Eötvös Loránd Tudományegyetem
Természettudományi Kar

Matematika BSc

Alkalmazott matematikus szakirány

A generikus programozás matematikai vonatkozásai

Száraz Anna

Belső témavezető:
Dr. Kiss Emil
Algebra és Számelmélet Tanszék

Külső témavezető:
Dr. Porkoláb Zoltán
Programozási Nyelvek és
Fordítóprogramok Tanszék
Informatikai Kar

Budapest, 2023

NYILATKOZAT

Név: Száraz Anna
ELTE Természettudományi Kar, szak: Matematika Bsc
NEPTUN azonosító: BBNMKF
Szakdolgozat címe: A generikus programozás matematikai vonatkozásai

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

Budapest, 2023.06.01

a hallgató aláírása

Köszönetnyilvánítások

Szeretném megköszönni a konzulensemnek, Dr. Porkoláb Zoltánnak, valamint Szalay Richárd doktorandusznak a rengeteg támogatást, amit a szakdolgozatom elkészítéséhez, illetve a szakmai fejlődésemhez nyújtottak. Továbbá hálás vagyok a belső témavezetőmnek, Dr. Kiss Emilnek, hogy a matematikával kapcsolatos kérdéseimben mindig készséggel rendelkezésre állt.

Tartalomjegyzék

Bevezetés

> „Ahhoz, hogy az ember jó programozó legyen, meg kell értenie a generikus programozás alapelveit. Ahhoz, hogy az ember megértse a generikus programozás alapelveit, meg kell értenie az absztrakciót. Ahhoz, hogy az ember megértse az absztrakciót, meg kell értenie a matematikát, amin alapszik."
>
> —Alexander A. Sztyepanov, *From Mathematics to Generic Programming [1]*

Ahogy a fenti idézet előrevetíti, ennek a szakdolgozatnak a témája a generikus programozás, illetve az azt megalapozó absztrakt algebra lesz.

Először lefektetjük az absztrakt algebrai alapokat, amelyek elengedhetetlenek a dolgozat megértéséhez, majd a generikus programozás fogalmait tekintjük át a C++ programozási nyelvben, külön tárgyalva a Standard Template Library (STL) elveit. Ezek után bemutatjuk a kapcsolatot az absztrakt algebra és a generikus programozás között egy ókorból származó szorzás algoritmuson keresztül. A kapcsolatot később tovább tárgyaljuk az euklideszi algoritmus segítségével. A generikus programozás alapelveit követve a szakdolgozat végére eljutunk a két algoritmus általánosított, és emiatt rendkívül széleskörűen alkalmazható alakjához. Mindeközben pedig megfigyeljük, hogy a matematika és az informatika miként képes kiegészíteni egymást.

A diplomamunka elsődlegesen Alexander A. Sztyepanov *From Mathematics to Generic Programming* [1] című könyvét követi, felhasználva az abban szereplő programkódokat, valamint kiegészítve saját példákkal. Az algebrai alapok forrása Dr. Kiss Emil *Bevezetés az Algebrába* [2] című könyve.

1. fejezet

Absztrakt algebrai alapok

Ebben a fejezetben a legfontosabb absztrakt algebrai fogalmakat és tételeket fogjuk bemutatni, amik alapját képezik az egész generikus programozás témakörnek. Az absztrakt algebra azért különösen hasznos, mert általános struktúrákról tudunk konkrét dolgokat belátni, anélkül, hogy ismernénk a specifikus tulajdonságaikat.

1.1. Csoportok, monoidok, félcsoportok

1.1. Definíció. *Legyen G egy nemüres halmaz. Ekkor azt mondjuk, hogy G csoport, ha értelmezett rajta egy kétváltozós $\circ$ művelet a következő tulajdonságokkal:*

- *$A \circ$ művelet asszociatív: $x \circ (y \circ z) = (x \circ y) \circ z$.*
- *$\exists$ e egységelem: $x \circ e = e \circ x = x$.*
- *Minden elemnek létezik inverze: $\forall\, x \in G$: $\exists\, x^{-1}$, amire $x \circ x^{-1} = x^{-1} \circ x = e$.*

A csoportok zártak a műveletükre, azaz $x \circ y$ nem vezet ki a halmazból. Ha a művelet kommutatív is, akkor *Abel-csoport*ról beszélünk. Ennek egy speciális esete az *additív csoport*, ahol a művelet az összeadás.

1.2. Példa. *A következők csoportot alkotnak:*

- *Egész számok az összeadásra $\mathbb{Z}^+$*
- *A $\{0, 1, ..., m-1\}$ halmaz a modulo m összeadásra: $\mathbb{Z}_m^+$*
- *A racionális számok az összeadásra: $\mathbb{Q}^+$*
- *$n \times n$-es invertálható mátrixok a szorzásra*

Bizonyos esetekben a csoport-tulajdonságoknál kevesebbet is elég megkövetelni. Például lehet, hogy az inverzre éppen nincs szükség, de minden mást meg szeretnénk tartani.

1.3. Definíció. *Legyen M egy nemüres halmaz. Ekkor azt mondjuk, hogy M monoid, ha értelmezett rajta egy kétváltozós $\circ$ művelet a következő tulajdonságokkal:*

- *$A \circ$ művelet asszociatív: $x \circ (y \circ z) = (x \circ y) \circ z$.*
- *$\exists$ e egységelem: $x \circ e = e \circ x = x$.*

1.4. Példa. *Monoidok például:*

- *Véges hosszú stringek a konkatenációra*
- *Egész számok a szorzásra: $\mathbb{Z}^{\times}$*

Ha pedig az egységelemet is elengedjük, akkor félcsoportot kapunk:

1.5. Definíció. *Legyen S egy nemüres halmaz. Ekkor azt mondjuk, hogy S félcsoport, ha értelmezett rajta egy kétváltozós $\circ$ művelet, amire teljesül az asszociativitás: $x \circ (y \circ z) = (x \circ y) \circ z$.*

1.6. Példa. *Félcsoportot alkotnak például:*

- *Pozitív egészek az összeadásra $\mathbb{Z}_{+}^{+}$*
- *Páros számok a szorzásra: $2\mathbb{Z}^{\times}$*

Vajon lehet tovább enyhíteni a követelményeket? A válasz az hogy lehet, de nem érdemes. Ha az asszociativitást is elhagyjuk –ezzel egy úgynevezett *magma*-t kapva–, akkor nincs axióma, azaz nincs mit bizonyítani. Az 1.1 táblázat összefoglalja a fent említett struktúrákat.

struktúra	asszociativitás	egységelem	inverz	kommutativitás
magma	-	-	-	-
félcsoport	+	-	-	-
monoid	+	+	-	-
csoport	+	+	+	-
Ábel-csoport	+	+	+	+

1.1. táblázat. Struktúrák.

1.7. Definíció. *Ha egy G csoportnak n eleme van, akkor azt mondjuk hogy G rendje n, ahol n lehet véges, vagy végtelen.*

1.8. Definíció. *Egy $a \in G$ csoportelem rendje n, ha n a legkisebb olyan nemnulla szám, amire $a^n = e$. Ha ilyen n nem létezik, akkor a rendje végtelen. $o(n)$-nel jelöljük.*

1.9. Állítás. *Legyen $a \in G$ elem rendje n. Ekkor az a hatványának a rendje:*

$$o(a^k) = \frac{o(a)}{(o(a), k)} = \frac{n}{(n, k)}$$

Bizonyítás. Képzeljük el, hogy egy bolha ugrál k-asával egy n szög 0-tól $n-1$-ig számozott csúcsain. Ha a 0-ról indul, hány ugrás után fog visszaérni a kiinduló helyzetbe? m ugrás után a $km \ modn$ csúcsban lesz, ami akkor egyezik meg a kiindulóhelyzettel, ha $n|km$. Tehát a legkisebb ilyen m számot keressük. Nem nehéz meggondolni, hogy

$$n \mid km \iff \frac{n}{(n,k)} \Big| \frac{k}{(n,k)} m$$

(Itt (n, k) a legnagyobb közös osztót jelöli, lásd 1.37 szakasz). Az ekvivalencia jobb oldala csak akkor tud teljesülni, ha $\frac{n}{(n,k)} \Big| m$, mivel $\frac{n}{(n,k)}$-nak és $\frac{k}{(n,k)}$-nak már csak az 1 a közös osztója. Nyilván a legkisebb pozitív ilyen m maga az $\frac{n}{(n,k)}$, tehát ennyi ugrás után ér vissza a 0-ba.
Ebben az analógiában a rend fogalma annak felelt meg, hogy a bolha mikor jut vissza először a kiindulópontba, és az a^k jelentette azt, hogy k-asával ugrál a bolha. a^k rendje a különböző hatványainak a száma, vagyis a bolha által meglátogatott csúcsok száma, ami tehát $\frac{n}{(n,k)}$. □

1.2. Részcsoportok, ciklikus csoportok

1.10. Definíció. *Legyen G egy csoport, $H \subseteq G$. Azt mondjuk, hogy H részcsoportja G-nek, ha:*

- $e \in H$
- $a \in H \implies a^{-1} \in H$
- $a, b \in H \implies a \circ b \in H$

Részcsoportokra a következő jelölést alkalmazzuk: $H \leq G$. A csoportra vonatkozó axiómákat nem kell külön feltenni, az következik abból, hogy G-re teljesülnek, és $H \subseteq G$.

Minden csoportnak van legalább 2 részcsoportja: önmaga és az a csoport, amelyik csak az egységelemet tartalmazza. Ezek a *triviális részcsoport*ok. Az egyetlen csoport, ami ez alól kivételt képez az a halmaz, amely csak az egységelemet tartalmazza.

1.11. Példa. *Néhány egyszerűbb példa:*

- $2\mathbb{Z}^+$*(a páros számok)* $\leq \mathbb{Z}^+$
- $5\mathbb{Z}^+$*(az öttel osztható számok)* $\leq \mathbb{Z}^+$
- $\mathbb{Z}^+ \leq \mathbb{Q}^+ \leq \mathbb{R}^+ \leq \mathbb{C}^+$
- $\mathbb{Q}^\times \leq \mathbb{R}^\times \leq \mathbb{C}^\times$

- *Vegyük a modulo 7 nemnulla maradékokat:* $\{1,2,3,4,5,6\}$ *és a hozzá tartozó szorzástáblát:*

$\times$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

1.2. táblázat. Modulo 7 szorzástábla.

Ekkor minden részcsportban az 1*-nek benne kell lennie, és ha* x *benne van, akkor* x^{-1} *is. Illetve minden* $y \in H$*-ra* $x \circ y \in H$*-nak is teljesülnie kell. Így a következő részcsoportokat kapjuk:* $\{1\}, \{1,6\}, \{1,2,4\}, \{1,2,3,4,5,6\}$.

