annak kell teljesülnie, hogy

$$\begin{aligned} u_i &\equiv -1 \pmod{p_1} \\ u_i &\equiv -1 \pmod{p_2} \\ &\vdots \\ u_i &\equiv -1 \pmod{p_r} \end{aligned}$$

Ha N egy Carmichael szám, akkor az algoritmus legalább 1/2 valószínűséggel meg is adja N osztóját.

Ha a hiba valószínűségét ε alá akarjuk csökkenteni, akkor $\log_2 \varepsilon^{-1}$-szer kell végrehajtani. Ha mindig azt a választ kapjuk, hogy '*valószínűleg prím*', akkor N prím.

Példa. Teszteljük le, hogy 97 valószínű prímszám-e. Ehhez a következő lépéseket kell elvégezni:

1. 1: Keressük meg d és s értékeket, amelyre $96 = d \cdot 2^s$ alakú, ahol d páratlan szám.
   - Kezdjük $s = 0$ értékkel, ekkor $d = 96$.
   - Növeljük s értékét. Ekkor $d = 3$ és $s = 5$ ezért $96 = 3 \cdot 2^5$
2. Válasszuk a értéket ($1 < a < 97 - 1$). Most az $a = 2$ választással dolgozunk.
3. Számítsuk ki $a^d \pmod n$ értékét, azaz $2^3 \pmod{97}$ értékét. Mivel ez nem kongruens $1 \pmod{97}$ folytatjuk a tesztet a következő feltétel ellenőrzésével.
4. Számítsuk ki $2^{3 \cdot 2^r} \pmod{97}$ értékét ($0 \leq r < s$). Amennyiben ez kongruens $96 \pmod{97}$, akkor 97 valószínű prím. Egyébként 97 biztosan összetett.
   - $r = 0:\ 2^3 \equiv 8 \pmod{97}$
   - $r = 1:\ 2^6 \equiv 64 \pmod{97}$
   - $r = 2:\ 2^{12} \equiv 22 \pmod{97}$
   - $r = 3:\ 2^{24} \equiv 96 \pmod{97}$

Kaptuk, hogy 97 valószínű prím.

6.5. Mersenne prímek

6.5.1. Tétel. *Ha $2^p - 1$ prím, akkor p prím.*

A bizonyítást könnyű meggondolni, ugyanis $2^{uv} - 1$ osztható $2^u - 1$-gyel. Ez abból adódik, hogy ha p összetett, akkor $2^p - 1 = 2^{uv} - 1 = (2^u)^v - 1^v = (2^u - 1)(\ldots)$.

Mersenne 1644-ben a következőt állította: $2^p - 1$ prímszám, ha

$$p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$$

és minden más $p < 257$-re összetett.

Ez az állítás sok matematikus érdeklődését keltette föl, így például Euler 1772-ben megmutatta, hogy $2^{31} - 1$ prím, ezzel megcáfolta Mersenne állítását. Lucas száz évvel később bebizonyította, hogy $2^{127} - 1$ prím, de $2^{67} - 1$ nem az. Powers 1911-ben bővítette a listát: $2^{89} - 1$ prím, $2^{107} - 1$ prím, amellyel tehát újabb Mersenne prímeket talált. Ezután Kraichik 1922-ben megmutatta, hogy $2^{257} - 1$ összetett, amellyel tehát Mersenne utolsó számáról mutatta meg az eredeti sejtés hamisságát.

Bár a számítógépek használatával sokkal hatékonyabbak lehetünk, azóta sem találtak *sokkal* több Mersenne prímet.

- 2016.január: 49. Mersenne prím: $2^{74,207,281} - 1$,
- 2017. december 26: 50. Mersenne prím: $2^{77,232,917} - 1$, amely a jegyzet írásának időpontjában egyben a legnagyobb ismert prímszám, nagyjából 23 millió számjeggyel.

Megjegyezzük, hogy 1997 óta az összes Mersenne prímet a GIMPS projekt (Great Internet Mersenne Prime Search) nevű, teljesen elosztott számítógépes rendszerrel keresik.

6.6. AKS algoritmus

Az AKS algoritmus a szerzőkről, három indiai matematikusról (Manindra Agrawal, Neeraj Kayal és Nitin Saxena) lett elnevezve. A 2002-es publikációjuk címe elárulja a lényeget, 'PRIME is in P'. Ez az első olyan eljárás, amely determinisztikus, futási ideje polinomiális és nem alapszik semmilyen hipotézisre. A következő, egyébként régóta ismert azonosságra épül.

6.6.1. Tétel. *Az n egész szám akkor és csakis akkor prím, ha lnko$(a, n) = 1$ és*

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

ahol a maradékos osztást a polinom együtthatóin kell elvégezni.

Ez lényegében a Fermat kis tételének általánosítása $(x = 0)$.

Példa. Igazoljuk, hogy az 5 prímszám. Ekkor

$$(x + 1)^5 = x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1 \equiv (x^5 + 1) \pmod{5},$$

a polinom minden együtthatójának 5-tel való osztási maradéka 0, kivéve az első és utolsó együtthatókat.

Az algoritmus részletes tárgyalása túlmutat a jegyzet keretein, azonban a kapcsolódó Wikipédia oldalon `http://en.wikipedia.org/wiki/AKS_primality_test` találhatunk egy szép illusztrációt is az algoritmus futására.

6.7. Nagyméretű prímszámok generálása

Valószínűleg prímszámok generálása Az algoritmus bemenete egy k pozitív egész szám és t egy biztonsági paraméter, kimenete pedig egy k hosszúságú, 10-es alapú számrendszerbeli valószínűleg prímszám. Az algoritmus végrehajtásához felhasználjuk a Miller-Rabin teszt eljárást.

```
p = rand(10^(k-1), 10^k)();
if (p mod 2 = 0) p = p + 1;
while (true){
  if (miller_rabin(p, t) = 1) return p;
  p = p + 2;
}
```

Erős prímszámok generálása Egy p prímszámot *erős prímnek* nevezünk, ha léteznek az r, s, q egész számok a következő tulajdonságokkal:

- $p - 1$-nek van egy nagy prímtényezője, jelöljük ezt r-rel,
- $p + 1$-nek van egy nagy prímtényezője, jelöljük ezt s-sel,

- és $r-1$ -nek van egy nagy prímtényezője, jelöljük ezt q-val.

A következő eljárás használja az imént ismertetett prímszám generátort, amelyre `random_prim` függvényként hivatkozunk a kódban. A bemenet itt is egy k pozitív egész szám és t biztonsági paraméter, a kimenet pedig egy k hosszúságú, 10-es számrendszerbeli erős prímszám.

```
k1 = floor(k/2);  // alsó egészrész
q = random_prim(k1, t);
i = 1;
while (true){
  r = 2 * i * q + 1;
  if (miller_rabin(r, t) == 1) break;
  i++;
}
s = random_prim(k1, t);
mh = mod(power(s,r-2), r); // moduláris hatványozással számítva!
p0 = 2 * s * mh - 1;
j = 1;
while (true){
  p = p0 + 2 * j * r * s;
  if (miller_rabin(p, t)==1) return p;
  j++;
}
```

6.8. Feladatok

1. A *levágható prím* olyan prímszám, amelynek a számjegyeit levágva rendre prímszámokat kapunk. Például a 9137 prímszám egy balról levágható prím, hiszen 137, 37 és 7 mind prímszámok. Ugyanakkor a 7393 prímszám egy jobbról levágható prím, mert 739, 73, 7 is mind prímszámok. Összesen 4260 balról levágható- és 83 jobbról levágható prímszámot ismerünk. Feladatunk, hogy írjunk egy eljárást levágható prímek keresésére.

2. Egy prímszámot *mírp*-nek nevezzük, ha visszafelé olvasva egy másik prímszámot ad. Írjunk eljárást, amely mírpeket keres.

3. Jelölje p_n az n-edik prímszámot. Ekkor $p_n\#$ *primoriálist* az első n darab prímszám szorzataként definiáljuk:

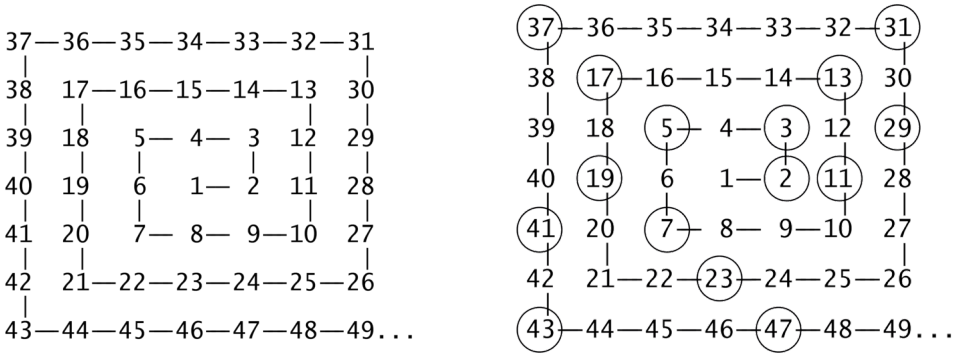
$$p_n\# = \prod_{i=1}^{n} p_i,$$

   ahol tehát p_i az i-edik prímszám.
   A *primoriálisprím* egy olyan prímszám, amely $p_n\# \pm 1$ alakban felírható. Az eddig ismert legnagyobb primoriálprím a $1098133\# - 1$ ($n = 85586$), amelynek 476.311 számjegye van. Írjunk eljárást, amely primoriálisprímeket keres.

4. A *prímhézag* két egymás utáni prímszám különbsége, melynek jelölése: $g_n = p_{n+1} - p_n$. Bizonyítsuk be, hogy

$$p_{n+1} = 2 + \sum_{i=1}^{n} g_i.$$

5. Az Ulam spirál a prímszámok furcsa mintázatát mutató megjelenítés. Az alábbi ábra bal oldalán az egész számok spriális elrendezését látjuk, míg a jobb oldalon pedig ebben a sorozatban a prímszámokat tartalmazza[1]:



Készítsünk számítógépes implementációt az Ulam spriál nagyfelbontású kirajzolására.

[1] forrás: `https://hu.wikipedia.org/wiki/Ulam-spirÃąl`

7. fejezet

Prímfaktorizálás

Ebben a fejezetben az alábbi fontos tételből indulunk ki.

7.0.1. Tétel (Számelmélet alaptétele)**.** *Minden pozitív egész n egyértelműen írható*

$$n = p_1 \cdot p_2 \cdot \ldots \cdot p_t, \qquad p_1 \le p_2 \le \ldots \le p_t,$$

alakban, ahol mindegyik p_k prímszám.

Ez alapján a következő feladatokat vizsgáljuk:

(i) meghatározni p_t-t, a legnagyobb prímtényezőt,

(ii) meghatározni t értékét, azaz a prímtényezők számát,

(iii) megtalálni a fenti fölbontást.

7.1. Legnagyobb prímtényező

Foglalkozzunk először röviden az első kérdéssel: mekkora szokott p_t lenni?

Karl Dickman, 1930: mi annak a valószínűsége, hogy egy 1 és x közötti véletlenszám legnagyobb prímosztója $\le x^\alpha$? Azt találta, hogy $x \to \infty$ esetén ez a valószínűség az $F(\alpha)$ határeloszláshoz tart, ahol

$$F(\alpha) = \int_0^\alpha F\left(\frac{1}{1-t}\right)\frac{dt}{t}, \ \ (0 \le \alpha \le 1); \qquad F(\alpha) = 1 \ \text{ ha } \ \alpha > 1.$$

Ha $1/2 \le \alpha \le 1$, akkor ez a formula az

$$F(\alpha) = 1 - \int_\alpha^1 F\left(\frac{1}{1-t}\right)\frac{dt}{t} = 1 - \int_\alpha^1 \frac{dt}{t} = 1 + \ln\alpha$$

alakra egyszerűsödik. Így például $\alpha = 1/2$-re kapjuk, hogy $1 - F(1/2) = \ln 2$, vagyis durván 69 százalék annak a valószínűsége, hogy egy x-nél kisebb véletlen egész szám osztható egy $\sqrt{x}$-nél nagyobb prímmel! Illusztrációként a `https://en.wikipedia.org/wiki/Table_of_prime_factors#901_to_1000` wikipedia oldalt (amely a 900 és 1000 közötti számok prímfelbontását tartalmazza) átböngészve azt találjuk, hogy csak ezen 100 szám között nagyon sok van olyan, aminek a legnagyobb prímtényezője $\sqrt{1000} = 31.623$ értéknél nagyobb.

7.2. Prímtényezők száma

Intenzíven vizsgálták t értékét is, az összes prímosztók számát.

Nyilván $1 \leq t \leq \log_2 n$, ezeket a korlátokat azonban csak ritkán éri el. Megmutatható, hogy ha n-et véletlenül választjuk 1 és x között, akkor bármely rögzített c-re, annak a valószínűsége, hogy $t \leq \ln\ln x + c\sqrt{\ln\ln x}$, a

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{c} e^{-u^2/2} du$$

értékhez tart, ha $x \to \infty$. Más szavakkal: t lényegében normális eloszlású.