1.12. Definíció. *Egy nemüres csoport ciklikus, ha*

$$\exists a \in G : \forall b \in G \; \exists n : a^n = b$$

(az n*. hatvány a csoportművelet ismételt alkalmazását jelöli). Azaz létezik olyan a elem, amit hatványozva minden elem megkapható. Ekkor azt mondjuk, hogy a generálja a csoportot.*

1.13. Példa. *A következők ciklikus csoportot alkotnak:*

- $\mathbb{Z}_+^+$*, a generátoreleme az 1*
- $\mathbb{Z}_n^+$*, a generátoreleme az 1*

1.14. Lemma. *Egy véges csoport egy elemének hatványai részcsoportot alkotnak. Más szóval: egy véges csoport minden eleme benne van egy ciklikus részcsoportban, aminek az adott elem generátora.*

1.3. Mellékosztályok, Lagrange tétele

1.15. Definíció. *Legyen* G *egy csoport,* $H \leq G$ *részcsoport. Ekkor* $a \in G$*-nek a bal oldali mellékosztálya:*

$$aH = \{g \in G \mid \exists h \in H : g = ah\}$$

Hasonlóan, a Ha jobb oldali mellékosztály. Azaz aH G *azon elemeit tartalmazza, amik megkaphatóak úgy, hogy a* H*-beli elemeket megszorozzuk* a*-val (additív csoport esetén összeadjuk vele).*

1.16. Példa. *Vegyük az egész számok additív csoportját,* $\mathbb{Z}^+$*-t, és ennek egy részcsoportját,* $3\mathbb{Z}^+$*-t. Ennek 3 különböző mellékosztálya van: a* $3n$*, a* $3n+1$ *és a* $3n+2$ *alakú számok, ahol* n *tetszőleges egész.*

A következő néhány lemma a szakasz fő tételének a bizonyításához fog kelleni:

1.17. Lemma. *Egy véges csoport tetszőleges H részcsoportjának az elemszáma megegyezik az aH mellékosztálynak az elemszámával.*

Bizonyítás. Tudjuk, hogy $S = \{s_1, \ldots, s_n\}$, akkor $aS = \{as_1, \ldots, as_n\}$ az 1.15 definíció miatt. Azt akarjuk belátni, hogy $as_i \neq as_j$. Indirekt tegyük fel, hogy aS-ben van két azonos elem: as_i és as_j. Ekkor

$$\begin{aligned} a^{-1}(as_i) &= a^{-1}(as_j) && \\ (a^{-1}a)s_i &= (a^{-1}a)s_j && \text{asszociativitás miatt} \\ es_i &= es_j && a^{-1}a = e \\ s_i &= s_j && ex = x \end{aligned}$$

Tehát ha a transzformáció után két elem egyenlő, akkor a transzformáció előtt is egyenlőek voltak. Kontrapozíció segítségével az előző állításból következik, hogy ha az inputok nem voltak egyenlőek, akkor a transzformáció után se lesznek. Tehát ha n különböző kiindulási elem volt, akkor n különböző elemet kapunk. Vagyis beláttuk, hogy $|S| = |aS|$ □

1.18. Lemma. *Egy G csoport minden eleme benne van egy H részcsoport valamelyik mellékosztályában.*

Bizonyítás. H tartalmazza az egységelemet, mivel részcsoport, így tehát $\forall a \in G : a \in aH$. □

1.19. Lemma. *A mellékosztályok vagy diszjunktak vagy azonosak (azaz ha van közös elem akkor megegyeznek).*

Bizonyítás. Tegyük fel, hogy létezik közös eleme xH és az yH mellékosztályoknak, legyen ez a. Definíció szerint ekkor $\exists u, v \in H$, hogy $a = xu, a = yv$, azaz $xu = yv$. Alkalmazzuk u inverzét az előbbire: $x = yvu^{-1}$. Vegyünk most egy tetszőleges b elemet xH-ból, ekkor definíció szerint $b = xw$, ahol $w \in H$. Behelyettesítve az előzőt, kapjuk: $b = (yvu^{-1})w$. Mivel H csoport, az asszociativitás miatt $b = w(u^{-1}vy)$. Ekkor $wu^{-1}v \in H$, mivel $u, v, w \in H$, és a csoporttulajdonság miatt u inverze is H-beli, illetve a csoport zárt a műveletére. De ekkor a mellékosztály definíciója szerint $b \in yH$. Tehát ha létezik közös eleme az xH és az yH mellékosztályoknak, akkor bármely xH-beli benne van yH-ban is. Természetesen ez fordítva is igaz, azaz xH és yH kölcsönösen egymás részhalmazai, tehát azonosak. □

1.20. Tétel (Lagrange tétele). *Egy G véges csoport minden H részcsoportjának a rendje osztja a csoport rendjét.*

Bizonyítás. Felhasználjuk a korábbi lemmákat: Az 1.18 lemma miatt tudjuk, hogy G-t lefedik a H mellékosztályai, valamint az 1.19 lemma szerint a különböző mellékosztályok diszjunktak,

jelölje m ezeknek a darabszámát. Ha $|H| = n$, akkor a mellékosztályok is mind n méretűek az 1.17 lemma miatt. Tehát G rendje nm, azaz G rendje többszöröse a H részcsoport rendjének, ami ekvivalens az állítással. □

Lagrange tételéből levezethetünk néhány hasznos következményt:

1.21. Következmény. *Egy véges csoport minden elemének a rendje osztja a csoport rendjét.*

1.22. Következmény. *Egy n-edrendű csoport tetszőleges elemét n-edik hatványra emelve az egységelemet kapjuk.*

Lagrange tételével a kis Fermat-tétel is könnyen kijön:

1.23. Tétel (**kis Fermat-tétel**). *Ha p prím,* $a^{p-1} \equiv 1 \pmod{p}$ *minden* $0 < a < p$.

Bizonyítás. Vegyük a modulo p maradékok multiplikatív csoportját: $\mathbb{Z}_p^\times$, ez $p-1$ nemnulla maradékot tartalmaz. Mivel a csoport rendje $p-1$, az 1.22 következmény miatt $a^{p-1} = e$ minden a-ra, ami a csoportban van, azaz $\forall a \in \{1, 2, \ldots, p-1\}$. A mi esetünkben az e az 1-nek felel meg, pontosabban:

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

Ezzel beláttuk a kis Fermat-tételt. □

Az 1.22 következményt felhasználva pedig az Euler tételnek a bizonyítása:

1.24. Tétel (**Euler**). *Ha a és n relatív prímek,* $a^{\varphi(n)} \equiv 1 \pmod{n}$ *minden* $0 < a < n$.

Bizonyítás. Vegyük az invertálható modulo n maradékok multiplikatív csoportját. Ennek a csoportnak a rendje $\varphi(n)$, mivel $\varphi(n)$ definíció szerint az n-hez relatív prímek számát jelenti, és ezek mind invertálhatóak. Az előző bizonyításhoz hasonlóan az 1.22 következmény miatt $a^{\varphi(n)} = e$ minden a-ra, ami a csoportban van, azaz $\forall 0 < a < n$, ahol $(a, n) = 1$. A mi esetünkben az e megint az 1-nek felel meg, azaz:

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Ezzel beláttuk Euler tételét. □

1.4. Gyűrűk, oszthatóság

1.25. Definíció. *Az R halmaz gyűrű, ha értelmezett rajta egy + (összeadás) és egy ∗ (szorzás) művelet a következő tulajdonságokkal:*

- *R Abel-csoport az összeadásra, azaz + asszociatív, kommutatív, van nullelem, és minden elemnek van ellentettje.*
- *R félcsoport a szorzásra, azaz ∗ asszociatív.*
- *Teljesül a disztributivitás:* $\forall x, y, z \in R$: $(x + y) * z = x * z + y * z$ *és* $z * (x + y) = z * x + z * y$

Ha a szorzás kommutatív is, akkor kommutatív gyűrűről, ha pedig létezik szorzásra nézve egységelem, akkor egységelemes gyűrűről beszélünk. A későbbiekben a ∗ jelet elhagyjuk, az $a * b$ szorzást ab-vel fogjuk jelölni.

1.26. Megjegyzés. *A gyűrűk tekinthetőek úgy, mint az egész számok absztrakciója, mivel bizonyos tekintetben ugyanúgy viselkednek, ami motiválta a fogalom bevezetését.*

1.27. Definíció. *Ha egy egységelemes gyűrű minden nemnulla elemének van multiplikatív inverze – azaz csoportot alkot a szorzásra – akkor ferdetestről beszélünk. Ha egy ferdetest kommutatív is a szorzásra nézve, akkor testnek nevezzük.*

1.28. Példa. *Kommutatív, egységelemes gyűrűk:*

- $\mathbb{C}$, $\mathbb{R}$, $\mathbb{Q}$ *(ezek testet is alkotnak)*
- $\mathbb{Z}$
- *valós együtthatós polinomok:* $\mathbb{R}[x]$
- *Gauss-egészek (lásd 4.4.2 szakasz)*
- *A* $\{0, 1, ..., m-1\}$ *halmaz a modulo m szorzásra és összeadásra:* $\mathbb{Z}_m$

Korábban a csoportoknál használtuk az egységelem, illetve a nullelem kifejezést, attól függően, hogy kommutatív vagy multiplikatív volt a struktúránk. A következő definíció ennek a két fogalomnak az általánosítása.

1.29. Definíció. *Legyen ∘ egy R halmazon értelmezett kétváltozós művelet. Egy* $e \in R$ *elem neutrális elem, ha* $\forall x \in R$: $e \circ x = x \circ e = x$. *Ha a művelet szorzás, akkor egységelemnek nevezzük, és 1-el jelöljük, ha pedig összeadás, akkor nullemelemnek nevezzük, és a 0 jelet használjuk.*

A csoportoknál használt inverz fogalma is megfogalmazható most már az előző definíció segítségével.

1.30. Definíció. *Legyen e neutrális eleme a $\circ$ műveletnek. Ha $x \circ y = e$, akkor x balinverze y-nak, és y jobbinverze x-nek. Ha $y \circ x = e$ is teljesül, akkor x és y egymás inverzei. Ha egy elemnek van kétoldali inverze, akkor invertálható.*

1.31. Megjegyzés. *A bal- és jobboldali inverz megkülönbözetetésére amiatt van szükség, mert nem minden esetben kommutatív művelettel van dolgunk.*

1.32. Definíció. *Legyen R gyűrű, $x, y \in R$. Ha $xy = 0$, de x és y egyike sem nulla, akkor x baloldali, y jobboldali nullosztó. Ha egy gyűrűben nincs ilyen elem, akkor nullosztómentes gyűrűről beszélünk.*

1.33. Példa. *$\mathbb{Z}_6$ gyűrűben a 2 és a 3 nullosztó.*

1.34. Definíció. *A nullosztómentes kommutatív gyűrűket integritástartománynak nevezzük. Ha egységelemes is, akkor szokásos gyűrűről beszélünk.*

1.35. Példa. *Integritástartományok például:*

- $\mathbb{Z}$
- *egész együtthatós polinomok:* $\mathbb{Z}[x]$
- *Gauss-egészek (lásd 4.4.2 szakasz)*
- *A* $\{0, 1, ..., p-1\}$ *halmaz a modulo p szorzásra és összeadásra, amennyiben p prím:* $\mathbb{Z}_p$

Most általánosítjuk az oszthatóság fogalmát, majd annak segítségével definiáljuk a kitüntetett közös osztó fogalmát.

1.36. Definíció. *Legyen R egy szokásos gyűrű, valamint $a, b \in R$. Amennyiben létezik olyan $k \in R$, hogy $ak = b$, akkor azt mondjuk, hogy a osztója b-nek (az R gyűrűben), és $a \mid b$-vel jelöljük.*

1.37. Definíció. *Legyen R szokásos gyűrű, valamint $a, b \in R$. Azt mondjuk, hogy a és b kitüntetett közös osztója egy $d \in R$ elem, ha $d \mid a$ és $d \mid b$, (azaz közös osztó), illetve $\forall c \in R$ közös osztóra teljesül, hogy $c \mid d$. A kitüntetett közös osztót a következő módon jelöljük:* (a, b).

1.38. Megjegyzés. *A kitüntetett közös osztó csak egységszeres erejéig egyértelmű, és az* (a, b) *jelölés bármelyiket jelentheti.*

1.39. Megjegyzés. *A kitüntetett közös osztó a közismert legnagyobb közös osztó fogalmát általánosítja, hiszen nincs minden gyűrűben rendezési reláció. Azaz nem mindig definiált legnagyobb elem, azonban kitüntetett közös osztó mindig létezik.*

1.40. Megjegyzés. *A kitüntetett közös osztót több elemre is lehet definiálni: olyan közös osztó, amely minden közös osztónak többszöröse. Mivel két elem közös osztóinak a halmaza megegyezik a két elem kitüntetetett közös osztójának az osztóinak a halmazával, azt kapjuk, hogy* $(a_1, \ldots, a_k) = ((\ldots((a_1, a_2), a_3)\ldots, a_{k-1}), a_k)$.

1.41. Definíció. *Két elem kitüntetett közös többszöröse az az elem, amely minden közös többszörösnek osztója. A kitüntetett közös többszöröst a következő módon jelöljük:* $[a, b]$.

A szakasz utolsó fogalma a félgyűrű. A félgyűrűk annyiban különböznek a gyűrűktől, hogy nincs megkövetelve az additív inverz létezése:

1.42. Definíció. *Az* $(S, \oplus, \otimes)$ *struktúrát félgyűrűnek nevezzük, ha a következőek teljesülnek:*

- $(S, \oplus)$ *kommutatív monoid, ahol az egységelem a* 0
- $(S, \otimes)$ *monoid, ahol az egységelem az* 1
- $\forall s \in S$: $0 \otimes s = s \otimes 0 = 0$
- *teljesül a disztributivitás:* $\forall s_1, s_2, s_3 \in S$: $s_1 \otimes (s_2 \oplus s_3) = (s_1 \otimes s_2) \oplus (s_1 \otimes s_3)$ *és* $(s_1 \oplus s_2) \otimes s_3 = (s_1 \otimes s_3) \oplus (s_2 \otimes s_3)$

A gyűrűre eddig úgy gondoltunk, mint az egész számok általánosítása. Ezt a szemléletmódot folytatva, a félgyűrű a természetes számok általánosításának tekinthető.

1.5. Permutációk

1.43. Definíció. *Az* X *véges halmazt önmagára képező bijekciókat az* X *halmaz permutációinak nevezzük. Az* n *elem összes permutációjának a halmaza csoportot alkot a kompozícióra nézve, amit szimmetrikus csoportnak nevezünk, és* S_n*-nel jelöljük. Az inverz elem az inverz permutáció, az identitás elem pedig a helybenhagyás.*

A szokásos jelölése a permutációknak a következő módon néz ki:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$$

A felső sor az eredeti állapot, az alsó pedig a permutáció eredménye.

1.44. Definíció. *Transzpozíciónak vagy cserének nevezzük azt a permutációt, ami 2 különböző* i, j *elemet cserél ki, és a többit helyben hagyja. Ezt* (i, j)*-vel jelöljük.*

1.45. Lemma. *Minden permutáció előáll cserék szorzataként.*

Bizonyítás. Van olyan transzpozíció, ami egy elemet a helyére tesz. Emiatt legfeljebb $n - 1$ cserével mind az n elemet el tudjuk vinni a kívánt helyre (azért csak $n - 1$, mert ha $n - 1$ elem jó helyen van, akkor szükségképpen az n. is). □

1.46. Definíció. *Ciklusnak nevezzük azokat az* $(x_1, x_2, \ldots, x_{k-1}, x_k)$ *permutációkat, amik* x_1*-et* x_2*-be,* x_2*-t* x_3*-ba, és így tovább,* x_{k-1}*-et* x_k*-ba és végül* x_k*-t* x_1*-be viszi. Ekkor a ciklus hossza* k*.*

1.47. Definíció. *Két ciklus diszjunkt, ha nincs közös elemük.*

1.48. Tétel. *Tegyük föl, hogy X véges halmaz. Ekkor X minden permutációja fölírható páronként diszjunkt ciklusok szorzataként.*

1.49. Állítás. *Bontsuk az f permutációt diszjunkt ciklusok szorzatára. Ekkor f rendje a ciklusfelbontásban szereplő ciklusok hosszának legkisebb közös többszöröse.*

Bizonyítás. Egy $(x_1, \ldots, x_k)$ k hosszú ciklus rendje k, mivel ha egy $\ell < k$ hatványra emeljük, akkor x_1-t $x_{\ell+1}$-be viszi, ami nem x_1, azaz $(x_1, \ldots, x_k)^\ell$ nem az identikus permutáció (helybenhagyás). De $(x_1, \ldots, x_k)^k$ már igen, vagyis k a legkisebb olyan pozitív szám, amire visszakapjuk az identitást, ez pedig pont a rend tulajdonság.

Legyen f diszjunkt ciklusokra való felbontása $f = f_1 \ldots f_m$. Ekkor

$$f^\ell = id \iff \forall i\colon f_i^\ell = id$$

mivel a ciklusok diszjunktak, azaz f_i^ℓ ugyanoda viszi a benne szereplő elemeket, mint f^ℓ, hiszen a többi ciklusban nem szerepelnek f_i elemei. Az ekvivalencia jobb oldala pontosan akkor teljesül, ha

$$\forall i\colon o(f_i) \mid \ell.$$

A legkisebb ilyen tulajdonságú pozitív l pedig pont a ciklusok rendjének, azaz ciklushosszaknak a legkisebb közös többszöröse. □

Összegzés

Ebben a fejezetben több alapvető, ámde rendkívül hasznos struktúrát tekintettünk át, amik alapjául szolgálnak a következő fejezeteknek, illetve magának az absztrakt algebrának további eredményeihez.

2. fejezet

Generikus programozás C++-ban

A generikus programozás egy programozási paradigma, amely lehetővé teszi általános adatszerkezetek és algoritmusok létrehozását, valamint azoknak a különböző adattípusokkal való felhasználását. Az elsődleges célja, hogy általános és újrafelhasználható programot hozzon létre, elkerülve a kód duplikációt, anélkül hogy az a hatékonyság rovására menne. A fogalom David Mussertől és Alexander A. Sztyepanovtól származik (1989), akik így definiálták a generikus programozást:

> „A generikus programozás azon az elképzelésen alapszik, hogy konkrét és hatékony algoritmusok absztrakciójával olyan általánosított algoritmusokhoz juthatunk, amelyeket különböző adatreprezentációkkal kombinálva rendkívül hasznos és széleskörűen felhasználható szoftvereket gyárthatunk." [3]

Fontos megjegyezni, hogy ez az ötlet nem koncentrálódik egyetlen nyelvre. Különböző programozási nyelvekben különféle eszközök állnak rendelkezésre: Haskellben, Juliaban, Scalaban, és ML-ben paraméteres polimorfizmusnak nevezzük, C++-ban illetve D-ben pedig templateeknek. Mi a C++-beli megjelenésével fogunk foglalkozni.

A középpontban generikus algoritmusok állnak, amelyek paraméterezhetőek és teljesen függetlenek az alapul szolgáló adatreprezentációtól. A típusokra vonatkozó követelmények rendszerint specifikus és hatékony algoritmusokból származnak, melyek úgynevezett koncepciók (conceptek) segítségével formalizálhatóak hasonlóan ahhoz, ahogy az absztrakt algebrában az algebrai elméleteket absztrahálják.

A koncepció egy olyan nyelvi elem, ami megmondja, hogy egyes típusok beletartoznak-e egy fogalmi halmazba. A C++-ban a koncepciók a template paraméterekre vonatkozó követelmények halmazát határozzák meg, ezzel lehetővé téve a típusellenőrzést (ezáltal csökkentve a hibalehetőséget), a pontosabb és közérthetőbb hibaüzeneteket, valamint javítják a kód olvashatóságát. C++20 óta ezt ki is lehet fejezni kóddal a `concept` és a `requires` kulcsszavak segítségével.

2.1. Expression problem

Az Expression Problem egy gyakran előkerülő programozási probléma, amely a funkcionális és az objektum-orientált programozási paradigmák kapcsán merül fel. Az alaphelyzet a következő: meg vannak írva adattípusaink, illetve azokon működő metódusaink. Van hogy az objektumok halmazát szeretnénk kibővíteni, van hogy új metódusra van szükségünk. Az előbbi könnyen megvalósítható objektum-orientált nyelvekben, az utóbbi pedig funkcionális nyelvekben. De előfordul, hogy mindkettőre szükség van, és itt merül fel a probléma [4].