7.3. Prímfelbontás

Jelen ismereteink szerint ezek nehezebb feladatok, mint a prímteszt. Tehát az egy megoldatlan probléma, hogy vajon létezik-e polinomiális idejű algoritmus a prímfelbontásra

7.3.1. Próbálgatásos osztás

Osszuk és bontsuk tényezőkre: vagyis osszuk el n-et a

$$p = 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, \ldots$$

számokkal (próbaosztókkal) mindaddig, amíg eljutunk az

$$n \mod p = 0$$

esetig. Ekkor p az n legkisebb prímosztója, és az eljárást folytatjuk az $n = n/p$-vel.

A fenti sorozatban az első három tag után felváltva mindig 2-vel illetve 4-gyel növelünk.

Ha azt találjuk, hogy

$$n \mod p \neq 0 \quad \text{és} \quad \lfloor n/p \rfloor \leq p \tag{7.1}$$

akkor n prímszám. Ennek belátásához tegyük fel, hogy n összetett, mégis teljesül (7.1). Akkor

$$n \geq p_1^2,$$

ahol p_1 az n első prímtényezője. Ugyanakkor $p_1 > p$ miatt[1]

$$\begin{aligned} p_1^2 \geq (p+1)^2 > p(p+1) &> p^2 + (n \mod p) \\ &\geq \lfloor n/p \rfloor p + (n \mod p) = n. \end{aligned}$$

Az első egyenlőtlenség azért teljesül, mert $p_1 - p \geq 1$. A második egyenlőtlenség triviális. A harmadik a $p > n \mod p$ miatt teljesül. A negyedik a $p \geq \lfloor n/p \rfloor$ tulajdonságon alapszik, ami a (7.1) feltételek egyike, amikről feltettük, hogy teljesülnek. Végül, $n = kp + r$ alakú, ahol $r = n \mod p$ és $\lfloor n/p \rfloor p = kp$.

[1] ott még nem tart az algoritmus a p értékekkel, hogy elérje a p_1 értékét, hiszen az (7.1) feltétel szerint a p NEM osztja n-et, viszont tudjuk, hogy p_1 osztja n-et

Példa. Legyen $n = 25852$. Azonnal adódik, hogy $n = 2 \cdot 12926$, így $p_1 = 2$. Ezután $12926 = 2 \cdot 6463$, tehát $p_2 = 2$. Most $n = 6463$ nem osztható $2, 3, 5, \ldots, 19$ egyikével sem, viszont $n = 23 \cdot 281$, tehát $p_3 = 23$. Végül $281 = 12 \cdot 23 + 5$ és $12 \leq 23$, vagyis $p_4 = 281$.

Az $n = 25852$ prímtényezőinek meghatározásához összesen 12 osztási műveletre volt szükségünk Ha a 25849 számot próbáltuk volna fölbontani (ami egyébként prímszám), akkor legalább 38 osztást kellett volna elvégeznünk. Ez talán jól illusztrálja, hogy az algoritmus futási ideje kb. $\max(p_{t-1}, \sqrt{p_t})$-vel arányos. Ha n kicsi, akkor érdemesebb táblázatból kinézni. Például, ha n kisebb, mint egymillió, akkor csak az ezer alatti 168 darab prímet kell használjunk (ugyanis elég a próbaosztást $\sqrt{n}$-ig elvégezni).

Beiktathatunk egy prímtesztet is (lásd előző fejezet), ez általában gyorsít. Ilyenkor a futási idő arányos p_{t-1}-gyel, ami tehát n második legnagyobb prímtényezője.

A Próbálgatásos osztás algoritmus általában talál néhány apró prímtényezőt, majd hosszúra nyúló kutatásba kezd a fennmaradó nagyok után. Ha $n = (2^{521} - 1) \cdot (2^{607} - 1)$ és faktorizálandó a $10n$, akkor a $2 \cdot 5 \cdot n$ felbontást hamar megtalálja, de utána nagyon sokáig eredménytelenül keresgél.

Prímek száma adott korlátig. Legyen $\pi(x)$ az x-ig terjedő prímek száma tehát $\pi(2) = 1, \pi(10) = 4$, stb.

Charles de la Vellée Poussin 1899-ben a következőt állította:

$$\pi(x) = \int_2^x \frac{dt}{\ln t} + \mathcal{O}(xe^{-A\sqrt{\log x}})$$

alkalmas $A > 0$-val.

Bernhard Riemann sejtése 1859-ből:

$$\pi(x) = L(x) - \frac{1}{2}L(x^{1/2}) - \frac{1}{3}L(x^{3/2}) + \ldots,$$

ahol $L(x) = \int_2^x \frac{1}{\ln t} dt$. Ezt a sejtést Littlewood 1914-ben megcáfolta: megmutatta, hogy alkalmas C pozitív konstanssal

$$\pi(x) > L(x) + C\sqrt{x} \cdot \log\log\log x / \log x$$

teljesül végtelen sok x-re.

Ezek az eredmények mind azt jelzik, hogy a prímszámok rejtélyes valamik, eloszlásuk megismeréséhez nagyon mély matematikai elméletekre van szükség.

7.3.2. Pollard-féle Monte Carlo módszer

Nézzünk akkor most egy módszert, amely valószínűségi alapon keresi egy összetett szám prímtényezős felbontását.

Az algoritmus az alábbi észrevételen alapszik. Ahhoz, hogy két szám, x és y, kongruens modulo p legyen $0,5$ valószínűséggel,

$$1.177 \cdot \sqrt{p}$$

számot kell véletlenszerűen kiválasztani[2].

Legyen $x \equiv y \pmod{p}$. Ha p az n egész prímfaktora, akkor

$$p \leq \text{lnko}(x - y, n) \leq n$$

mivel p osztója az $(x - y)$-nak[3] és n-nek egyaránt. Egy valószínűségi módszert mutatunk.

[2]Ez ugyanaz, mint a születésnap probléma: mekkora házibulit kell rendeznünk ahhoz, hogy legalább $0,5$ valószínűséggel legyen két azonos születésnapú vendégünk? Meglepő válasz: 23 ember elég.

[3]hiszen a feltevésből következik, hogy $x - y \equiv 0 \pmod{p}$

Feltételezve, hogy a p prímszám osztója n-nek, ezért az $x_0, x_1, x_2, \ldots$ sorozatban, ahol $x_i = f(x_{i-1})$ és $f : \mathbb{Z}_p \to \mathbb{Z}_p$, ismétlődéseket fogunk találni, azaz bizonyos k értékre

$$x_{2k} \equiv x_k \pmod{p}$$

Természetesen p-t nem ismerjük, ezért a modulo n szerinti maradékos osztást vizsgáljuk, Tehát most $f : \mathbb{Z}_n \to \mathbb{Z}_n$ alakú, hiszen csak n-et ismerjük, csak ezt a függvényt tudjuk legyártani. Ilyenkor is kapunk ismétlődést az f által generált sorozatban, és bár ezek nem ugyanazok a számok, mint amit a mod p-vel számolt függvénnyel kapnánk, de attól még hívjuk őket x_k-nak és x_{2k}-nak. Azt biztosan tudjuk, hogy $x_k - x_{2k}$ különbséget a p osztja. Tehát bár a p értékét továbbra sem tudjuk, de föltettük, hogy $p|n$.

Amennyiben

$$1 < d = \text{lnko}(x_k - x_{2k}, n) < n$$

teljesül, akkor d egy nem triviális osztója n-nek.

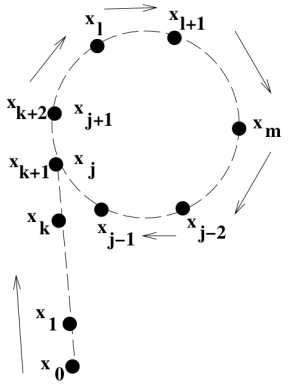
Azt szeretnénk elérni, hogy az $x_0, x_1, \ldots$ sorozat mutasson valami véletlenszerűséget. Ez az $f(x)$ polinom választásán múlik. Az $f(x) = ax + b$ bizonyítottan nem jó. A következő legegyszerűbb eset: $f(x) = x^2 + 1$ Azt nem tudjuk, hogy ez a függvény elég véletlen-e, ugyanakkor csak $(p + 1)/2$ különböző értéket vesz föl mod p, ami jó tulajdonság

A módszer pszeudokódja az alábbi:

```
1: a=2; b=2;
2: while (true) {
3:    a = (a * a + 1) mod n;  // f(x) = x^2 + 1 (mod n)
4:    b = (b * b + 1) mod n;
5:    b = (b * b + 1) mod n;  // x = f(f(x))
6:    d = lnko(a-b, n);
7:    if (1 < d && d < n) return d;
8:    if (d == n) return n; // kudarc
9: }
```

A 4. és 5. sorok együtt biztosítják, hogy a b sorozat kétszer gyorsabban megy, mint az a. Ha $\text{lnko}(a - b, n) = 1$, akkor folytatódik a ciklus.

A tapasztalat azt mutatja, hogy nagyobb n-ekre elég jól működik, kicsi n-re nem érdemes használni. Hívják ρ módszernek is, a ciklikusság miatt. Ennek magyarázatához lásd a 7.1. ábrát, ahol a $x_i \mod p$ sorozat lépéseit mutatjuk be.



7.1. ábra. Pollard faktorizációs eljárást ρ-módszernek is nevezzük.

Ha a módszer kudarcot vall, akkor kipróbálhatjuk az $f(x) = x^2 + c$ függvényt is, ilyenkor a $c \neq 0, 1, -2$.

Ezzel az algoritmussal faktorizálták a 8. Fermat számot. Ezek az $F_n = 2^{2^n} + 1$, alakú egészek, amely $n = 8$ esetben összetett számot ad. Az $n = 8$-ra azért működött hatékonyan, mert az F_8 két prímszám szorzata: $p_1 p_2$, és $p_1 = 1238926361552897$ sokkal kisebb, mint p_2.
Rendkívül érdekes tény, hogy csak $n = 1, 2, 3, 4$-re tudjuk, hogy F_n prímszám.

Példa. Faktorizáljuk az $n = 1387 = 19 \cdot 73$ számot. Legyen $x_0 = 0$ és használjuk az $y_i \equiv x_{2i} - x_i \mod 1387$ függvényt.

i	x_i	x_{2i}	y_i	lnko$(yi, 1387)$
1	2	1	1	1
2	2	26	24	1
3	5	620	615	1
4	26	582	556	1
5	677	829	152	19

Az ötödik lépés végén a 19 prímtényezőt meghatároztuk.

7.3.3. Fermat-faktorizáció

A Fermat-faktorizáció a következő megfigyelésen alapszik. Tetszőleges páratlan szám felírható két négyzetszám különbségeként, azaz $n = a^2 - b^2$. Az algoritmus pszeudokódja az alábbi.

```
Bemenet: n páratlan szám
a = ceil(sqrt(n)); // felső egészrész
b = a * a - n;
while (not negyzetszam(b1)){
   a++
   b = a * a - n
}
return a + sqrt(b);
```

Láthatjuk, hogy az algoritmus szisztematikusan keres olyan a és b értékeket, amelyre $n = a^2 - b^2$.

A Fermat-faktorizáció jól működik, ha n két azonos nagyságrendű osztóval rendelkezik.

Négyzetszámok meghatározására az alábbi karakterizációt használhatjuk: az utolsó két jegyük

$$00, s1, s4, 25, t6, s9$$

alakú, ahol s páros és t páratlan.

Példa. Legyen $n = 517$. Az algoritmus az alábbi lépéseket hajtja végre.

a1	b1	megjegyzés
23	12	ceil($\sqrt{517}$) = 23
24	59	
25	108	
26	159	
27	212	
28	267	
29	324	$324 = 18 \cdot 18$, négyzetszám

Az algoritmus itt a $29+18 = 47$ visszatérési értékkel leáll, azaz $517 = 29^2 - 18^2 = (29-18)(29+18) = 11 \cdot 47$.

7.3.4. Euler-faktorizáció

Az Euler módszer alapja, hogy megkeresse az adott számot két négyzetszám összegeként kifejezve, mégpedig azt két különbözőképpen megadva. Például az 1000009 felírható 1000^2+3^2 vagy 972^2+235^2 alakban, ahonnan Euler módszere az $1000009 = 293 \cdot 3413$ felbontást adja.

A módszer a Fermat-faktorizációnál általában gyorsabb, ha a faktorok nincsennek közel egymáshoz. Ugyanakkor fontos megjegyeznünk, hogy amennyiben valamelyik prímfaktor $4k + 3$ alakú, akkor a módszer nem működik.