Az objektum-orientált nyelvekben az absztrakciót általában öröklődéssel valósítják meg. Egy egyszerű példa, hogy különböző síkidomokaink vannak, mindegyik rendelkezik az őt jellemző szükséges információval. Ezeket szeretnénk valahogy tárolni, illetve számolni velük, első körben mondjuk a területet. Ehhez megírunk egy általános síkidom osztályt, majd leszármaztatunk belőle kör, téglalap, háromszög, stb... osztályokat. Mindegyik tárolja a maga releváns adattagját (sugár, oldalhosszok), továbbá specifikáljuk rájuk a terület kiszámítását.

```
// nyelv: C++
// ---------------- shape -----------------
class Shape {
public:
    virtual double getArea() = 0;
};

// ---------------- triangle -----------------
class Triangle : public Shape {
    double base;
    double height;
    double side_a;
    double side_c;

public:
    Triangle(double b, double h, double a, double c) : base(b), height(h), side_a(a),
        side_c(c) {}
    double getArea() override {
        return 0.5 * base * height;
    }
};

// ---------------- rectangle -----------------
class Rectangle : public Shape {
    double width;
    double height;
```

```
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() override{
        return width * height;
    }
};

// ---------------- square -----------------
class Square : public Rectangle {
public:
    Square(double s): Rectangle(s, s) {}
};
```

Ha később egy újabb objektumot kell implementálni, akkor nem okoz gondot a bővítés, egy új osztály leszármaztatható bármelyik már meglévőből. Például egy kör osztályt leszármaztatunk a bázisosztályból, vagy egy szabályos háromszöget a háromszögből. Ekkor csak az alábbi pár sort kell megírnunk, anélkül, hogy a korábbiakhoz hozzányúlnánk.

```
// ---------------- circle.h -----------------
#include <cmath>

class Circle : public Shape {
    double radius;

public:
    Circle(double r) : radius(r) {}
    double getArea() override {
        return M_PI * pow(radius, 2);
    }
};
```

A gond akkor van, ha egy már kialakult hierarchiába akarunk új metódust bevezetni, például kerület kiszámítását. Ekkor minden leszármazott osztályra külön meg kéne gondolni, hogy hogy legyen megírva, és bele kéne nyúlni a már meglévő kódba. Ez azon kívül, hogy erőforrásigényes, potenciális hibaforrás is.

```
// ---------------- shape -----------------
class Shape {
public:
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
};
```

```
// ---------------- triangle -----------------
class Triangle : public Shape {
    double base;
    double height;
    double side_a;
    double side_c;

public:
    Triangle(double b, double h, double a, double c) : base(b), height(h), side_a(a),
        side_c(c) {}
    double getArea() override {
        return 0.5 * base * height;
    }
    double getPerimeter() override {
        return base + side_a + side_c;
    }
};

// ---------------- rectangle -----------------
class Rectangle : public Shape {
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() override{
        return width * height;
    }
    double getPerimeter() override {
        return 2.0 * (width + height);
    }
};

// ---------------- square -----------------
class Square : public Rectangle {
public:
    Square(double s): Rectangle(s, s) {}
};

// ---------------- circle -----------------
#include <cmath>

class Circle : public Shape {
    double radius;
```

```
public:
    Circle(double r) : radius(r) {}
    double getArea() override {
        return M_PI * pow(radius, 2);
    }
    double getPerimeter() {
        return 2.0 * M_PI * radius;
    }
};
```

Természetesen ez egy nagyon leegyszerűsített példa; ameddig a mi tulajdonunkban vannak a file-ok, addig nem okoz különösebb nehézséget bármelyikhez hozzáadni pár új sort. De amint egy olyan osztályhierarchiát használunk, amit valaki más írt meg, nem férünk hozzá feltétlenül a bázisosztály forráskódjához, és a probléma máris bonyolultabbá válik.

A másik oldalról pedig a probléma tipikusan a funkcionális nyelvekben (Haskell, Erlang) jelentkezik. Itt is definiáljuk a háromszög, téglalap, illetve a négyzet osztályokat, illetve megírjuk hozzájuk a területet kiszámoló függvényt.

```
-- nyelv: Haskell
data Shape = Triangle Double Double Double Double
           | Rectangle Double Double
           | Square Double

getArea :: Shape -> Double
getArea (Triangle a b c mb) = 0.5 * b * mb
getArea (Rectangle w h) = w * h
getArea (Square a) = a ^^ 2
```

Egy kerület kiszámoló függvényt könnyen hozzáírhatunk:

```
getPerimeter :: Shape -> Double
getPerimeter (Triangle a b c m) = a + b + c
getPerimeter (Rectangle w h) = 2 * (w + h)
getPerimeter (Square a) = 4 * a
```

Azonban ha implementálni akarjuk a kör alakzatot, akkor módosítani kell az összes függvényt, ami szintén csak addig nem okoz problémát, ameddig a mi tulajdonunkban van a forráskód, illetve nem túl bonyolult a programunk.

```
mPi :: Double = 3.1415926535897931

data Shape = Triangle Double Double Double Double
           | Rectangle Double Double
           | Square Double
           | Circle Double

getArea :: Shape -> Double
getArea (Triangle a b c mb) = 0.5 * b * mb
getArea (Rectangle w h) = w * h
getArea (Square a) = a ^^ 2
getArea (Circle r) = mPi * r ^^ 2

getPerimeter :: Shape -> Double
getPerimeter (Triangle a b c m) = a + b + c
getPerimeter (Rectangle w h) = 2 * (w + h)
getPerimeter (Square a) = 4 * a
getPerimeter (Circle r) = 2 * mPi * r
```

A problémát Philip Wadler fogalmazta meg és nevezte el expression problemnek a Rice University's Programming Languages Team-nek (PLT) küldött levelében [5]. Ebben leírta, hogy már korábban is voltak, akik foglalkoztak a témával, de még nem igazán született rá válasz, illetve ő maga adott egy megoldást Generic Javaban, ami a Java programozási nyelv generikus típusokkal való kibővítése.

2.2. STL

A generikus programozás paradigmájának az egyik legfontosabb megjelenése, valamint az expression problem legismertebb megoldása a Standard Template Library (STL) C++-ban. Ezt Alexander A. Sztyepanov dolgozta ki és 1993-ban prezentálta az ANSI/ISO bizottság előtt, mely aztán 1994-ben el is fogadta a javaslatot.

Két fő komponense van: a konténerek (általánosított adatszerkezetek, osztályok), illetve az azokon működő algoritmusok (függvények). Ezt a két dolgot az iterátorok kötik össze, ezek segítségével lehet elérni illetve módosítani a konténerben lévő elemeket. Az iterátorok koncepciója tulajdonképpen a pointerek absztrakciója, az aktuális helyzetet (memóriacím) fejezik ki a konténerben. Több fajta létezik, például csak előre, oda-vissza, vagy random irányba többet lépni képes.

Az STL-en kívüli algoritmusok is működnek STL-es konténerekkel, illetve az STL-en kívüli konténerek is használhatóak az STL-beli algoritmusokkal. Erre jó példa a Boost C++ Libraries [8]. Ez egy modern, nyílt forráskódú, C++ szabványnak megfelelő könyvtárcsomag, amit a C++

széles körű alkalmazására fejlesztettek ki. Több mint 100 könyvtárból áll, melyek sok területet lefednek, például az algoritmusokat, a konténereket, az iterátorokat, a numerikus módszereket, a fájlok kezelését, valamint a többszálú programozást [9].

3. fejezet

Egyiptomi szorzás

Ebben a fejezetben az egyiptomi szorzást fogjuk megvizsgálni, kezdve azzal, hogy milyen problémát volt hivatott megoldani, majd pedig néhány optimalizációs lépés után rátérünk arra, hogy miként lehet általánosítani az algoritmust, szemléltetve a generikus programozás irányelveit. Végül pedig három alkalmazást fogunk látni, köszönhetően az általánosított algoritmus széleskörű felhasználhatóságának.

3.1. Motiváció

Az egyiptomi szorzás egyike az első rögzített algoritmusoknak. Az ókori egyiptomiaknak nem maradt fent sok matematikával kapcsolatos írásos emlékük ebből az időszakból, de azok a papirusztekercsek amik előkerültek, azt mutatják, hogy főleg geometriával foglalkoztak (a terület- és térfogatszámítás elengedhetetlen az építkezésekhez), illetve algebrával: szorzás, törtekkel való számolás, egyenletek megoldása. Azaz olyan problémákkal, amik a hétköznapi életben előfordulnak.

Az ókori egyiptomi számrendszer nem használt helyiértéket, se 0-t, (a római számrendszerhez hasonlóan). Emiatt rendkívül nehéz volt az $n \cdot a$ szorzatot kiszámolni. Egyik módja az, hogy összeadunk n db a-t. Ez kézzel nyilvánvalóan túl sok idő lett volna, de még a modern számítógépekkel is lassabb mint elvárható ($O(n)$ futásidejű). Az algoritmus C++-ban a következőképp fest:

```
int multiply(int n, int a) {
    if (n == 1) return a;
    return multiply(n - 1, a) + a;
}
```

A célunk a fejezet során, hogy ezt az algoritmust optimalizáljuk, majd generikussá tegyük.

3.2. Megoldás

Az általunk vizsgált algoritmus forrása az úgynevezett Rhind-papirusz, amit Ahmes írnok rögzített Kr.e. 1650 körül, és Alexander Henry Rhind skót régiségkereskedő vásárolt meg 1858-ban. Az Ahmes által leírt módszer az összeadáson, és annak az asszociativitásán alapszik, azon belül pedig a következőn: $4a = ((a + a) + a) + a = (a + a) + (a + a)$. Az ötlet az, hogy n-et felezgetjük, a-t duplázgatjuk (önmagával összeadni egy számot nem volt nehéz), majd felírjuk n-et 2-hatványok összegeként. Akkoriban persze nem változókkal írták fel az algoritmust, hanem egy példával demonstrálták, amit aztán lehetett követni más adatokkal. Ahmes a következő módon szemléltette az $n = 41$ és az $a = 59$ számok szorzatát:

1	x	59
2		118
4		236
8	x	472
16		994
32	x	1888

Jobb oldalon mindig a felette lévő érték duplája szerepel, kezdve a-val; bal oldalt egymás után a kettőhatványok, amiket addig kell felírni, hogy ne lépje túl n-et. A kiválasztott sorok pedig azt jelzik, hogy az ott lévő 2 értéket kell összeszorozni, majd ezeket összeadni: $41 \cdot 59 = (1 \cdot 59) + (8 \cdot 59) + (32 \cdot 59)$. Tulajdonképpen az x-ek a 1-es biteknek felelnek meg az n bináris számrendszerben való felírásában, de az ókori egyiptomiak nem ismerték ezt a fogalmat.

C++-ban a következőképp írható le az algoritmus:

```
int egypt_multiply(int n, int a) {
    if (n == 1) return a;
    int result = egypt_multiply(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

Köszönhetően annak, hogy a számítógépek bináris számrendszert használnak, az `odd(x)` és a `half(x)` műveletet könnyű implementálni:

```
bool odd(int n) {return n & 0x1;} //legkisebb helyierteku bit
int half(int n) {return n >> 1;} //lefele lepteto muvelet
```

Az algoritmusnak el kell döntenie, hogy n páros vagy páratlan, amit az ókori görögök (akik állítólag az egyiptomiaktól vették át a matematikai eredményeket) a következőképp definiáltak [13]:

$$n = \frac{n}{2} + \frac{n}{2} \implies \texttt{even(n)}$$

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \implies \texttt{odd(n)}$$

Páratlan számok esetén `half(n) = half(n-1)`.

Mivel felezgetjük `n`-et, $log(n)$ db rekurzív hívás történik, és minden hívásban elvégezzük az `a+a` összeadást, valamint páratlan esetekben a `result + a` összeadást. Összességében ez egy $O(log(n))$ futásidejű algoritmus, ami lényegesen jobb, mint a kezdeti `multiply`.

3.3. Optimalizálás

Most ezen algoritmus további optimalizációjával fogunk foglalkozni. A lépések tárgyalása során elsődlegesen a könyvet [1] követjük. Egy úgynevezett accumulate segédfüggvényt fogunk használni, ami az `r+na`-t fogja számolni, ahol `r` az éppen aktuális részeredmény.

```
int egypt_mul_acc(int r, int n, int a) {
    if (n == 1) return r + a;
    if (oddn(n)) {
        return egypt_mul_acc(r + a, half(n), a + a);
    } else {
        return egypt_mul_acc(r, half(n), a + a);
    }
}
```

Fejleszthetünk ezen a függvényen, ha refaktorizáljuk olyan módon, hogy tail-recursive legyen. A tail recursion egy olyan technika, amely során a rekurzív függvényhívás a legutolsó művelete a függvénynek, ezáltal csökkenti a tárhelyhasználatot és elkerüli a túlcsordulást, ugyanis ekkor nem kell eltárolni a függvényhívásokat a veremben. A teljesen tail recursive függvényeket a fordítóprogram optimalizálni tudja; ezt tail call optimalizationnak nevezzük. Lényegében az történik, hogy átalakítja egy ciklussá, elkerülve a költséges függvényhívásokat [10].
Ezt a technikát alkalmazva kapjuk a következőt:

```
int egypt_mul_acc2(int r, int n, int a) {
    if (odd(n)) {
        r = r + a;
        if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
    return egypt_mul_acc2(r, n, a);
}
```

Végül pedig könnyen iteratív alakra hozhatjuk kézzel is, ha kicseréljük a tail recursiont egy `while(true)` ciklusra:

```
int egypt_mul_acc3(int r, int n, int a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

3.1. Megjegyzés. *Az iteratív alakra hozás jól demonstálja, hogy hogy működik a tail call optimalization a gyakorlatban, vagyis hogy mit próbál elérni a fordítóprogram. Jelen esetben -O2 flaggel fordítva megfigyelhetjük, hogy az* `egypt_mul_acc2` *és az* `egypt_mul_acc3` *függvényre ugyanaz az assembly kód generálódik, azaz a fordító sikeresen el tudja végezni az optimalizálást.*

Mostantól ezt a segédfüggvényt fogjuk használni az eredeti algoritmusban:

```
int egypt_multiply2(int n, int a) {
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    //tudjuk hogy ha even(n - 1) akkor n - 1 nem lehet 1
    return egypt_mul_acc3(a, half(n - 1), a + a);
}
```

Egyelőre csak pozitív egészekre van definiálva az algoritmusunk. De miért ne lehetne a 0 vagy a negatív számok az értelmezési tartomány része, vagy akár valami teljesen más matematikai entitás, például mátrixok? Mi a legbővebb értelmezési tartománya az algoritmusunknak, és hogy tudnánk azt általánosan implementálni? A következő szakaszban erre fogunk választ adni.

3.4. Az algoritmus generalizálása

Egy hatékony és megfelelően működő generikus kód megírásához egy már helyes algoritmusból kell kiindulnunk. Ez az egyiptomiaknak köszönhetően már megvan, csak még túl specifikus. Most vizsgáljuk meg a jelenlegi adattípusainkat, hogy milyen követelményeknek kell megfelelniük, illetve hogy egymással hogy függenek össze.

Megfigyelhető, hogy azok a részek amik `n`-et érintik (`odd(n)`, `n==1`, `n=half(n)`), illetve amik `a`-t és `r`-et (`a=a+a`, `r=r+a`), diszjunktak egymástól. Azaz nem szükséges, hogy azonos típusúak legyenek, mivel úgysem keverednek. Első lépésben alakítsuk át `n`, `a` és `r` típusát tetszőlegessé template-ek segítségével.

```
template <typename A, typename N>
A multiply_accumulate(A r, N n, A a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

3.4.1. Követelmények a típusokra

Vizsgáljuk meg először az `A` típus követelményeit! A következő műveleteket kell támogatnia:

- összeadás (C++-ban: `operator+`)
- érték szerinti átadás (C++-ban: másoló konstruktor)
- értékadás (C++-ban: `operator=`)

További követelmény, hogy az összeadás legyen asszociatív (az értelmezési tartományon belül), hiszen ez volt az alapja az algoritmusnak már az ókorban is, amikor felfedezték.

C++20-szal bevezették az úgynevezett `regular type` koncepciót [11], ami az alábbiakkal rendelkezik:

- konstruktor, destruktor
- másoló konstruktor és értékadás operátor
- move konstruktor és értékadás operátor

Ide tartoznak a beépített típusok, mint például az `int`-ek.

A követelményeinknek eleget tesz, ha `A`-t `regular type`-nak definiáljuk asszociatív összeadással. Az 1.1 szakaszban áttekintett algebrai struktúrák közül a félcsoport az, ami megfelel ezeknek az elvárásoknak és a lehető legáltalánosabb: van asszociatív kétváltozós műveletete, de ennél többet nem követel meg, ahogy az az 1.1 táblázatból kiolvasható. A félcsoport egyetlen axiómájához tudnunk kell ellenőrizni az egyenlőséget, amit a regular type biztosít. Tehát arra

jutottunk, hogy A-nak additív félcsoportnak kell lennie. Ha ki akarjuk hangsúlyozni, hogy a kommutativitás nem elvárás, akkor nevezhetjük nemkommutatív additív félcsoportnak (itt a nemkommutatív szó arra utal, hogy nincs megkövetelve a kommutativitás, nem arra, hogy ne lenne megengedett).

Most vizsgáljuk meg az `N` típus követelményeit is! `N`-nek is `regular type`-nak kell lennie a következő műveletekkel:

- `half`
- `odd`
- `== 1`

Ezeken kívül az alábbiaknak kell teljesülnie, ahogy korábban megvizsgáltuk 3.2:

- `even(n)` $\implies$ `half(n) + half(n) = n`
- `odd(n)` $\implies$ `even(n-1)`
- `odd(n)` $\implies$ `half(n) = half(n-1)`

Ezek alapján azt mondhatjuk, hogy `N Integral` típusú [12], azaz egész. Ez lehet például `uint_8t`, `int_32t`, `uint_64t`, stb.

Ahogy az korábban szerepelt (lásd 2. fejezet), C++20 óta léteznek conceptek, amikkel a template-ekre vonatkozó feltételeket lehet megszorítani. De mivel mi eddig C++17-et használtunk, makrókként fogjuk kifejezni a követelményeinket a típusokkal kapcsolatban:

```
#define NonCommutativeAdditiveSemigroup typename
#define Integral typename
```

Így írhatjuk le a multiply-accumulate függvényünket teljesen generikusan:

```
template <NonCommutativeAdditiveSemigroup A, Integral N>
A multiply_accumulate_semigroup(A r, N n, A a) {
    // elofeltetel: n >= 0
    if (n == 0) return r;
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

Itt ha $n = 0$ akkor nem kell csinálnunk semmit, visszaadjuk r-et. Ennek segítségével a sima multiply algoritmus a következő:

```
#define NonCommutativeAdditiveSemigroup typename
#define Integer typename

template <NonCommutativeAdditiveSemigroup A, Integral N>
A multiply_semigroup(N n, A a) {
    // elofeltetel: n > 0
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return multiply_accumulate_semigroup(a, half(n - 1), a + a);
}
```

3.4.2. Új követelmények n-re

Eddig előfeltétel volt a multiply algoritmus esetében az, hogy n szigorúan nagyobb legyen mint 0, hiszen az ókorban is ilyen környezetben gondolkoztak. De valójában nincs szükség erre a feltételre, csak meg kell tudnunk mondani, hogy mit adjon vissza a függvény egyéb esetekben.

Ha n nulla, akkor azt az értéket kéne visszaadni, ami nem változtat az eredményen, ha ráalkalmazzuk a jelenlegi félcsoport-műveletet, azaz az összeadást. Ez az additív egységelem lenne, csakhogy a félcsoport definíciójában nem szerepel az egységelem létezése. Emiatt kell kikötni, hogy n pozitív, amennyiben ragaszkodunk a félcsoport koncepciójához.

Azonban hasznosabb megoldás lehet, ha azt mondjuk ehelyett, hogy A legyen (nemkommutatív additív) monoid: annyival több a félcsoportnál, hogy létezik egységeleme is, a mi esetünkben ez a 0, aminek segítségével azt is megengedhetjük, hogy n nulla legyen. Így ha `n == 0`, akkor visszaadjuk az A egységelemét, egyébként pedig azt csináljuk, amit eddig is.

```
template <NoncommutativeAdditiveMonoid A, Integral N>
A multiply_monoid(N n, A a) {
    // elofeltetel: n >= 0
    if (n == 0) return A{}; //A(0)
    return multiply_monoid(n, a);
}
```

3.2. Megjegyzés. *Sztyepanov eredetileg `A(0)`-val jelölte `A` egységelemét, ami koncepcionálisan helyes: az `A` típus nullelemét szeretnénk inicializálni. Sajnos azonban a C++ implementáció adott esetben veszélyes lehet. Az `std::string` szabványkönyvtári osztály (C++23 előtt) rendelkezik egy olyan konstruktorral, amely `const char*` paraméterű, és egy string literálból – például „hello” – hozza létre*

az `std::string` *objektumot. A kellemetlenséget az okozza, hogy az integer 0 literál konvertálódik* `const char*` *típusra, és pont a nullpointer értéket veszi fel, azonban az* `std::string` *osztály* `const char*` *parameterű konstruktora nem definiált működésű ebben az esetben. Így, hogy ha a példában szereplő* `A` *osztály* `std::string` *lenne, akkor a fenti kód nagy valószínűséggel futási hibához vezetne. Ebből is látszik, hogy a matematikai koncepciók biztonságos implementációja közel sem triviális feladat.*