Legyen tehát $n = a^2 + b^2 = c^2 + d^2$ alak ismert, ebből n-et két négyzet-összeg szorzataként adjuk meg. Először

$$a^2 - c^2 = d^2 - b^2$$

ahonnan kapjuk, hogy

$$(a - c)(a + c) = (d - b)(d + b) \tag{7.2}$$

Legyen most $k = \text{lnko}(a-c, d-b)$ és $h = \text{lnko}(a+c, d+b)$ azaz léteznek olyan l, m, l', m' konstansok, amelyre

$$(a - c) = kl, \quad (d - b) = km, \quad \text{és lnko}(l, m) = 1,$$

valamint

$$(a + c) = hm', \quad (d + b) = hl', \quad \text{és lnko}(l', m') = 1.$$

Ezeket behelyettesítve az (7.2) képletbe kapjuk, hogy

$$klhm' = kmhl'$$

A közös tagokat kiejtve adódik, hogy

$$lm' = l'm$$

Használjuk ki, hogy (l, m) és (l', m') relatív prímekből álló párok, tehát

$$l = l' \quad \text{és} \quad m = m'$$

Innen

$$(a - c) = kl, \;\; (d - b) = km, \;\; (a + c) = hm, \;\; (d + b) = hl.$$

Láthatjuk, hogy $\text{lnko}(a + c, d - b)$ és $l = \text{lnko}(a - c, d + b)$. A Bahmagupta–Fibonacci azonosságot[4] alkalmazva kapjuk, hogy

$$\begin{aligned}
\left(k^2 + h^2\right)\left(l^2 + m^2\right) &= (kl + hm)^2 + (km - hl)^2 \\
&= \big((a - c) + (a + c)\big)^2 + \big((d - b) - (d + b)\big)^2 \\
&= (2a)^2 + (2b)^2 \\
&= 4n.
\end{aligned}$$

Mivel minden faktor két négyzetszám összege, ezért az egyikük páros számpárokat tartalmaz: vagy (k, h) vagy pedig (l, m). Tegyük fel, hogy a (k, h) páros. A faktorizáció eredménye ekkor

$$n = \left(\left(\frac{k}{2}\right)^2 + \left(\frac{h}{2}\right)^2\right)\left(l^2 + m^2\right).$$

[4] $\left(a^2 + b^2\right)\left(c^2 + d^2\right) = (ac - bd)^2 + (ad + bc)^2 = (ac + bd)^2 + (ad - bc)^2$

7.4. Faktorizáló algoritmusokról

A prímfaktorizáló eljárásokat alapvetően két kategóriába szokás sorolni.

- A speciális célú kategóriába azok az algoritmusok tartoznak, amelyeknek a futási ideje a faktorizálandó egész valamelyik prímtényezőjének tulajdonságától (mérete, speciális alakja, stb) függ. A tényleges futási idő algoritmusonként általában különböző. Ebbe a kategóriába tartoznak azok is tehát, amelyek a legkisebb prímfaktor méretétől függő futási idővel rendelkeznek:
  - a 7.3.1. szakaszban tárgyalt próbálgatásos módszer,
  - a 7.3.2. szakaszban tárgyalt Pollard-féle ρ-módszer,
  - a 7.3.3. szakaszban tárgyalt Fermat-módszer,
  - valamint a 7.3.4. szakaszban tárgyalt Euler-faktorizáció.

  Megjegyezzük, hogy természetesen további eljárások is léteznek.

- Az általános célú kategóriába azok az algoritmusok tartoznak, amelyek futási ideje a faktorizálandó egész méretétől függ. Ilyen algoritmus például a Dixon eljárás, amely a következő ötleten alapszik. Mint azt láttuk, a Fermat-faktorizációban olyan a és b egészeket keresünk, amelyre $n = a^2 - b^2$. Dixon eljárásában elég olyan számpárt találni, amelyre $a^2 \equiv b^2 (\mod n)$ Például az $n = 84923$ esetében a keresést 292-nél kezdve (ez az első olyan szám, amely $\sqrt{n}$-nél nagyobb) találjuk, hogy $5052 \mod 84923 = 256$, amely pontosan a 16 négyzete. Ezért $(505 - 16)(505 + 16) = 0 \mod 84923$. Ha kiszámítjuk az n és $505 - 16$ legnagyobb közös osztóját, akkor kapjuk, hogy az 163, amely az n egyik prímfaktora.
  Az algoritmusban használunk egy alkalmas B korlátot, amelynek választása alapvetően tetszőleges, ugyanakkor túl kicsi korlát esetén csökken annak a valószínűsége, hogy nemtriviális prímfaktort találjuk, míg túl nagy korlát esetén nagyon sokáig fut az eljárás, mert túl sok ellenőrzést kell elvégezni. Például 292 darab olyan négyzetszám van, amely kisebb, mint 84923, ebből 662 dabab 84923-nél kisebb olyan van, amelynek prímfaktorai a $2, 3, 5$ vagy 7 prímszámok, és 4767 darab szám van, amelynek prímfaktorai 30-nál kisebbek. Az általános fogalom itt a B-sima szám, amelyre teljesül, hogy egyik prímfaktora sem nagyobb, mint B.

Megemlítjük még, hogy a kvantumszámítógépekre is készült egy faktorizáló eljárás, amely Shor-algoritmus néven ismert.

7.5. További megjegyzések

7.5.1. Szemiprímek

Adott számjegyű számok közül nem mindegyiknek ugyanolyan nehéz a prímfaktorizálása. Ezek közül különösen fontos halmazt alkotnak a szemiprímek, amelyek az $n = pq$ alakú egészek, ahol p és q prím. Más néven ezek az RSA számok, a jelenleg ismert eljárásoknak ezek a legnehezebben felbontható számok.

Az 'RSA factoring challenge' egy 1991-2007 között zajló, pénzdíjas verseny volt, amelyben megadott összetett számok szemiprím felbontását kellett megkeresni. Bár a verseny már nem aktív, a keresés nem állt le. A legnagyobb kihívás az RSA-2048, amelynek 617 decimális számjegye van (2.048 bit), egyelőre sikertelen a felbontás megtalálása. A legnagyobb sikeresen faktorizált RSA szám 232 decimális számjegyű (768 bit), ezt 2009-ben találták meg.

7.5.2. Ikerprímek

Az ikerprímek azok a prím számpárok, amelyeknél a prímhézag pontosan 2. Bár nagyméretű prímek esetén a prímhézag általában egyre nagyobb, a mai napig nyitott kérdés, hogy vajon végtelen sok ikerprímszám van-e.

7.6. Feladatok

1. Legyenek m és n relatív prímek. Faktorizálás nélkül mutassuk meg, hogy $m|N$ és $n|N$ fennállásából következik, hogy $mn|N$. Faktorizálás felhasználásával könnyebb a bizonyítás?

2. Egy természetes számot k-majdnem prímszámnak nevezünk, ha pontosan k darab prímfaktora van. Írjunk eljárást, amely ilyen számokat keres adott k értékre.

3. Legyen $\phi(n)$ azon egészek száma, amelyek relatív prímek n-hez (Euler-függvény). Mutassuk meg, hogy pozitív n egészre
$$n = \sum_{d|n, d>0} \phi(d)$$
teljesül.

4. A Lucas-Lehmer teszt a 6.5. szakaszban tárgyalt Mersenne prímek keresésére szolgál. A teszt a következő. Legyen p páratlan prím. A $2^p - 1$ Mersenne szám akkor és csak akkor prímszám, ha $2^p - 1$ osztja $S(p-1)$-et, ahol
$$S(n+1) = \begin{cases} (S(n)^2 - 2 & \text{ha } n > 0, \text{ és} \\ 4 & \text{ha } n = 0. \end{cases}$$
Írjunk programot, amely a beépített típusok felhasználásával kiszámítja a Mersenne prímeket. Legfeljebb a 47. ilyen prímig menjünk el.

5. Írjunk hatékony Mersenne szám faktorizációs eljárást.

6. Készítsük el a 7.3.4. szakaszban tárgyalt Euler-faktorizáció implementációját.

8. fejezet

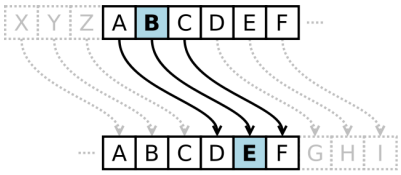
Titkosítás, véletlenszámok

8.1. Titkosítás

A feladat, amivel ebben a részben foglalkozunk a következőképpen fogalmazható meg. B egy üzenetet akar küldeni A-nak úgy, hogy C, aki hallgatózik, ne értse az üzenetet. A rendelkezik egy kulccsal, ami a rejtjelezést megfejti.

8.1.1. Cézár-rejtjel

Az ötlet rendkívül egyszerű: toljuk el az ábécé karaktereit k lépéssel, természetesen cirkulárisan. Az $k = 4$ eltolást a 8.1. ábra illusztrálja.



8.1. ábra. Cézár-rejtjelezés $k = 4$ eltolással. Forrás: Wikipédia.

8.1.2. One-time pad

Arra a kérdésre ad *igen* választ, hogy létezik-e tökéletes titkosítási eljárás, azaz olyan eljárás, melyben a titkosított szöveg ismerete alapján semmiféle információt nem lehet megtudni a nyílt szöveg tartalmáról.

A módszer a következő: legyen az üzenet n hosszúságú egy k ábécé felett. A kulcs is n hosszúságú, amelyet betűnként hozzáadunk $\mod k$.

A módszer nagyon biztonságos, de a kulcs túl hosszú, amennyiben az üzenet is hosszú.

Példa. Tegyük fel, hogy 26 elemű az ábécénk és a kulcsunk az XMCKL karaktersorozat. Elküldendő szó: HELLO. Az üzenet titkosításának lépéseit a 8.1. táblázat mutatja, míg a visszafejtését pedig a 8.2. táblázat.

A kulcsok előállításához használhatunk (pszeudo-)véletlenszám generátort (lásd a 8.2. szakaszt ebben a fejezetben). Amennyiben A és B ugyanazt a kiindulási értéket (*seed*-et) használja, akkor ugyanazt a sorozatot kapják. A módszer jellemzője, hogy mindkét félnél tehát ugyanaz a kulcs van. Az ilyen protokollokat *szimmetrikusnak* nevezzük.

Ettől különbözik a következő szakaszban tárgyalt Diffie-Hellman protokoll (1976), amely nyilvános kulcsú titkosítást tesz lehetővé.

8.1. táblázat. A HELLO üzenet titkosítása 26 elemű ábécével és az XMCKL kulcs használatával

	H	E	L	L	O	üzenet
	7 (H)	4 (E)	11 (L)	11 (L)	14 (O)	üzenet
+	23 (X)	12 (M)	2 (C)	10 (K)	11 (L)	kulcs
=	30	16	13	21	25	üzenet + kulcs
=	4 (E)	16 (Q)	13 (N)	21 (V)	25 (Z)	üzenet + kulcs (mod 26)
	E	Q	N	V	Z	titkosított szó

8.2. táblázat. A HELLO üzenet visszafejtése 26 elemű ábécével és az XMCKL kulcs használatával

	E	Q	N	V	Z	titkosított szó
	4 (E)	16 (Q)	13 (N)	21 (V)	25 (Z)	titokszó
−	23 (X)	12 (M)	2 (C)	10 (K)	11 (L)	kulcs
=	-19	4	11	11	14	titokszó − kulcs
=	7 (H)	4 (E)	11 (L)	11 (L)	14 (O)	titokszó − kulcs (mod 26)
	H	E	L	L	O	üzenet

Diffie-Hellman protokol

A protokoll két külön kulcsot használ, egyet a titkosításhoz és egyet a megfejtéshez. Ez tehát *aszimmetrikus* protokoll. A módszer illusztrálásához tekintsük a következő példát.

Példa. Alice és Bob megegyeznek, hogy a $p = 23$ prímszámot és a $g = 5$ bázist használják. Alice választ egy titkos egész számot: $a = 6$, majd elküldi Bob-nak az $A = g^a \mod p$ értéket:

$$A = 5^6 \mod 23, \quad A = 15.625 \mod 23 \Rightarrow A = 8.$$

Bob is választ egy titkos egész számot: $b = 15$, majd elküldi Alice-nek a $B = g^b \mod p$ értéket:

$$B = 5^{15} \mod 23, \quad B = 30.517.578.125 \mod 23 \Rightarrow B = 19.$$

Alice ezután kiszámolja az $s = B^a \mod p$ értéket:

$$s = 19^6 \mod 23, \quad s = 47.045.881 \mod 23 \Rightarrow s = 2$$

Bob kiszámolja az $s = A^b \mod p$ értéket:

$$s = 8^{15} \mod 23, \quad s = 35.184.372.088.832 \mod 23 \Rightarrow s = 2$$

Alice és Bob közös titka tehát $s = 2$.

Nyilvánvaló, hogy ha valaki tudja a két személyes egészet (6-ot és 15-öt), akkor kiszámíthatja s értékét:

$$\begin{aligned} s &= 5^{6\cdot 15} \mod 23 \\ s &= 5^{15\cdot 6} \mod 23 \\ s &= 5^{90} \mod 23 \\ s &= 807.793.566.946.316.088.741.610.050.849.573.099.185.363.389.551.639.556.884.765.625 \mod 23 \\ s &= 2 \end{aligned}$$

Vegyük észre, hogy ebben a számolásban a moduláris hatványozást használjuk, amelynek hatékony kiszámítását a 2.4. szakaszban tárgyaltuk.