Mi van akkor, ha negatív számmal is szeretnénk tudni szorozni? Ehhez az kell, hogy ez értelmes legyen, azaz hogy létezzen inverz. Újból módosítanunk kell a koncepciónkat: csoportra lesz szükségünk. Ahogy az az 1.1 szakaszban szerepelt, a csoportok a monoidok tulajdonságai és axiómái mellett rendelkeznek inverzzel és az ahhoz tartozó axiómával is. Mi most (nemkommutatív) additív csoportot használunk, és az inverzet mínusszal jelöljük. A hozzá tartozó axióma: $x + -x = -x + x = 0$. A következő módosítással `n` már tetszőleges egész szám lehet, és ugyanúgy működni fog az algoritmus:

```
template <NoncommutativeAdditiveGroup A, Integral N>
A multiply_group(N n, A a) {
    if (n < 0) {
        n = -n;
        a = -a;
    }
    return multiply_monoid(n, a);
}
```

3.4.3. A művelet általánosítása

A művelet általánosításának ötlete a következő megfigyelésből származik: Ha a kódban + helyett $*$-ot írunk, akkor az algoritmus $n * a$ helyett a^n-et fogja kiszámolni, ugyanis a duplázás helyett négyzetre emelés történik. Így néz ki a módosított függvény, ami az ra^n-t számolja:

```
template <MultiplicativeSemigroup A, Integral N>
A power_accumulate_semigroup(A r, N n, A a) {
    // elofeltetel: n >= 0
    if (n == 0) return r;
    while (true) {
        if (odd(n)) {
            r = r * a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a * a;
    }
}
```

Ennek segítségével pedig az a^n-t számoló függvény:

```
template <MultiplicativeSemigroup A, Integral N>
A power_semigroup(N n, A a) {
    // elofeltetel: n > 0
    while (!odd(n)) {
        a = a * a;
        n = half(n);
    }
    if (n == 1) return a;
    return power_accumulate_semigroup(a, a*a, half(n-1));
}
```

Azonban ez egyáltalán nem egy szép megoldás, hogy két, lényegében azonos kódunk van. Ráadásul létezik még sok más félcsoport is a maga asszociatív műveletével, például modulo 5 szorzás. Minden egyes esetre külön kódot írni felesleges. Helyette általánosíthatjuk magát a műveletet is, amit aztán n-hez és a-hoz hasonlóan paraméterként vár az algoritmus.

```
template <Regular A, Integral N, SemigroupOperation Op>
// requires (Domain<Op, A>)
A power_accumulate_semigroup(A r, N n, A a, Op op) {
    // elofeltetel: n >= 0
    if (n == 0) return r;
    while (true) {
        if (odd(n)) {
            r = op(r, a);
            if (n == 1) return r;
        }
        n = half(n);
        a = op(a, a);
    }
}
```

`A` most már csak Regular type -mivel nem tudjuk előre, hogy milyen félcsoport; lehet multiplikatív, additív, vagy valami teljesen más, az Op-tól függően. Mivel Op egy SemigroupOperation és az értelmezési tartománya `A`, emiatt `A` automatikusan félcsoport lesz. Ezt fejezi ki a `requires (Domain<Op, A>)` sor is (lásd 2 szakasz), de ahogy korábban is szerepelt, mi most C++17-et használunk, emiatt csak kommentként szerepel.

Az előző alapján a `power` függvény (bár már nem hatványt számol, a nevét megtartjuk):

```
template <Semigroup A, Integral N, SemigroupOperation Op>
// requires (Domain<Op, A>)
A power_semigroup(N n, A a, Op op) {
    // elofeltetel: n > 0
```

```
    while (!odd(n)) {
        a = op(a, a);
        n = half(n);
    }
    if (n == 1) return a;
    return power_accumulate_semigroup(a, op(a, a), half(n-1), op);
}
```

Ahogy az összeadásnál, most is kiterjeszthetjük az algoritmust monoidokra, ha definiáljuk az egységelemet. Ezt viszont magából a műveletből kell kikövetkeztetni, mivel nem tudjuk előre, hogy mi maga a monoid és annak a művelete.

```
template <Regular A, Integer N, MonoidOperation Op>
// requires(Domain<Op, A>)
A power_monoid(A a, N n, Op op) {
    // elofeltetel: n >= 0
    if (n == 0) return identity_element(op);
    return power_semigroup(a, n, op);
}
```

Az `identity_element` függvényt külön implementálni kell, erre két példa (additív és multiplikatív műveletekre):

```
template <NoncommutativeAdditiveMonoid T>
T identity_element(std::plus<T>) { return T(0); }
template <MultiplicativeMonoid T>
T identity_element(std::multiplies<T>) { return T(1); }
```

Csoportokra is implementálható az algoritmus, ehhez szükség lesz az inverz műveletre, ami szintén a `GroupOperation`-nek egy művelete.

```
template <Regular A, Integer N, GroupOperation Op>
// requires(Domain<Op, A>)
A power_group(A a, N n, Op op) {
    if (n < 0) {
        n = -n;
        a = inverse_operation(op)(a);
    }
    return power_monoid(a, n, op);
}
```

Két alapvető példa az inverzműveletre:

```
template <AdditiveGroup T>
std::negate<T> inverse_operation(std::plus<T>) {
    return std::negate<T>();
}
template <MultiplicativeGroup T>
reciprocal<T> inverse_operation(std::multiplies<T>) {
    return reciprocal<T>();
}
```

Az std-ben azonban csak negate függvény van beépítve, reciprocal nincs. Egy lehetséges implementációja function object-tel (olyan objektum, ami meghívható úgy, mint egy közönséges függvény, azaz van operator()-a):

```
template <MultiplicativeGroup T>
struct reciprocal {
    T operator()(const T& x) const {
        return T(1) / x;
    }
};
```

3.5. Alkalmazás

A tény, hogy egy egyszerű szorzás algoritmust ilyen szinten általánosítani lehet, önmagában nagyon szép. De ami ennél még hasznosabb, hogy számos alkalmazása is van. Például a Fibonacci számokat is ki tudjuk vele számolni hatékonyan, illetve gráfelméleti feladatokat is meg tudunk oldani a segítségével.

3.5.1. Fibonacci számok

A Fibonacci számok kiszámítására egy naiv implementáció valami ilyesmi lenne:

```
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Ugyanakkor ez a függvény rengetegszer számol ki újra bizonyos részeredményeket, ami már egy kis példán is látszik:

$$\begin{aligned} f_6 &= f_5 + f_4 \\ &= (f_4 + f_3) + (f_3 + f_2) \\ &= ((f_3 + f_2) + (f_2 + f_1)) + ((f_2 + f_1) + (f_1 + f_0)) \\ &= ((f_2 + f_1) + (f_1 + f_0)) + ((f_1 + f_0) + f_1) + ((f_1 + f_0) + f_1 + (f_1 + f_0)) \end{aligned}$$

Itt 22 összeadás történik, és az $(f_1 + f_0)$ érték 5-ször számolódik ki.

Dinamikus programozással ennél jobbat is tudunk, ehhez el kell tárolni az utolsó két eredményt:

```
int fibonacci2(int n) {
    int fst = 0, snd = 1, nt;
    if( n == 0)
       return fst;
    for(int i = 2; i <= n; i++)
    {
      nt = fst + snd;
      fst = snd;
      snd = nt;
    }
    return snd;
}
```

Így már csak $O(n)$ művelet történik. De $O(log(n))$-ben is megvalósítható mátrixok, és az korábbi szorzás algoritmus segítségével a következő módon:
Legyen $v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ azaz $\begin{bmatrix} f_1 \\ f_0 \end{bmatrix}$. Ekkor

$$v_1 = \begin{bmatrix} f_2 \\ f_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Általánosan:

$$v_i = \begin{bmatrix} f_{i+1} \\ f_i \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_i \\ f_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} v_{i-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Tehát az n-ik Fibonacci számhoz csak hatványozni kell egy mátrixot (majd egy vektorral szorozni). Mivel a mátrixok multiplikatív monoidot alkotnak (egységelem az egységmátrix, jelen esetben 2×2-es), használhatjuk az $O(log(n))$-es `power` algoritmusunkat.

3.3. Megjegyzés. *Mátrixhatványozásra ismert másik algoritmus is: bázistranszformációval diagonális alakra hozzuk, amit elemenként lehet hatványozni. Ennek a segítségével explicit képletet kapunk a*

Fibonacci sorozat tagjaira:

$$f_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}$$

Ennek a módszernek a részletes leírása a [22]-es forrásban található.

3.5.2. Gráfelméleti alkalmazások

Most 2 alkalmazást fogunk tárgyalni; mindkettő a mátrixszorzás általánosítását használja a nemnegatív, egész elemű mátrixok félgyűrűjén. A szokásos mátrixszorzás tulajdonképpen szorzatok összegzését jelenti, de a félgyűrű 2 művelete nem szükséges, hogy a hagyományos összeadás és szorzás legyen, csak a műveletekre előírt tulajdonságoknak kell teljesülniük.

Legrövidebb út irányított gráfokon

Egy klasszikus példa, hogy van egy G irányított gráf az élein nemnegatív költségfüggvénnyel (például távolságok), és meg szeretnénk találni minden csúcsból minden csúcsba a legalacsonyabb költségű (legrövidebb) utat.

A gráfot reprezentálhatjuk egy $n \times n$-es M mátrixszal, ahol az $m_{i,j}$ az i. csúcsból a j. csúcsba menő él költsége. Ha nincs ilyen él, akkor ∞-t írunk. A hagyományos mátrixszorzást a következőképpen módosítjuk: a $\oplus$ helyett minimumot, a $\otimes$ helyett összeadást veszünk.

$$b_{i,j} = min_{k=1}^{n}(m_{i,k} + m_{k,j})$$

Ekkor azt mondjuk, hogy ezt a mátrixszorzást az úgynevezett trópusi félgyűrű, azaz a $\{\mathbb{N} \cup \{\infty\}, min, +\}$ generálja. Ha a minimum helyett maximumot veszünk, azt is szokás trópusi félgyűrűnek hívni, ezzel kapcsolatban nem egységes az irodalom.

Ha ezzel az újfajta szorzással összeszorozzuk M-et önmagával $n-1$-szer, azaz $n-1$-ik hatványra emeljük, akkor pont azt kapjuk meg minden csúcsra, hogy mi a legrövidebb séta a többi csúcsba, ami legfeljebb $n-1$ élet használ. A hatványozás elvégzésére pedig alkalmazhatjuk a korábban általánosított `power` algoritmust.

Gráf tranzitív lezártja és a szociális hálók

A következő probléma szociális hálókban megkeresni, hogy egy adott ember kikkel van közvetve vagy közvetlenül kapcsolatban. Gráfok nyelvén: ha az emberek a csúcsok, élek az ismerettségek, akkor a feladat megkeresni minden csúcsra, hogy mely csúcsok érthetők el belőle irányított vagy irányítatlan úton. Azaz meg kell határozni a gráf tranzitív lezártját.

Ezt mátrixokkal úgy reprezentálhatjuk, hogy vesszük azt az M $n \times n$-es mátrixot, ahol $m_{i,j} = 1$, ha az i. és a j. csúcs között megy él, különben 0, illetve feltesszük, hogy minden ember ismeri saját magát, azaz a főátló csupa 1.

Most a mátrix szorzást a következőképpen módosítjuk:

$$\oplus := \vee(\textit{logikai VAGY})$$
$$\otimes := \wedge(\textit{logikai ÉS})$$

Tehát most a mátrixszorzást a Boolean félgyűrű, azaz az $\{\mathbb{N} \cup \{\infty\}, \vee, \wedge\}$ generálja. Ekkor ha megszorozzuk ilyen módon M-et önmagával, akkor megkapjuk egy ember barátainak barátait. Ha $n-1$-ik hatványra emeljük, akkor pedig megkapjuk a keresett tranzitív lezártat. Ehhez szintén használhatjuk a korábbi `power` algoritmusunkat.

Összegzés

Ebben a fejezetben megismertük az egyik legrégebbi, ámde rendkívül hasznos algoritmust. Néhány optimalizációs lépés után a generikus programozás elveit követve teljesen általános alakra hoztuk, mind az értelmezési tartományt, mind magát a műveletet illetően. Ennek köszönhetően pedig három különböző alkalmazásra nyílt lehetőségünk. Most áttérünk egy másik nagy jelentősségű, valamint széles körben alkalmazott algoritmusra.

4. fejezet

Euklideszi algoritmus

Ebben a fejezetben az euklideszi algoritmust fogjuk megvizsgálni a kezdeti – még csak természetes számokra definiált – alakjából kiindulva. Belátjuk, hogy helyesen működik és véges, majd két további matematikai struktúrára is általánosítjuk, aminek segítségével eljutunk a legbővebb értelmezési tartományához. Ez után kitérünk a kiterjesztett változatára, valamint megfigyeljük, hogy speciális esetekben hogyan tudjuk optimalizálni az algoritmust. Végül ismét megemlítünk néhány fontos alkamazást.

4.1. Motiváció

Az ókori görögök, mint például Pitagorasz, Thalész vagy Eratoszthenész, rendkívül fontos szerepet játszottak a matematika fejlődésében, azon belül is a geometriai, számelméleti és kombinatorikai alapok lefektetésében. Gyakorlati példák helyett elkezdtek módszeresen vizsgálni problémákat, logikai érveléssel bizonyítani állításokat. Elméleti eredményeiknek ugyanakkor számos gyakorlati alkalmazása is volt, például a csillagászatban, építészetben, illetve a hajózás tudományában.

Ami a számelméletet illeti, különösen foglalkoztatták őket a számok jellegzetes tulajdonságai, ami többek között a prímszámok felfedezéséhez vezetett. Eratoszthenész egy jól működő módszert talált fel a prímszámok kiválogatására egy nem túl nagy számhalmazból; ez az eljárás ma Eratoszthenészi szitaként ismert. Elkezdték vizsgálni, hogy két szakasznak hogyan lehet meghatározni az úgy nevezett „közös mértékét”, azaz azt a szakaszt, aminek a hossza maradék nélkül megvan a másik két szakasz hosszában. Ez vezetett el az euklideszi algoritmus felfedezéséhez.

4.2. Megoldás

Az euklideszi algoritmus egy hatékony módszer két (természetes) szám legnagyobb közös osztójának megtalálására. Euklidész az Elemek című könyvében írta le először, ami definíciók, axiómák, tételek gyűjteménye, azaz a matematika elemeinek redszerezése. A mű számos későbbi tudós (Kopernikusz, Kepler, Galilei, Newton) munkájában játszott nagy szerepet, valamint a 20. századig a matematikatanítás alapjául szolgált.

Az algoritmus azon a megfigyelésen alapszik, hogy a legnagyobb közös osztója két számnak nem változik, ha a nagyobból kivonjuk a kisebbet, és ezzel helyettesítjük a nagyobbat. Maga az eljárás ezeknek a lépéseknek az ismétlése, ameddig 2 egyenlő számot nem kapunk, amik a keresett legnagyobb közös osztót adják. C++-ban a következőképp írható le az algoritmus (egyelőre pozitív egészekre, hiszen az ókorban ilyen számokban gondolkoztak):

```
int gcd(int a, int b) {
    while (a != b) {
        if (b < a) {
            a = a - b;
        } else {
            b = b - a; //std::swap(a, b);
        }
    }
    return a;
}
```

Időszámításunk szerint az 5. század körül megjelent a nulla szám fogalma, ezáltal a maradékokat már nem az $\{1, .., n\}$ halmaz reprezentálja, hanem a $\{0, .., n-1\}$. Az algoritmusban megengedjük, hogy a 0 értéket vegyen fel, de b továbbra is szigorúan pozitív.

Felmerülhet az igény a maradék és a hányados kiszámítására is. Mivel ez a kettő nagyon hasonlóan megy, célszerű úgy megírni a kódot, hogy mind a 2 eredményt kiszámolja és vissza is adja. Ezt a programozási elvet szokás úgy nevezni, hogy „The Law of Useful Return”.

A következőképpen néz ki az algoritmus, amely a hányadost és a maradékot is visszaadja.

```
std::pair<int, int> quotient_remainder(int a, int b) {
    // elofeltetel: b > 0
    if (a < b) {
        return {0, a};
    }
    int c = largest_doubling(a, b);
    int n = 1;
    a -= c;
    while (c != b) {
        c = half(c);
```

```
            n += n;
            if (c <= a) {
                a -= c;
                n += 1;
            }
        }
        return {n, a};
    }

    //segedfuggveny:
    int largest_doubling(int a, int b) {
    // elofeltetel: b != 0
    while (a - b >= b) b = b + b;
    return b;
    }
```

Az utóbbi segédfüggvénnyel egy hatékony implementáció adható a maradék kiszámítására, amivel az eredeti problémát, a *gcd* kiszámítását tudjuk optimalizálni:

```
int remainder(int a, int b) {
    // elofeltetel: b != 0
    if (a < b) return a;
    int c = largest_doubling(a, b);
    a = a - c;
    while (c != b) {
        c = half(c);
        if (c <= a) a = a - c;
    }
    return a;
}