A protokoll alkalmazásánál természetesen sokkal nagyobb a, b és p értékeket kell használni. A példában a $g^{ab} \mod 23$ műveletnek csak 23 különböző eredménye lehet, ez egyszerűen végigpróbálgatható.

Azonban ha p legalább 300 jegyű prím, a és b pedig 100 jegy hosszú, akkor a megtalálásához az összes rendelkezésre álló számítógép sem lenne elegendő, pedig ismerhetjük: $g, p, g^b \mod p$ és $g^a \mod p$ értékeit. Ezt *diszkrét logaritmus* problémának nevezzük, amit tehát véges ciklikus testek felett értelmezünk.

Megjegyezzük, hogy egyébként g-nek nem kell nagynak lennie, sokszor elég a 2,3 vagy 5.

Diszkrét logaritmus. Legyen $\mathbb{Z}_m$ az egészek modulo m véges halmaz. A $\mathbb{Z}_m^*$ az *invertálható* elemek halmaza, azaz olyan $a \in \mathbb{Z}_m$, amelyre létezik $b \in \mathbb{Z}_m$ úgy, hogy

$$ab \equiv 1 \pmod{m}.$$

Ezek egyébként azok az elemek, amelyre $\text{lnko}(a, m) = 1$. Az az eset az érdekes most, amikor m egy p prímszám, ekkor $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.

Egy g számot *generátornak* nevezünk, ha a $g, g^2, g^3, \ldots, g^{p-1} \mod p$ tartalmazza az összes $1, 2, \ldots p-1$ számot. Minden p prímhez létezik generátor. Nagyon sok g generátor létezik, de ezek megtalálására nincs ismert determinisztikus polinomiális idejű eljárás. Sőt, még annak a letesztelése, hogy egy adott szám generátor-e sem megoldott determinisztikus polinomiális időben.

Például a $p = 13$ a $\mathbb{Z}_p^*$ generátora a $g = 2$, ugyanakkor a $g = 4$ nem az.

Ezen definíciók felhasználásával a *diszkrét logaritmus* meghatározása a következő. Legyenek adottak az y, g és p számok, ahol p prím és g a $\mathbb{Z}_p^*$ generátora, keressük meg azt az x-et, amelyre

$$g^x \equiv y \pmod{p}$$

Ez lesz az y szám g alapú diszkrét logaritmusa modulo p.

8.2. Véletlenszám generálás

Tekintsük a következő, Java nyelven írt kódot:

```
public static void main(String ... args) {
    System.out.println(randomString(-6225973)+' '+
                       randomString(1598025));
}

public static String randomString(int seed) {
    Random rand = new Random(seed);
    StringBuilder sb = new StringBuilder();
    for (int i=0; i<5; i++)
        sb.append((char) ('a' + rand.nextInt(26)));
    return sb.toString();
}
```

Vajon mi lesz a kimenete?

A példa jól illusztrálja azt a tényt, hogy bár látszólag egy olyan kódot mutattunk, amelynek kimenetétől azt várjuk, hogy mutasson valami véletlenszerűséget, az mégis teljesen determinisztikus.

8.2.1. Véletlenszámok előállítása külső eszközök felhasználásával

Tegyük fel, hogy szükségünk van egy véletlen számjegyre és ezt nem számítógépen akarjuk előállítani. Az alábbi módszerek közül melyek alkalmasak erre? A következő lista a [1] könyvből származik:

1. Kinyitunk találomra egy telefonkönyvet és az első telefonszám egyes helyiértékű számjegyét használjuk.

2. Ugyanaz, mint az előbb, de most az *oldalszám* egyes helyiértékű jegyét tekintjük.

3. Feldobunk egy szabályos ikozaédert, amelynek a 20 oldalára rendre a $0, 0, 1, 1, \ldots, 9, 9$ számok vannak írva és a legfelső lapon lévő számot használjuk.

4. Egy Geiger-Müller-számlálót egy percig radioaktív sugárzásnak teszünk ki, és a számláló egyes helyiértékű számjegyét használjuk. Feltesszük, hogy a számláló tízes számrendszerben számlál és eredeti tartalma nulla.

5. Rápillantunk a karóránkra. Ha a másodpercmutató $6n$ és $6(n+1)$ másodperc között áll, akkor az n számjegyet választjuk.

6. Megkérjük egy barátunkat, hogy mondjon egy véletlen számjegyet.

7. Megkérjük egy ellenségünket, hogy mondjon egy számjegyet véletlenszerűen és ez a számot használjuk.

8. Tegyük fel, hogy tíz ló indul egy versenyen, és semmit sem tudunk a képességeikről. A lovakhoz valamilyen módon hozzárendeljük a $0-9$ számjegyeket és a győztes ló számát használjuk.

8.2.2. Neumann négyzetközép

Neumann János 1946-ban a következő, *négyzetközép-módszernek* nevezett eljárást javasolta. Induljuk ki egy tetszőleges, lehetőleg sok számjegyet tartalmazó számból. Az utolsó számot a sorozatban emeljük négyzetre és ennek a középső jegyei alkossák a következő számot.

Ha tehát például 10 jegyű számokra van szükségünk, és az utolsó szám 5 772 156 649, akkor ennek négyzete

$$33317792380594909201,$$

és így a következő szám 7 923 805 949 lesz.

Sajnos a módszer nem igazán jó: általában hamar periodikussá válik. A periodikusság nem meglepő, hiszen csak véges sok 10 jegyű szám van, és amint egy szám megismétlődik, a sorozat periodikus lesz. Az igazi kérdés az, hogy ez milyen hamar következik be. Sajnos a tapasztalat szerint a módszerben általában pár száz lépés után a periodikusság bekövetkezik.

Vegyük észre, hogy a módszer úgy javasol véletlenszerű sorozat gyártását, amelynek minden tagját egyértelműen meghatározza az előtte levő. Ez önmagában nem feltétlenül probléma, viszont az biztos, hogy csak azzal az elvárással élhetünk, hogy '*nem véletlen, de annak látszik*' típusú sorozatot gyártunk. Az alábbi két követelménynek kell eleget tenni az $x_0, x_1, \ldots x_i$ sorozatnak:

(i) legyen (majdnem) egyenletes eloszlású, és

(ii) ne lehessen x_i-t kitalálni az $x_0, \ldots x_{i-1}$ ismeretével.

8.2.3. Lineáris kongruenciák

Egyenletes eloszlású, pszeudovéletlen számok generálása lineáris kongruenciával az alábbi iteráción alapszik:

$$x_{i+1} \equiv ax_i + c \pmod{m}$$

A definícióból nyilvánvaló, hogy a sorozat ciklikus lesz. Tapasztalat szerint a következőket tudjuk:

1. a $c = 0$ esetben a generálási eljárás egy picit gyorsabb, viszont lecsökken a periódus hossza;

2. az $a = 1$ eset elvethető, olyankor nem kapunk véletlenszerűen viselkedő sorozatot, az $a = 0$ még rosszabb választás;
3. valamint a leghosszabb (m lépésű) periódus eléréséhez szükséges és elegendő, hogy
   - c és m legyenek relatív prímek
   - $b = a - 1$ osztható az m minden prímfaktorával
   - $b = a - 1$ a 4 többszöröse, ha m is a 4 többszöröse
   - b nem 0.
4. A fentiek alapján az m gyakran 2^{32} vagy 2^{64}, mert akkor csak levágni kell a modulus számításához.

Példa. Nézzünk most egy látszólag jónak (eléggé szofisztikáltnak) tűnő választást. Kérdés, hogy mi annak a lineáris kongruenciasorozatnak a periódushossza, amelyben a következő paraméter választással éltünk:

$$\begin{aligned} x_0 &= 5772156648, & a &= 3141592621 \\ c &= 2718281829, & m &= 10000000000 \end{aligned}$$

A válasz, talán nagy meglepetésre: 3.

RANDU. A RANDU eljárást még a 60-as években használták, amely a következő kongruencia sorozatot használja:

$$x_{i+1} = 65539x_i \pmod{2^{31}}$$

A kiindulási érték x_0 páratlan szám kell, hogy legyen. A paraméterek választását a 32-bites szó használatát alkalmazó számítógépek indokolták, amelyeken a $\bmod 2^{31}$ művelet hardveres szinten gyorsan elvégezhető. Azonban $65539 = 2^{16} + 3$ és

$$x_{i+2} = (2^{16} + 3)x_{i+1} = (2^{16} + 3)^2 x_i \pmod{2^{31}}$$

ha most kibontjuk a négyzetes tagot:

$$x_{i+2} = (2^{32} + 6 \cdot 2^{16} + 9)x_i = (6 \cdot (2^{16} + 3) - 9)x_i \pmod{2^{31}}$$

mivel $2^{32} \bmod 2^{31} = 0$.
Ebből már látszik, hogy

$$x_{i+2} = 6x_{i+1} - 9x_i$$

Ez lényegében megkérdőjelezi az összes 60-as években számított (tudományos) eredményt, ahol ezt a módszert használták.

Lineáris kongruenciák a gyakorlatban A gyakorlatban úgy állítunk elő véletlenszámokat, hogy az

$$x_{n+1} = (ax_n + c) \bmod m$$

sorozatból indulunk ki, ahol az x-ek *egészek*; ezután az $u_n = x_n/m$ *törteket* használjuk. Ekkor az u_n-ekre vonatkozó rekurzió:

$$u_{n+1} = (au_n + c/m) \bmod 1.$$

Vegyük figyelembe azonban, hogy az au_n szorzat olyan nagy, hogy a c/m akár nem is változtat rajta. Továbbá a $\bmod 1$ is levághat értékes jegyeket, ezért lebegőpontos számításnál ekkor *dupla* pontosság kell.

8.2.4. Mersenne twister

A módszert Matsumoto és Nishimura javasolt 1997-ben. Szintén lineáris kongruencia módszer, viszont a periódusa $2^{19937} - 1$, amely egy Mersenne prím. A standard változat az MT19937 nevet viseli és 32-bites szóhosszat használ. A 64-bites változat az MT19937-64, ami egy másik sorozatot gyárt.

Az algoritmus sokkal gyorsabb, mint más generátorok; mégis: nem alkalmas titkosításra, mert elegendően sok tag megfigyelése után kitalálható a sorozat. További hátránya még, hogy az inicializálásra nagyon érzékeny, az elején esetleg nem is felel meg véletlen teszteknek.

Ezzel együtt szinte az összes, széles körben használt programozási nyelvben ez a standard eljárás a `rand()` függvényre.

8.2.5. Statisztikai próbák

A statisztikai próbák arra adnak lehetőséget, hogy matematikailag megalapozott módon eldöntsük egy sorozatról, hogy véletlenszerű-e.

Csak említés szintjén: a χ^2-próba diszkrét (nominális (vagy kategoriális)) változók vizsgálatára, a Kolmogorov-Szmirnov-próba folytonos eset (amikor tehát a véletlen valós számok végtelen sok értéket vehetnek fel).

Diehard tesztek A teszt-sorozat egy alapvetésként szolgál véletlenszerűség vizsgálatára. Röviden ismertetjük a felhasznált módszereket.

Születésnap beosztás Válaszzunk véletlen pontokat egy nagy intervallumból. A pontok közötti távolságok aszimptotikusan exponenciális eloszlást kell, hogy kövessenek. A teszt neve a korábban említett születésnap paradoxonból jön.

Átlapoló permutációk: Vizsgáljunk 5 egymás utáni véletlenszám sorozatát. A 120 különböző sorbarendezés ugyanakkora valószínűséggel kellene, hogy előforduljon.

Mátrixok rangjai Válaszzunk néhány bitet néhány számból, amelyek véletlenszámokból származnak. Ezekből képezzünk $\{0, 1\}$ feletti mátrixokat. Számoljuk össze a rangjaikat.

Majom teszt: Tekintsük bitek egy sorozatát 'szavaknak'. Számoljuk össze az átlapoló szavak számát. Azon szavak száma, amelyek nem fordulnak elő egy adott eloszlást kell, hogy kövessenek. Az elnevezés a végtelen majom problémából ered.

Számoljuk össze az 1-eseket: Számoljuk össze az 1-es bitek számát, majd ezt alakítsuk 'betűkké'. Számoljuk össze az öt betűs 'szavak' előfordulásának számát.

Parkolóház teszt: Helyezzünk el véletlenszerűen egységköröket egy 100×100-as négyzetbe. Egy kör akkor parkolt sikeresen, ha nincs átfedése a már eddig sikeresen parkolt körök egyikével sem. 12.000 próbálkozás után a sikeresen parkolt körök száma egy bizonyos normális eloszlást kell, hogy kövessen.

Minimális távolság teszt: Helyezzünk el 8000 pontot véletlenszerűen egy 10000×10000-es négyzetbe, majd számítsuk ki a párok közötti legrövidebb távolságot. Ennek a távolságnak a négyzete exponenciális eloszlást kell, hogy kövessen egy adott átlaggal.

Véletlen gömbök teszt: Válaszzunk véletlenszerűen 4000 pontot egy 1000 oldalhosszúságú kockából. Helyezzünk el gömböket a kiválasztott pontokra, amelyeknek a sugara a többi ponttól mért távolság minimuma legyen. A legkisebb gömb-térfogat exponenciális eloszlást kell, hogy kövessen.

The squeeze test: Szorozzuk meg a 2^{31}-et a $(0, 1)$ intervallumból véletlenül választott lebegőpontos számokkal addig, amíg nem érjük el az 1-et. Ismételjük meg ezt 100000 alkalommal. Az 1 eléréséhez szükséges próbálkozások száma egy adott eloszlást kell, hogy kövessen.

Átlapoló összegek teszt: Hozzunk létre a $(0, 1)$ intervallumból választott véletlen lebegőpontos számokból egy hosszú sorozatot. Adjuk össze 100 egymás után következő elemet. Az összegnek normális eloszlásúnak kell lennie.

Run teszt: Hozzunk létre a $(0, 1)$ intervallumból választott lebegőpontos számokból egy hosszú sorozatot. Számoljuk össze a növekvő és csökkenő run[1]-ok számát. Ezek egy adott eloszlást kell, hogy kövessenek.

Craps test Játszunk le 200000 craps[2] fordulót, miközben összeszámoljuk a nyeréseket és a dobások számát játékonként. Minden ilyen összeszámlálás egy adott eloszlást kell, hogy kövessen.

8.2.6. További megjegyzések

A `http://www.random.org/` honlapon véletlenszámok sorozatát érhetjük el. A szolgáltatás egy része ingyenes. Az oldal üzemeltetői szerint a megadott véletlenszám sorozatok atmoszférikus zaj eredetűek, így jobbak, mint a fentebb tárgyalt pszeudo véletlenszám generátorok.
Ugyanitt érdemes átolvasni a `http://www.random.org/analysis/` oldalt is, ahol a statisztikai elemzésekről találunk további érdekességeket, hivatkozásokat.

Az érdeklődő olvasó figyelmébe ajánljuk továbbá a [1] könyv 3.5 szakaszát, amely a '*Mit jelent az, hogy véletlen sorozat?*' címet viseli.

Végül megemlítjük, hogy a fentiekben lényegében egyenletes eloszlású véletlenszámok előállításáról volt szó, azonban másfajta eloszlásra is szükség lehet. Továbbá sok esetben véletlen permutációt (*shuffle*) vagy véletlen kombinációt kell előállítanunk - az ehhez használt módszerek a már megismert eljárások kiterjesztését kívánják.

8.3. Feladatok

1. Amennyiben a használt operációs rendszerünk lehetővé teszi nem csak szoftveres alapú véletlenszám generálást, akkor írjuk olyan programot, amely ezt a hardveres alapú generálást demonstrálja.
   Példaként a UNIX rendszerekben elérhető `/dev/random` nevű fájlt lehet itt említeni, amely a rendszerben működő eszközmeghajtóktól gyűjt zajt, és ezt használja föl pszeudo-véletlenszám generálására.

2. Feltéve, hogy adott egy eljárás, amely egyenletes eloszlással véletlenszámot generál, írjunk egy olyan eljárást, amely adott átlaggal és szórással normális eloszlású véletlenszámokat generál.

3. Cinkelés érme helyrehozatala. Amennyiben adott egy érménk, amely P_0 valószínűséggel 0 (fej) és P_1 valószínűséggel 1 (írás) értéket ad, ahol P_0 $neq P_1$, akkor annak a valószínűsége, hogy az 1 értéket 0 követ $P_1 \cdot P_0$, míg annak, hogy a 0 értéket 1 követ $P_0 \cdot P_1$. Ez a két szorzat valószínűség már megegyezik. Ezért a kiegyensúlyozást úgy végezhetjük el, hogy csak akkor adunk vissza 0 vagy 1 értéket, ha két egymás utáni dobás eredménye különböző.
   Ennek egy általánosítása a következő feladat. Írjunk egy eljárást, amely 0 vagy 1 értékkel tér

[1] Egy sorozat *run* értéke az a maximális részsorozat, amely egymás után azonos értékeket tartalmaz. Például a 22 hosszú sorozat: '++++—+++–++++++—-' összesen hat darab run-t tartalmaz, ezek közül 3 olyan van, amely '+' jelet, a többi pedig '-' jelet.

[2] a craps játékot két kockával jászták, bővebben lásd `https://hu.wikipedia.org/wiki/Craps`

vissza, de úgy, hogy az 1 érték, átlagosan egyszer fordul elő N hívásból, ahol $N \in [3, 6]$. Ezután írjunk egy másik eljárást, amely az előzőt használja a véletlenszám generálásra, viszont azt helyrehozza.

4. Generáljunk 100 darab véletlen (x, y) számpárt egyenletes eloszlással úgy, hogy azokra teljesüljön a

$$10 \leq \sqrt{x^2 + y^2} \leq 15$$

egyenlőtlenség sorozat. Rajzoljuk ki a kapott (x, y) számpárokat egy Descartes-féle koordináta rendszerben.

5. Adott (n, e, d) RSA kulcsokkal írjunk RSA titkosító eljárást.

9. fejezet

Polinomok faktorizálása

A polinomok gyökeinek egzisztenciájára vonatkozó tétel alapján bebizonyítható, hogy a komplex és valós számtestek fölötti polinomok irreducibilis faktorizációja létezik és egyértelmű.

Olyan polinomokat keresünk, amelyek a polinomgyűrűben ugyanolyan szerepet játszanak, mint az egész számok gyűrűjében a prímszámok.

A polinomokat általában valamilyen test felett definiáljuk (ahol az osztás elvégezhető). Leggyakrabban:

- racionális számok felett,
- valós vagy komplex számok felett,
- vagy mint a fejezetben bemutatott példában is: egészek modulo p prím felett.

Külön jelentőségű az egészek mod 2 feletti polinomoknak: ezek számos analógiát mutatnak a kettes számrendszerben kifejezett egészekkel. Azonban polinomoknál az együtthatók nincsennek kapcsolatban, nincs *átvitel*.

A test, ami fölött a polinomokat tekintjük, legrögzítése fontos lehet, mert például az x^2-2 polinom az egészek és a racionális számok felett irreducibilis, ugyanakkor felbontható a valós és komplex számok felett:

$$x^2 - 2 = (x - \sqrt{2})(x + \sqrt{2}).$$

Másik példa: az $x^2 + 1$ a racionális és valós számok felett irreducibilis, ugyanakkor

$$x^2 + 1 = (x - i)(x + i)$$

A következőkben rátérünk a talán legfontosabb eset tárgyalására, amely a véges testek feletti polinomok faktorizációja. Az ide tartozó eljárások a fordulnak elő leggyakrabban az egyes számítógépes algebrai rendszerekben. Néhány szemléletes példa erre az esetre (a $\mathbb{Z}_p[x]$ jelentése: polinomok, amelyeknek az együtthatói a $\{0, \ldots, p-1\}$ halmaz elemei):

- a $\mathbb{Z}_5[x]$ esetén $x^2 - 4 = (x + 2)(x + 3)$
- a $\mathbb{Z}_5[x]$ esetén $x^2 + 4 = (x + 1)(x + 4)$
- ...

9.1. Felbontás modulo p

A következőkben tárgyalt eljárást Berlekamp 1967-ben javasolta.

Legyen p prímszám; a következőkben a polinomokkal mindenféle számítást modulo p végzünk. Tegyük fel, hogy adott egy $u(x)$ polinom, amelynek együtthatói a $\{0, 1, \ldots, p-1\}$ halmazból valók, és feltehetjük, hogy $u(x)$ normált. Célunk az, hogy $u(x)$-et

$$u(x) = p_1(x)^{e_1} \cdot \ldots \cdot p_r(x)^{e_r}$$

alakban fejezzük ki, ahol $p_1(x), \ldots, p_r(x)$ egymástól különböző, normált irreducibilis polinomok.
Előkészítő lépés: eldöntjük, hogy $e_i > 1$ valamely i indexre. Ha

$$u(x) = u_n x^n + \ldots + u_0 = v(x)^2 w(x)$$

alakú, akkor a szokásos módon, de modulo p, kifejezett deriváltjára

$$u'(x) = n \cdot u_n x^{n-1} + \ldots + u_1 = 2v(x)v'(x)w(x) + v(x)^2 w'(x),$$

ami tehát a $v(x)$ többszöröse. Ezért határozzuk meg a

$$d(x) = \text{lnko}(u(x), u'(x))$$

polinomot. Ha

- $d(x) = 1$: akkor $u(x)$ négyzetmentes.
- $d(x) \neq 1$ és $d(x) \neq u(x)$: akkor d az u valódi tényezője
- $d(x) \neq 1$ és $d(x) = u(x)$: akkor $u'(x) = 0$, és ebből következik, hogy x^k-nak az u_k együtthatója csak akkor nem 0, ha a k a p többszöröse.
  Ebből következik, hogy
  $$u(x) = v(x^p) = (v(x))^p$$
  alakú, tehát ilyenkor valójában a $v(x)$ irreducibilis tényezőit kell meghatározni.

9.1.1. Állítás.

$$v(x^p) = (v(x))^p \tag{9.1}$$

Bizonyítás. Ha $v_1(x)$ és $v_2(x)$ modulo p vett polinomok, akkor

$$\begin{aligned}(v_1(x) + v_2(x))^p &= v_1(x)^p + \binom{p}{1} v_1(x)^{p-1} v_2(x) + \ldots \\ &\quad \ldots + \binom{p}{p-1} v_1(x) v_2(x)^{p-1} + v_2(x)^p \\ &= v_1(x)^p + v_2(x)^p,\end{aligned}$$

hiszen a binomiális együtthatók mind többszörösei a p-nek. Továbbá, a Fermat tétel szerint (amelyet a 6.3. szekcióban tanultunk) bármilyen a egész számra:

$$a^p \equiv a \pmod{p}.$$

Ezért, ha $v(x) = v_m x^m + \ldots + v_0$, akkor

$$\begin{aligned}v(x)^p &= (v_m x^m)^p + (v_{m-1} x^{m-1})^p + \ldots + (v_0)^p \\ &= v_m x^{mp} + v_{m-1} x^{(m-1)p} + \ldots + v_0 \\ &= v(x^p),\end{aligned}$$

amivel a (9.1) összefüggést bebizonyítottuk. ■

Feltesszük tehát, az előzőek alapján, hogy $u(x)$ négyzetmentes, keressük az

$$u(x) = p_1(x) \cdot \ldots \cdot p_r(x)$$

alakot, ahol $p_i(x)$ irreducibilis.

Ez volt az előkészítő rész, mehetünk tehát az algoritmus fő lépéseire.

A kínai maradéktétel érvényes polinomokra is. Ha $(s_1, s_2, ..., s_r) \mod p$ vett egészek bármilyen rendezett r-ese, akkor létezik pontosan egy olyan $v(x)$ polinom, hogy

$$\begin{aligned} v(x) &\equiv s_1 \pmod{p_1(x)} \\ &\cdots \\ v(x) &\equiv s_r \pmod{p_r(x)} \\ \deg(v) < \deg(p_1) + \ldots + \deg(p_r) &= \deg(u) \end{aligned} \tag{9.2}$$

Jelölés:

$$\begin{aligned} g(x) &\equiv h(x) \pmod{f(x)} \text{ ugyanaz, mint} \\ g(x) &\equiv h(x) \pmod{f(x) \text{ és } p}, \end{aligned}$$

mivel most modulo p aritmetikával dolgozunk.

A (9.2)-beli $v(x)$ polinom módot ad arra, hogy $u(x)$ tényezőit megkapjuk. A (9.2)-ből következik: ha $r \geq 2$, és $s_1 \neq s_2$, akkor

$$\begin{gathered} \text{lnko}(u(x), v(x) - s_1) \text{ osztható } p_1(x)\text{-szel,} \\ \text{de nem osztható } p_2(x)\text{-szel} \end{gathered}$$

Ez a megfigyelés azt mutatja, hogy a (9.2) megfelelő $v(x)$ megoldásaiból információt szerezhetünk $u(x)$ tényezőire nézve.

Ezért vizsgáljuk (9.2)-t: $v(x)$ kielégíti a

$$v(x)^p \equiv s_j^p = s_j \equiv v(x) \pmod{p_j(x)} \;\; 1 \leq j \leq r$$

feltételt (ne felejtsük: modulo p dolgozunk), ebből következik

$$v(x)^p \equiv v(x) \pmod{u(x)}, \quad \deg(v) < \deg(u) \tag{9.3}$$

Továbbá, bebizonyítható, hogy

$$x^p - x \equiv (x-0)(x-1)\ldots(x-(p-1)) \pmod{p}$$

amiből következik, hogy

$$v(x)^p - v(x) = (v(x)-0)(v(x)-1)\ldots(v(x)-(p-1)) \tag{9.4}$$

bármilyen $v(x)$ polinomra érvényes, amikor modulo p dolgozunk.

Ha $v(x)$ kielégíti (9.3)-t, akkor $u(x)$ osztja $(v(x)^p - v(x))$-et, ezért $u(x)$ minden irreducibilis tényezője kell, hogy ossza a (9.4) jobb oldalnak p darab relatív prím tényezőjéből valamelyiket. Azaz (9.3) *minden* megoldásának (9.2) alakúnak kell lennie valamely $s_1, \ldots, s_r$ számokkal, és (9.3) megoldásainak száma p^r, ahol r az $u(x)$ prímtényezőinek száma.

Ezért aztán (9.3) kongruencia $v(x)$ megoldásai $u(x)$ felbontásához adnak kulcsot.

Első pillantásra úgy tűnik, hogy (9.3) összes megoldásának megtalálása nehezebb $u(x)$ felbontásánál, de valójában ez nincs így, hiszen a (9.3) megoldáshalmaza az összeadásra nézve zárt. Másképpen fogalmazva: a (9.3) tulajdonságot teljesítő polinomok halmaza vektortér. Az alábbiakban megmutatjuk, hogy ennek a vektortérnek a bázisa a következő kulcs a megoldáshoz.

Legyen $\deg(u) = n$. Állítsuk elő a

$$Q = \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,n-1} \\ \vdots & \vdots & & \vdots \\ q_{n-1,0} & q_{n-1,1} & \cdots & q_{n-1,n-1} \end{pmatrix}$$

$n \times n$-es mátrixot, ahol

$$x^{pk} \equiv q_{k,n-1}x^{n-1} + \ldots + q_{k,1}x + q_{k,0} \pmod{u(x)}, \tag{9.5}$$

ahol $k = 0, \ldots, n-1$.

Ekkor $v(x) = v_{n-1}x^{n-1} + \ldots + v_1x + v_0$ pontosan akkor megoldása (9.3)-nak, ha

$$(v_0, v_1, \ldots, v_{n-1}) \cdot Q = (v_0, v_1, \ldots, v_{n-1})$$

(ne felejtsük, hogy modulo p dolgozunk)

9.1.2. Állítás. *A $(v_0, v_1, \ldots, v_{n-1}) \cdot Q = (v_0, v_1, \ldots, v_{n-1})$ egyenlőség akkor és csak akkor igaz, ha*

$$v(x) = \sum_j v_j x^j = \sum_j \sum_k v_k q_{k,j} x^j \equiv \sum_k v_k x^{pk} = v(x^p) \equiv v(x)^p,$$

ahol a kongruenciákat $\pmod{u(x)}$ kell érteni.

Bizonyítás. Vegyük sorba az egyenlőségeket:

- Az első egyenlőség csak a $v(x)$ definíciója
- A második egyenlőség a $vQ = v$ egyenlet részletesebb kiírása
- A harmadik egyenlőség az már egy kongruencia, az (9.5) miatt igaz
- Utána a negyedik egyenlőség az polinom definíció, semmi különös,
- Végül az utolsó egyenlőség megint egy kongruencia, az pedig a korábban látott és bizonyított (9.1) miatt igaz.

■

A fentiek képezik a Berlekamp algoritmus alapötletét. Knuth II. 430-434 De a következőkben leírunk egy *rövidített* változatot

9.1.1. Berlekamp algoritmus

Input: $u(x)$ polinom és p prímszám

Output: $u(x)$ faktorizációja modulo p

1. lépés. Vizsgáljuk meg, hogy $\text{lnko}(u(x), u'(x))$ értékét. Ha ez 1, akkor mehetünk tovább, hiszen $u(x)$ négyzetmentes. Ha nem 1, akkor csináljuk végig azokat a lépéseket, amiket a fóliasor elején láttunk.

2. lépés. Készítsük el a Q mátrixot.

3. lépés. Keressük meg a $(Q-I)^T$ mátrix nullterét. A nulltér dimenziója lesz az $u(x)$ faktorainak száma modulo p. A fenti jelöléseinket használva ez r.
Ennek eredménye olyan r elemű vektorsorozat lesz, amelyre $v^{(j)}(Q-I) = 0$ teljesül ($1 \leq j \leq r$).

4. lépés. Ha $r > 1$, akkor számítsuk ki az

$$\text{lnko}(u(x), v^{(2)}(x) - s) \bmod p$$

értékét, ahol $v^{(2)}(x)$ a $v^{(2)}$ vektor által reprezentált polinom és $0 \leq s < p$.

5. lépés. Ha $v^{(2)}$ fölhasználásával nem sikerült r részre bontani $u(x)$-et, akkor a további faktorok meghatározását az

$$\text{lnko}(v^{(k)}(x) - s, w(x)) \bmod p$$

kiszámításával folytatjuk, ahol $0 \leq s < p$, $w(x)$ pedig az összes eddig megkapott tényező, $k = 3, 4, \ldots$, mindaddig, amíg az r darab tényezőt meg nem kapjuk.

Műveletigény: ha p kicsi, akkor $O(n^3 + prn^2)$. megmutatható, hogy r átlagos száma $\ln(n)$

9.1.2. Példa

Az alábbiakban részletesen végigszámolunk egy példát a fenti algoritmus demonstrálására. Ehhez a Maple rendszert használjuk, és nem az algoritmus implementációját mutatjuk be, hanem a számolás menetét. A példában az

$$u(x) = x^8 + x^6 + 10\,x^4 + 10\,x^3 + 8\,x^2 + 2\,x + 8$$

polinom faktorizálását végezzük el $p = 13$ választással.

```
> with(PolynomialTools): with(LinearAlgebra):
> u := x^8+x^6+10*x^4+10*x^3+8*x^2+2*x+8:
> p := 13:
```

Deriváljuk $u(x)$-et és megnézzük, hogy a legnagyobb közös osztója $u(x)$-szel micsoda.

```
> du:=diff(u,x);
```

$$du := 8\,x^7 + 6\,x^5 + 40\,x^3 + 30\,x^2 + 16\,x + 2$$

```
> gcd(u, du);
```

$$1$$

Mivel 1 lett a legnagyobb közös osztó, ezért $u(x)$ négyzetmentes, és így mehetünk tovább. Előállítjuk a Q mátrixot.

```
> Q:=Matrix(8):
> Q[1]:=Transpose(Vector([1,0,0,0,0,0,0,0]));
```

$$Q_1 := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> Q[2]:=Transpose(CoefficientVector(modpol(x^13, u, x, p), x));
```

$$Q_2 := \begin{bmatrix} 2 & 1 & 7 & 11 & 10 & 12 & 5 & 11 \end{bmatrix}$$

```
> Q[3]:=Transpose(CoefficientVector(modpol(x^26, u, x, p),x));
```

$$Q_3 := \begin{bmatrix} 3 & 6 & 4 & 3 & 0 & 4 & 7 & 2 \end{bmatrix}$$

```
> Q[4]:=Transpose(CoefficientVector(modpol(x^39, u, x, p),x));
```

$$Q_4 := \begin{bmatrix} 4 & 3 & 6 & 5 & 1 & 6 & 2 & 3 \end{bmatrix}$$

```
> 39+p;
```

$$52$$

```
> Q[5]:=Transpose(CoefficientVector(modpol(x^52, u, x, p),x));
```

$$Q_5 := \begin{bmatrix} 2 & 11 & 8 & 8 & 3 & 1 & 3 & 11 \end{bmatrix}$$

```
> 52+p;
```

$$65$$

```
> Q[6]:=Transpose(CoefficientVector(modpol(x^65, u, x, p),x));
```

$$Q_6 := \begin{bmatrix} 6 & 11 & 8 & 6 & 2 & 7 & 10 & 9 \end{bmatrix}$$

```
> 65+p;
```

$$78$$

```
> Q[7]:=Transpose(CoefficientVector(modpol(x^78, u, x, p),x));
```

$$Q_7 := \begin{bmatrix} 5 & 11 & 7 & 10 & 0 & 11 & 7 & 12 \end{bmatrix}$$

```
> 78+p;
```

$$91$$

```
> Q[8]:=Transpose(CoefficientVector(modpol(x^91, u, x, p),x));
```

$$Q_8 := \begin{bmatrix} 3 & 3 & 12 & 5 & 0 & 11 & 9 & 12 \end{bmatrix}$$

```
> Q;
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 7 & 11 & 10 & 12 & 5 & 11 \\ 3 & 6 & 4 & 3 & 0 & 4 & 7 & 2 \\ 4 & 3 & 6 & 5 & 1 & 6 & 2 & 3 \\ 2 & 11 & 8 & 8 & 3 & 1 & 3 & 11 \\ 6 & 11 & 8 & 6 & 2 & 7 & 10 & 9 \\ 5 & 11 & 7 & 10 & 0 & 11 & 7 & 12 \\ 3 & 3 & 12 & 5 & 0 & 11 & 9 & 12 \end{bmatrix}$$

Megvan a Q mátrix. Most meg kell oldanunk a $v^T(Q-I)=0$ homogén lineáris egyenletrendszert, ami azt jelenti, hogy meg kell keresnünk a $(Q-I)^T$ mátrix nullterét.

```
> QIT:=Transpose(Q-IdentityMatrix(8));
```

$$QIT := \begin{bmatrix} 0 & 2 & 3 & 4 & 2 & 6 & 5 & 3 \\ 0 & 0 & 6 & 3 & 11 & 11 & 11 & 3 \\ 0 & 7 & 3 & 6 & 8 & 8 & 7 & 12 \\ 0 & 11 & 3 & 4 & 8 & 6 & 10 & 5 \\ 0 & 10 & 0 & 1 & 2 & 2 & 0 & 0 \\ 0 & 12 & 4 & 6 & 1 & 6 & 11 & 11 \\ 0 & 5 & 7 & 2 & 3 & 10 & 6 & 9 \\ 0 & 11 & 2 & 3 & 11 & 9 & 12 & 11 \end{bmatrix}$$

```
> ns:=Nullspace(QIT) mod p;
```

$$ns := \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 5 \\ 5 \\ 0 \\ 9 \\ 5 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 9 \\ 11 \\ 9 \\ 10 \\ 12 \\ 0 \\ 1 \end{bmatrix} \right\}$$

Ezeket a vektorokat, mint polinomokat kell most megvizsgálnunk.

```
> v2:=FromCoefficientVector(ns[2],x);
```

$$v2 := x^6 + 5\,x^5 + 9\,x^4 + 5\,x^2 + 5\,x$$

```
> f:=0:
> for s from 0 to p-1 do
> Gcd(u, v2-s) mod p:
> if (%<>1) then
> f:=f+1:
> faktor[f]:=%%:
> fi:
> end do:
> print("faktor"); f;
> for ff from 1 to f do
> faktor[ff];
> end do;
```

$$\text{„}faktor''$$

$$2$$

$$x^5 + 5\,x^4 + 9\,x^3 + 5\,x + 5$$

$$x^3 + 8\,x^2 + 4\,x + 12$$

Ezek alapján már 2 faktorunk megvan, de kell egy harmadik is (ezt onnan tudjuk, hogy az előbb a nulltér 3 dimenziós volt).

```
> v3:=FromCoefficientVector(ns[3],x);
```

$$v3 := x^7 + 12\,x^5 + 10\,x^4 + 9\,x^3 + 11\,x^2 + 9\,x$$

```
> for s from 0 to p-1 do
> Gcd(faktor[1], v3-s) mod p;
> end do;
```

$$1$$
$$1$$
$$1$$
$$1$$
$$1$$
$$1$$
$$x^4 + 2\,x^3 + 3\,x^2 + 4\,x + 6$$
$$1$$
$$x + 3$$
$$1$$
$$1$$

1
1

Végül nézzük meg, mit ad a Maple beépített faktorizálója (elsősorban az kimenet formátuma miatt érdekes, nyilván az eredmény ugyanaz, mint a fentiek).

```
> Factors(u) mod p;
```

$$[1, [[x+3, 1], [x^3+8\,x^2+4\,x+12, 1], [x^4+2\,x^3+3\,x^2+4\,x+6, 1]]]$$

9.2. Feladatok

1. A Rolle-féle gyöktétel szerint egész együtthatós polinom minden $x = p/q$ racionális megoldására (ahol p és q relatív prím egészek, $q \neq 0$) teljesül, hogy:

   - p olyan egész, amely osztója a_0-nak (konstans tag), és
   - q olyan egész, amely osztója a_n-nek (főegyüttható).

   A tétel felhasználható arra, hogy eldöntsük, egy polinomnak vannak-e racionális gyökei, és pozitív válasz esetén meg is keressük azokat.
   Egy példán bemutatva: az $x^3 - 7x + 6$ polinom esetén a gyökjelöltek azok a racionális számok, amelyeknek a számlálója 6 osztója, nevezője pedig 1 osztója. Adódik, hogy a jelöltek a $\pm 1, \pm 2, \pm 3$ és ± 6. Ezek $1, 2$ és -3 gyökök (az összes, mint tudjuk). A visszaellenőrzést a Horner-módszerrel elvégezhetjük.
   Írjunk egy eljárást, amely adott polinomra elvégzi az itt vázolt műveleteket.

2. Mutassuk meg, hogy $x^2 + x + 1$ irreducibilis $\mod 5$ és $\mod 29$.

3. Mutassuk meg, hogy $x^3 - a$ irreducibilis $\mod 7$ hacsak $a = 0$ vagy $a = \pm 1$.

4. A PARI/GP szofver csomag tartalmazza a fejezetben ismertetett Berlekamp algoritmus implementációját. Ez elérhető a `http://pari.math.u-bordeaux.fr/` címen. A feladatunk, hogy fedezzük fel az implementáció részleteit.

5. Kronecker módszer. Mivel az egész együtthatós polinomok faktorainak is egész együtthatós polinomoknak kell lenniük, és egy egész együtthatós polinom helyettesítési értéke egész pontokban, egész (mivel egész számok véges szorzata, véges összege egész, vagyis az az egész számok zártak szorzásra és összeadásra), így polinom értékeinek prímfaktorizációját felhasználva, véges számú elemre redukálhatjuk a lehetséges (polinom)faktorokat.
   Például, vegyük az $f(x) = x^5 + x^4 + x^2 + x + 2$ polinomot. Ha ez a polinom felbontható az egész számok felett, akkor legalább az egyik faktorának első vagy másodfokúnak kell lennie (lásd az algebra alaptételének egy megfelelő változatát). Három értékre van szükségünk ahol, hogy egy legfeljebb másodfokú polinomot egyértelműen meg tudjunk határozni. Legyenek ezek a pontok az $f(0) = 2, f(1) = 6$ és az $f(-1) = 2$. Ha a 3 közül bármelyik érték 0 akkor máris találtunk egy gyököt és így a faktorizációs tétel alapján egy faktort is. Ha viszont egyik sem 0 akkor mindegyiknek csak véges mennyiségű osztója van. Például 2 egész számok segítségével csak a következőképpen írható:

   $$1 \cdot 2, 21, (-1) \cdot (-2), \text{ vagy } (-2) \cdot (-1).$$

   Vagyis ha másodfokú faktor létezik akkor annak $1, 2, -1$ vagy -2 értékűnek kell lennie az $x = 0$ pontban, és az $x = -1$ pontban. Mivel 6-ot 8-féleképpen lehet felbontani így összesen $4 \cdot 4 \cdot 8 =$

128 kombinációs lehetőség van, amelynek fele csak előjelben különbözik így ezeket elhagyva csak 64 másodfokú polinom marad, amit le kell tesztelni. Csak ezek a lehetséges faktorai $f(x)$-nek. Ezek tesztelése szerint

$$p(x) = x^2 + x + 1$$

amit úgy kaptunk, hogy $p(0) = 1, p(1) = 3$ és $p(-1) = 1$ osztja $f(x)$-et a megfelelő pontokban. Az $f(x)$ polinomot $p(x)$-szel osztva megkapjuk a másik faktort: $q(x) = x^3 - x + 2$, így $f = pq$. Mivel kiderül, hogy mind p és q irreducibilis így f irreducibilis faktorizációja:

$$f(x) = p(x)q(x) = (x^2 + x + 1)(x^3 - x + 2).$$

Feladatunk, hogy a példa alapján készítsük el a Kronecker módszer implementációját.

10. fejezet

Programkönyvtárak

10.1. BLAS

A BLAS (Basic Linear Algebra Subrutines) egy specifikáció, amely meghatározza, hogyan kell megvalósítani alacsony szintű eljárásokat általános lineáris algebrai műveletekre, mint például vektorok összeadása, skalárral történő szorzás, skaláris szorzat, lineáris kombináció és mátrixok szorzása. Ezek lényegében de facto standard műveletek lineáris algebrai könyvtárak részére. Bár a specifikáció általános, a BLAS implementációk általában sebességre optimalizáltak egy adott architektúrára.

A referencia implementáció 1979-ben készült még FORTRAN nyelvre, megtalálható a `http://netlib.org/blas/blast-forum` címen.

A BLAS funkcionalitás szempontjából 3 eljárás halmazra bontható, amelyeket szinteknek nevezünk. Ezek egyrészt kronológiai sorrendben jönnek egymás után, ugyanakkor a számítási költségük is különbözik egymástól. A modern BLAS implementációk mindhárom szintet tartalmazzák.

BLAS-1 Az alapjai 1973-1977 között készültek. Összesen 15 (főleg) vektor műveletet tartalmaz, amelyek közül néhány például az „AXPY" ($y = \alpha x + y$), skaláris szorzat, $x = \alpha x$ alakú művelet, stb. 4 verzióban áll rendelkezésre (Single, Double, Complex, Integer), 46 eljárás.
Miért BLAS 1? Mert $\mathcal{O}(n^1)$ műveletet végzünk $\mathcal{O}(n^1)$ adaton.

BLAS-2 A BLAS-2 kifejlesztése 1984-1986 között történt. Jellemzően mátrix-vektor műveleteket tartalmaz, úgy mint a „GEMV" ($y = \alpha \cdot A \cdot x + \beta \cdot x$) és „GER" ($A = A + \alpha \cdot x \cdot yT, x = T - 1 \cdot x$) típusú műveletek. 4 verzió létezik (S/D/C/Z), 66 eljárás.
Miért BLAS-2? Mert $\mathcal{O}(n^2)$ műveletet végzünk $\mathcal{O}(n^2)$ adaton.

BLAS-3 A következő szint a BLAS-3, amelynek kifejlesztése 1987-1988 között történt. 9 (főleg) mátrix-mátrix műveletet tartalmaz, például „GEMM" típusúakat ($C = \alpha \cdot A \cdot B + \beta \cdot C, C = \alpha \cdot A \cdot AT + \beta \cdot C, B = T - 1 \cdot B$). 4 verzió lézetik (S/D/C/Z), 30 eljárás.
Miért BLAS-3? Mert $\mathcal{O}(n^3)$ műveletet végzünk $\mathcal{O}(n^2)$ adaton.

LAPACK A 'Linear Algebra Package' egy olyan eljárásgyűjtemény, amely numerikus lineáris algebrai algoritmusok implementációit tartalmazza. Többek között lineáris egyenletrendszerek megoldását, legkisebb négyzetek módszerét, sajátérték feladatok megoldását, szinguláris érték felbontást, valamint mátrix faktorizációs eljárásokat tartalmaz.

Elérhetőség Minden BLAS1/2/3 kód elérhető a `http://www.netlib.org/blas` címen. Része a standard matematikai könyvtáraknak.

10.1.1. Változatok

A BLAS tehát egy referencia, amelynek számos további implementációja készült, ezekből válogatunk néhányat.

ATLAS (Automatic Tuned Linear Algebra System) egy open source BLAS változat, automatikus tuning alapú az installláskor. Ezen tulajdonsága miatt gyakran ezzel a megvalósítás képezi az teljesítmény összehasonlítások alapját.
Letölthető a `http://math-atlas.sourceforge.net/` címről.

GotoBLAS Kazushige Goto által kézzel optimalizált assembly kód. Nagyjából 2008-ban leállt a továbbfejlesztése. Elérhető a `http://www.tacc.utexas.edu/tacc-projects/gotoblas2` címen.

OpenBLAS szintén open source, a GotoBLAS egy máig is aktívan fejlesztett és karbantartott fork-ja.
`http://www.openblas.net/`

Intel MTL az Intel cég Math Kernel Library nevű csomagja, amely nem csak BLAS implementációt, hanem többek között LAPACK könyvtárat is tartalmaz. Természetesen Intel processzorokra optimalizált kód.
Ingyenesen elérhető a `https://software.intel.com/mkl` címen.

10.2. GNU könyvtárak

10.2.1. GSL

A C nyelven írt, jelenleg 2.5 verziónál tartó Scientific Library elsődleges célja, hogy lecserélje a hagyományosan még FORTRAN nyelven implementált numerikus eljárásokat tartalmazó könyvtárakat.

A könyvtár a következő témaköröket fedi le:

Complex Numbers	Roots of Polynomials	Special Functions
Vectors and Matrices	Permutations	Combinations
Sorting	BLAS Support	Linear Algebra
CBLAS Library	Fast Fourier Transforms	Eigensystems
Random Numbers	Quadrature	Random Distributions
Quasi-Random Sequences	Histograms	Statistics
Monte Carlo Integration	N-Tuples	Differential Equations
Simulated Annealing	Numerical Differentiation	Interpolation
Series Acceleration	Chebyshev Approximations	Root-Finding
Discrete Hankel Transforms	Least-Squares Fitting	Minimization
IEEE Floating-Point	Physical Constants	Basis Splines
Wavelets	Sparse BLAS Support	Sparse Linear Algebra

A GSL a legtöbb modern, széles körben használt programozási nyelvre és környezetre elérhető. A csomag az `https://www.gnu.org/software/gsl/` oldalról tölthető le.

10.2.2. GMP

A 'GNU Multiple Precision Arithmetic Library' programkönyvtár tetszőleges pontosságú aritmetikát kínál egészekre, racionális számokra és lebegőpontos számokra egyaránt. A pontosságnak csak a használt számítógép memóriája szab határt. A fejlesztők elsődleges motivációja, hogy a GMP a leggyorsabb tetszőleges pontosságú aritmetikai könyvtár legyen. Aktívan használják kriptográfiai alkalmazásokban és általános számítógépes algebrai rendszerekben.

Elérhető a `http://gmplib.org/` oldalon.

10.2.3. MPFR

A C nyelven írt 'Multiple Precision Floating-Point Reliably' programkönyvtár a GMP rendszeren alapszik és tetszőleges pontosságú lebegőpontos számítást tesz lehetővé korrekt kerekítéssel[1] és kivételkezeléssel.

Elérhető a `http://www.mpfr.org/` oldalon.

További MPFR alapú csomagok

MPC Az MPC egy olyan C nyelven írt könyvtár, amely komplex számokra kínál tetszőleges pontosságú aritmetikát, korrekt kerekítéssel.
Elérhető a `http://www.multiprecision.org/` címen.

MPFI Az MPFI csomagban tetszőleges pontosságú lebegőpontos intervallum aritmetikai eljárásokat találunk. Használatával intervallumos befoglalásokat kapunk a numerikus számításainkra.
Elérhető a `http://mpfi.gforge.inria.fr/` oldalon.

10.3. További függvénykönyvtárak

10.3.1. MPFQ

MPFQ egy véges testek fölött végzett számításokra alkalmas könyvtár. A többi hasonló csomaghoz képest az MPFQ úgy optimalizál sebességre, hogy a felhasználónak előre rögzítenie kell bizonyos tulajdonságokat, mint például használt mod érték, majd ezt a kód fordításakor kihasználja.

Elérhető a `http://mpfq.gforge.inria.fr/` címen.

10.3.2. FFTW

A 'Fastest Fourier Transform in the West' könyvtár diszkrét Fourier transzformáció kiszámítására alkalmas, ingyenes. Ahogyan azt az X szakaszban tárgyaltuk, az eljárás műveletigénye $\mathcal{O}(n\log(n))$. Számos megvalósítást tartalmaz, amely lehetővé teszi, hogy a bemenetek előzetes vizsgálata után kiválassza az adott helyzetre leginkább alkalmas verziót a tényleges számítás elvégzésére. A kettő hatvány hosszúságú bemenetekre optimális, továbbá nagyon gyors még olyan vektorokra, amelyek hossza kis méretű prímtényezők szorzatából áll. 1999-ben elnyerte a 'Wilkinson Prize for Numerical Software' díjat.

Elérhető a `http://www.fftw.org/` oldalon.

10.3.3. NTL

Victor Shoup által C++ nyelven írt 'Number Theory Library' programcsomag számelméleti eljárások gyűjteménye. Támogatja a tetszőleges pontosságú egész és lebegőpontos aritmetikát, véges testek fölötti számításokat, vektorokkal és mátrixokkal végzett műveleteket, valamint lineáris algebrát is. Polinomok faktorizálására rendkívül gyors implementációt tartalmaz.

Ingyenesen elérhető a `http://shoup.net/ntl/` oldalon.

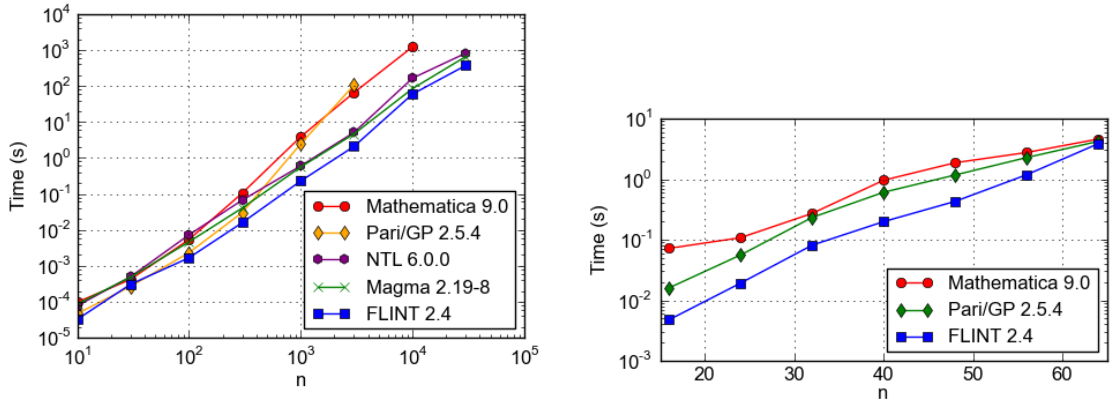
[1] azt mondjuk, hogy egy kerekítés korrekt, ha a szám és a kerekített érték között nincs az adott számítógépes környezetben elérhető ábrázolható szám (gépi szám)

10.3.4. FLINT

A 'Fast Library for Number Theory' egy C könyvtár, szintén számelméleti eljárások gyűjteménye. Ingyenesen elérhető a `http://flintlib.org/` oldalon.

Megjegyezzük, hogy Az NTL csomag szerzője készített egy (folyamatosan frissülő) összehasonlító tesztet a FLINT-tel, amely nagyon vegyes képet mutat: művelettől függően hol az egyik, hol a másik csomag a hatékonyabb, lásd a `http://shoup.net/ntl/benchmarks.pdf` dokumentumban.

Ugyanakkor a FLINT oldalon is találhatunk összehasonlító teszteket. Érdekességképpen két olyan példát ide is beemelünk, amelyek a jegyzet szempontjából relevánsak: polinom faktorizálás modulo 17 (lásd 10.1. ábra bal oldali grafikonja) és egy szóban elférő egészek faktorizálása, ahol az input 10,000 egymás utáni n bites egész $(2^{n-1}+1, \ldots, (2^{n-1}+10000$ alakban - lásd 10.1. ábra jobb oldali grafikonján.



10.1. ábra. A FLINT könyvtár futási idejének összehasonlítása néhány hasonló csomagéval. A bal oldali ábrán polinomok faktorizálása modulo 17 a feladat, míg a jobb oldali ábrán $2^{n-1}+k$ alakú egészek faktorizálása.

10.3.5. Boost

A Boost egy C++ eljárásgyűjtemény, több, mint 80 különféle könyvtárral, amelyeknek egy része tudományos számításra alkalmas algoritmusok implementációját tartalmazza. Ezek közül kiemeljük a következőket:

Boost.Multiprecision: nagypontosságú egész, racionális, lebegőpontos és komplex számokkal végzett aritmetikai műveleteket tartalmaz.

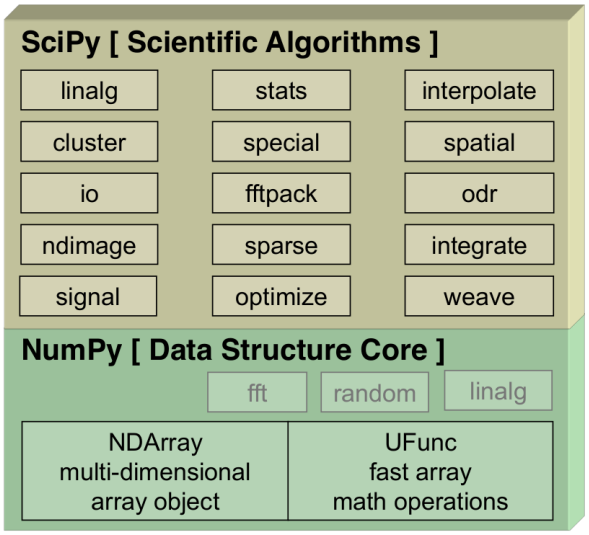
Boost.Random véletlenszám generátorokat és eloszlásfüggvényeket tartalmaz.

A könyvtár ingyenesen elérhető a `http://www.boost.org/` címen.

10.3.6. CLN

A 'Class Library for Numbers' ingyenes programkönyvtár, amely tetszőleges pontosságú aritmetikát tesz lehetővé. C++ nyelven íródott és használata lehetővé teszi az operátor felültöltést, valamint objektum orientált programozást.

Elérhető a `http://www.ginac.de/CLN/` címen.



10.2. ábra. A Numpy és a Scipy csomagok egymásra épülése és az elérhető adattípusok és matematikai függvények

10.4. Scipy és Numpy

A Numpy a Python programozási nyelvhez írt könyvtár, amely nagyméretű és magasdimenziós tömbök és mátrixok deklarálását és ezekkel végzett hatékony matematikai műveletek elvégzését támogatja. Használatával a MATLAB-hoz hasonló környezetet kapunk. Erősen támaszkodik BLAS könyvtárra (hasonlóan a MATLAB-hoz).

A Scipy tudományos és szimbolikus számítások elvégzésére alkalmas eljárásokat tartalmaz, amelyek a Numpy csomagra épülnek. Az 10.2. ábrán a két könyvtár egymásra épülése és az egyes komponensei láthatóak.

A könyvtárak a `https://www.numpy.org/` és `https://www.scipy.org/` oldalakon érhetőek el.

10.5. Rendszerek

A három óriás, MATLAB, Maple és Mathematica mellett további (ingyenes) számítógépes algebrai algoritmusokat tartalmazó és azok fejlesztését támogató rendszer létezik. Ezek közül szemezgetünk néhányat.

10.5.1. PARI/GP

A PARI/GP egy olyan számítógépes algebrai rendszer, amely elsősorban számelméleti műveletek hatékony végrehajtását támogatja. De ezen felül kezeli a mátrixokat, polinomokat, hatványsorokat, algebrai számokat, stb. A PARI részt C nyelven írták, a GP pedig egy ehhez tartozó szkript nyelv, amely interaktív végrehajtást tesz lehetővé. A GP nyelven keresztül hozzáférhetünk az összes PARI eljáráshoz.

Elérhető a `http://pari.math.u-bordeaux.fr/` címen.

10.5.2. SageMath

A SageMath egy teljesen ingyenes open source matematikai rendszer. Python-alapú interfészen keresztül érhető el, és magába foglal több másik open source matematikai programcsomagot. A mottó:

„*Creating a viable free open-source alternative to Magma, Maple, Mathematica and Matlab*”. Többek között az MPIR, MPFR, MPC, MPFI, PARI és NTL csomagokat egyaránt használja.

Elérhető a `http://www.sagemath.org/` oldalon.

Példa. Illusztrációképpen bemutatjuk, hogyan lehet a kiterjesztett euklidészi algoritmust, az Euler féle ϕ függvényt, valamint a kínai maradéktételt elérni, illetve használni SageMath-ban:

```
sage: d,u,v = xgcd(12,15)
sage: d == u*12 + v*15
True
sage: n = 2005
sage: inverse_mod(3,n)
1337
sage: 3 * 1337
4011
sage: prime_divisors(n)
[5, 401]
sage: phi = n*prod([1 - 1/p for p in prime_divisors(n)]); phi
1600
sage: euler_phi(n)
1600
sage: x = crt(2, 1, 3, 5); x
11
sage: x % 3  # x mod 3 = 2
2
sage: x % 5  # x mod 5 = 1
1
```

10.6. Online anyagok

10.6.1. NIST DLMF

A NIST 'Digital Library of Mathematical Functions' projektje a híres Abramowitz és Stegun szerzőpáros *Handbook of Mathematical Functions* című könyvének újraírását tűzte ki célul.

A honlap elérhető a `http://dlmf.nist.gov/` címen, 36 fejezetből áll. Érdekes módon egyébként nem csak matematikai függvények definícióit tartalmazza, hanem algoritmusok (általában nagyon) rövid leírását is, hivatkozással. A kapcsolódó könyv 968 oldalas.

10.6.2. Wolfram Functions

A Wolfram Functions honlap szintén matematikai függvények leírását tartalmazza: definíciókat, speciális értékeket, általános tulajdonságokat, sorként történő ábrázolásukat, integrálokat, differenciál egyenleteket, stb.

Elérhető a `http://functions.wolfram.com/` címen.

10.6.3. ESF, DDMF

Az 'Encyclopedia of Special Functions' és a 'Dynamic Dictionary of Mathematical Functions' szintén matematikai függvények gyűjteménye, a megjelenített képletek és ábrázolások azonban közvetlenül egy számítógépes algebrai rendszerből jönnek. Így a felhasználónak lehetősége van bizonyos paraméterek beállítására is és az eredmények interaktív lekérdezésére.

A két gyűjtemény elérhető a `http://algo.inria.fr/online.html` oldalról.

10.6.4. Interval Computations

A jegyzetben nem volt szó intervallum matematikáról, de azért itt megemlítjük a kapcsolódó gyűjtőoldalt, ahol bevezető anyagok, szoftverek, könyvek, kurzusok, konferenciák és alkalmazások vannak összegyűjtve.

Az portál az `http://www.cs.utep.edu/interval-comp/` címen érhető el.

10.6.5. OEIS

Az 'Online Encyclopaedia of Integer Sequences' pedig egy rendkívül gazdag és kimerítő adatbázisként szolgál a legkülönfélébb egész- és racionális sorozatokkal kapcsolatban. Az OEIS azonosító lényegében standard hivatkozás. Nemcsak cím szerint kereshető, de amennyiben a keresőjébe az illető sorozat első néhány tagját beírjuk, akkor is képes megtalálni az esetlegesen illeszkedő sorozatokat. Minden sorozatot külön oldalon tárgyal, általában néhány tagot megadva, valamint a képletét (ha van), elnevezéseit, és az ismert tulajdonságait fölsorolva szakirodalomra való hivatkozásokkal együtt.

Elérhető a `https://oeis.org/` címen.

10.7. Feladatok

1. Telepítsünk különböző BLAS könyvtárakat és teszteljük le különböző méretű mátrixokra vonatkozó műveletek (leginkább szorzás) végrehajtási ideje közötti különbségeket.

2. Implementáljunk gráf algoritmusokat mátrixok (és így BLAS könyvtárak) felhasználásával)

3. Keressük fel a 'More digits friendly' nevű verseny honlapját (`http://rnc7.loria.fr/competition.html`), ahol megtalálható az a 16 feladat, amelyet tetszőleges pontosságú aritmetikát használó programcsomagok tesztelésére javasoltak a szervezők. Válasszunk ki 2 ilyen csomagot és végezzünk el minél többet a tesztek közül.

4. Végezzünk futási idő összehasonlításokat a SageMath és a MATLAB, Maple és Mathematica csomagok között. Például: nagyméretű mátrixok szorzása, mátrixok SVD fölbontása, nagyméretű bináris mátrixok szorzása, prímtesztelés, egészek faktorizálása, nagyméretű hatványok kiszámítása és ilyen tagok összeszorzása.

5. Próbáljuk meg megkeresni a kedvenc numerikus programozási környezetünkben használt BLAS könyvtárat, és lecserélni egy másikkal.

Irodalomjegyzék

[1] D. Knuth. A számítógép programozás művészete 2, Szeminumerikus algoritmusok. Műszaki Könyvkiadó, Budapest, 1994.

[2] Gács Péter, Lovász László. Algoritmusok. Tankönyvkiadó, Budapest, 1991.

[3] A.G. Kuros. Felsőbb algebra. Tankönyvkiadó, Budapest, 1978.

[4] G. Laakmann McDowell. Cracking the coding interview (5th edition). CareerCup LLC, Palo Alto, CA, USA, 2014.

[5] A. Aziz, T-H. Lee, A. Prakash. Element of programming interviews (version 1.4.6). http://elementsofprogramminginterviews.com/

[6] Richard P. Brent, Paul Zimmermann. Modern Computer Arithmetic. https://arxiv.org/abs/1004.4710

[7] Jonathan M. Borwein, David H. Bailey, Roland Girgensohn. Experimentation in Mathematics: Computational Paths to Discovery. CRC Press, 2004.

[8] Márton Gyöngyvér. Kriptográfiai alapismeretek. Scientia Kiadó, Kolozsvár, 2008.

[9] Lax Péter. Lineáris algebra és alkalmazásai. Akadémiai Kiadó, Budapest, 2008.

[10] Johan Hastad. Advanced algorithms. KTH, 2000. https://www.nada.kth.se/~johanh/algnotes.pdf

[11] Richard Liska, Ladislav Drska, Jiri Limpouch, Milan Sinor, Michael Wester, Franz Winkler. Computer Algebra, Algorithms, Systems and Applications. 1999. http://www-troja.fjfi.cvut.cz/~liska/ca/

[12] Joachim von zur Gathen and Jürgen Gerhard. Modern Computer Algebra. Cambridge University Press, 1999.