int gcd_remainder(int a, int b) {
    while (b != 0) {
        a = remainder(a, b);
        std::swap(a, b);
    }
    return a;
}
```

4.3. Az algoritmus helyessége

Be kell még bizonyítani, hogy az algoritmus véges, illetve, hogy tényleg a legnagyobb közös osztót adja vissza. Ebben a következő két észrevétel lesz a segítségünkre.

Az első, hogy $0 \leq (a \mod b) < b$. Vagyis minden iterációban csökken a maradék, ami pedig nemnegatív egész lehet csak, emiatt véges sok lépésben véget ér az algoritmus.

A második észrevétel, hogy minden lépésben az a/b osztás maradéka számítódik ki, ami nem más mint $r = a - bq$, ahol q a hányados. A definíció miatt $gcd(a, b)|a$ és $gcd(a, b)|b$, így $gcd(a, b)|r$ is teljesül. A fenti egyenletet átrendezve kapjuk: $a = bq + r$. Hasonló érveléssel láthatjuk, hogy $gcd(b, r)|a$. Azaz (a, b) és (b, r) pároknak ugyanazok a közös osztói, tehát ugyanaz a legnagyobb közös osztójuk is. Ezzel beláttuk a következőt:

$$a = bq + r \implies gcd(a, b) = gcd(b, r) \tag{4.1}$$

Az algoritmus minden iterációban $gcd(a, b)$-t helyettesíti $gcd(b, r)$-rel, úgy, hogy kiszámolja a maradékot, majd felcseréli az argumentumokat:

$$\begin{aligned} r_1 &= remainder(a, b) \\ r_2 &= remainder(b, r_1) \\ r_3 &= remainder(r_1, r_2) \\ &\vdots \\ r_n &= remainder(r_{n-2}, r_{n-1}) \end{aligned}$$

Definíció szerint átírva:

$$\begin{aligned} r_1 &= a - bq_1 \\ r_2 &= b - r_1 q_2 \\ r_3 &= r_1 - r_2 q_3 \\ &\vdots \\ r_n &= r_{n-2} - r_{n-1} q_n \end{aligned}$$

A 4.1-es egyenlet miatt minden sorban ugyanaz a gcd:

$$gcd(a, b) = gcd(b, r_1) = gcd(r_1, r_2) = \cdots = gcd(r_{n-1}, r_n) \overset{(1)}{=} gcd(r_n, 0) \overset{(2)}{=} r_n$$

Itt (1) következik abból az észrevételből, hogy az eljárás ezen a ponton szakad meg, (2) pedig azon a tényen alapszik, hogy tetszőleges $g \in \mathbb{Z}^+$-ra $gcd(x, 0) = x$. Így tehát az algoritmus r_n visszatérési értéke valóban a és b legnagyobb közös orsztóját adja meg.

Az algoritmusunk tehát véges, helyes és hatékony. De vajon működik másra is mint egészekre? Az egyiptomi szorzással kapcsolatban felmerülő kérdések itt is fennállnak, és a következő szakaszban ezekre fogunk választ adni.

4.4. Az euklideszi algoritmus további matematikai struktúrákra

Korábban az euklideszi algoritmust egészekre definiáltuk, és a legnagyobb közös osztó megtalálására használtuk. Azonban egy általános struktúrában nem mindig definiálható legnagyobb

elem. Emiatt mostantól az 1.37 definícióban bevezetett kitüntetett közös osztó fogalmát fogjuk használni.

4.4.1. Polinomok

A polinomokat általában (kommutatív, egységelemes) gyűrűk felett szoktuk vizsgálni. Egy R gyűrű feletti polinomot $R[x]$-el jelölünk, ami azt jelenti, hogy az együtthatók R gyűrűbeliek. Különböző gyűrűkre más az $R[x]$ számelmélete. Test fölötti polinomgyűrűkben van maradékos osztás, amit az egész számokhoz hasonlóan definiálunk: f polinom g-vel vett maradékos osztása: $f = gq + r$, ahol q a hányados, r a maradék, és $deg(r) < deg(g)$, vagy $r = 0$.

Az alábbi példával szemléltetjük ezt:
Legyen $f = x^5 + x^4 - 15x^3 + 25x^2 + 2x - 3$, $g = x^2 + 4x - 5$. Ekkor $\mathbb{R}$ felett a maradékos osztás a következő:

$$
\begin{array}{rl}
 & x^3 - 3x^2 + 2x + 2 \\
\cline{2-2}
x^2 + 4x - 5 \big) & x^5 + x^4 - 15x^3 + 25x^2 + 2x - 3 \\
 & -x^5 - 4x^4 + 5x^3 \\
\cline{2-2}
 & \quad -3x^4 - 10x^3 + 25x^2 \\
 & \quad 3x^4 + 12x^3 - 15x^2 \\
\cline{2-2}
 & \qquad 2x^3 + 10x^2 + 2x \\
 & \qquad -2x^3 - 8x^2 + 10x \\
\cline{2-2}
 & \qquad\quad 2x^2 + 12x - 3 \\
 & \qquad\quad -2x^2 - 8x + 10 \\
\cline{2-2}
 & \qquad\quad 4x + 7
\end{array}
$$

Tehát a hányados $x^3 - 3x^2 + 2x + 2$, a maradék pedig $4x + 7$.
Az euklideszi algoritmus pedig ennek a segítségével a következő eredményt adja, minden lépésben elvégezve a fenti mintájára a maradékos osztást:

$$
\begin{aligned}
x^5 + x^4 - 15x^3 + 25x^2 + 2x - 3 &= \Big(x^2 + 4x - 5\Big) \cdot \Big(x^3 - 3x^2 + 2x + 2\Big) + \Big(4x + 7\Big) \\
x^2 + 4x - 5 &= \Big(4x + 7\Big) \cdot \Big(\tfrac{1}{4}x + \tfrac{9}{16}\Big) + -\tfrac{143}{16} \\
4x + 7 &= -\tfrac{143}{16} \cdot \Big(-\tfrac{64}{143}x - \tfrac{112}{143}\Big) + 0
\end{aligned}
$$

Azaz f és g kitüntetett közös osztója $\mathbb{R}$ felett $-\frac{143}{16}$, ami $\mathbb{R}[x]$-ben egység, így a két polinom relatív prím.

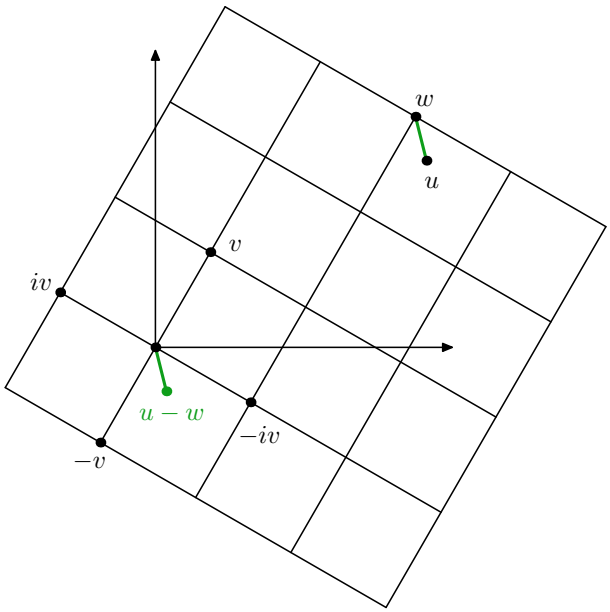
4.4.2. Gauss egészek

A Gauss egészek olyan $a + ib$ alakú komplex számok, ahol $a, b \in \mathbb{Z}$. Ahogy a neve is sugallja, Gauss vizsgálta először a komplex számok ezen speciális halmazát, és ő volt az, aki már úgy

tekintett a komplex számokra mint vektorokra, aminek köszönhető a következő felfedezése.

A Gauss egészekkel végezhető maradékos osztás, és a maradékot az alábbi módszer adja meg:

1. Legyen u és v a 2 gauss egész, aminek keressük a maradékát.
2. Vegyük azt a rácsot a komplex számsíkon, amit v, iv, $-iv$ és $-v$ feszít ki.
3. Keressük meg azt a négyzetet, ami tartalmazza u-t.
4. Keressük meg ennek a négyzetnek azt a csúcsát, ami a legközelebb van u-hoz: w
5. u-t v-vel osztva a maradék pont $u - w$ lesz.



4.1. ábra. Maradék meghatározása a Gauss-egészek körében.

4.1. Megjegyzés. *A Gauss-egészek valójában egy speciális esete az úgynevezett algebrai egészeknek: olyan komplex számok, amik előállnak egész együtthatós, 1 főegyütthatójú polinomok gyökeiként. Ezek kommutatív gyűrűt alkotnak. További példák:*

- $\mathbb{Z}$*: az* $x - c$ *polinomok gyökei*
- *n-edik egységgyökök:* $x^n - 1$ *gyökei*
- *Euler-egészek, azaz az* $a + \frac{-1+i\sqrt{3}}{2}b$ *alakú számok:* $x^3 - 1$ *polinom gyökei*
- *aranymetszés:* $\frac{1+\sqrt{5}}{2}$*:* $x^2 - x - 1$ *polinom gyöke*

4.4.3. Legbővebb értelmezési tartomány

Láthattuk, hogy az euklideszi algoritmust nem csak egészekre, hanem például polinomokra és komplex számokra is alkalmazhatjuk. Felmerül a kérdés, hogy mi a legáltalánosabb matema-

tikai struktúra, amire még szintén működik az algoritmus, azaz mi az értelmezési tartománya. A választ a következő definíció adja meg:

4.2. Definíció. *Azt mondjuk, hogy E Euklideszi gyűrű, ha teljesül, hogy:*

- *E integritástartomány*
- *értelmezve van E-n a quotient (maradék) és a remainder (hányados) művelet úgy, hogy*

$$b \neq 0 \implies a = quotient(a,b) \cdot b + remainder(a,b)$$

- *értelmezve van E-n egy nemnegatív norma:* $\|.\|$*:* $E \to \mathbb{N}$*, amire:*

$$\|x\| = 0 \iff x = 0$$
$$\|y\| \neq 0 \implies \|xy\| \geq \|x\|$$
$$\|remainder(x,y)\| < \|y\|$$

Egész számokra ez a norma az abszolútérték, Gauss-egészekre pedig a komplex norma. A polinomoknál megfelel az euklideszi normának a $deg(f) + 1$ függvény, azzal a kiegészítéssel, hogy a nullapolinomnak az euklideszi normáját nullának definiáljuk.

Az euklideszi algoritmus helyességének bizonyításánál azt a tényt használjuk ki, hogy a norma nemnegatív, egész, és végig csökken, emiatt véges az algoritmus. Tehát minden olyan entitásra, amely euklideszi gyűrűt alkot, működik az euklideszi algoritmus. Így implementálható C++-ban mostmár a lehető legáltalánosabban:

```
template <EuclideanDomain E>
E gcd(E a, E b) {
    while (b != E(0)) {
        a = remainder(a, b);
        std::swap(a, b);
    }
    return a;
}
```

Ez az általánosítási folyamat, amin a GCD algoritmus keresztül ment, tükrözi a generikus programozás alapelvét: Ahhoz, hogy egy működő és hatékony generikus kódot írjunk, először valami konkrétból kell kiindulnunk. Nem extra mechanizmusokat adunk hozzá, hanem lecsupaszítjuk az algoritmust a lényegére, eltávolítva a felesleges feltételeket.

4.5. Kiterjesztett euklideszi algoritmus

Kiterjesztett euklideszi algoritmus alatt azt értjük, ami a és b legnagyobb/kitüntetett közös osztójának a kiszámítása mellett előállítja azt $xa + yb$ alakban. Ebben a szakaszban látni fogjuk,

ilyen alak mindig létezik, illetve, hogy az x, y együtthatókat hogyan lehet meghatározni. Az előbbit a következő tétel, a Bézout azonosság mondja ki:

4.1. Tétel (**Bézout azonosság**). *Egy euklideszi gyűrűben $gcd(a, b)$ mindig felírható $ax + by$ alakban.*

Ennek a bizonyításához azonban előbb szükségünk lesz pár új algebrai fogalomra és lemmára.

4.3. Definíció. *Legyen R egy gyűrű, $I \subset R$. Ha I részcsoportot alkot az összeadásra nézve, továbbá $\forall a \in I, r \in R$ esetén teljesül, hogy $ra \in I$, akkor I-t balideálnak nevezzük. Hasonlóan, ha $ar \in I$, akkor I-t jobbideálnak nevezzük. Ha mindkettő teljesül, akkor (kétoldali) ideálnak nevezzük, illetve $I \lhd R$-vel jelöljük.*

Mivel az euklideszi gyűrűkben a szorzás kommutatív, innentől az ideál a kétoldali ideált fogja jelenteni. Ideálok például: a páros számok, valamint az egyváltozós polinomok, amiknek gyöke a 3.

4.4. Definíció. *Főideálgyűrűnek nevezzük az olyan kommutatív, egységelemes, nullosztómentes gyűrűket, amelyekben minden ideál főideál, azaz egyetlen elem többszöröseiből áll.*

4.5. Lemma. *Legyen R gyűrű, $a, b \in R$. Ekkor az $I = \{ua + vb : u, v \in R\}$ halmaz ideált alkot.*

Bizonyítás. $\{ua + vb\}$ zárt az összeadásra:

$$(u_1a + v_1b) + (u_2a + v_2b) = (u_1 + u_2)a + (v_1 + v_2)$$

$\{ua + vb\}$ zárt a tetszőleges elemmel való szorzásra:

$$\forall w \in R : w(ua + vb) = (wu)a + (wv)b$$ □

4.6. Lemma. *Egy euklideszi gyűrű minden ideálja zárt a maradékképzésre, illetve a legnagyobb közös osztó meghatározására.*

Bizonyítás. Zárt a maradékképzésre: Legyen I ideál, $a, b \in I$. A maradék és a hányados definíciója szerint:

$$remainder(a, b) = a - quotient(a, b) \cdot b$$

Az ideálok definíciója miatt $quotient(a, b) \cdot b \in I$, mivel $b \in I$. Ekkor jobb oldalt két ideálbeli elem különbsége áll, ami szintén benne van az ideálban, mivel összeadásra zárt, és az ellentett is benne kell legyen a csoporttulajdonság miatt. Tehát akkor a bal oldal is benne van az ideálban.

Zárt a gcd-re: Az euklideszi algoritmus csak olyan műveleteket használ, amik nem vezetnek ki egy ideálból: remainder operáció, összeadás, kivonás. □

4.7. Tétel. *Minden euklideszi gyűrű egyben főideálgyűrű is.*

Bizonyítás. Azt mutatjuk meg, hogy minden euklideszi gyűrűben lévő ideált generál egy specifikus, nevezetesen a(z egyik) legkisebb pozitív normával rendelkező eleme. Ilyen mindig van, mivel a norma definíciója (4.2) szerint nemnegatív és egész, ilyenek között pedig van minimális.

Vegyünk egy I ideált az euklideszi gyűrűből, legyen itt n a minimális norma. Tekintsük I egy tetszőleges a elemét. Ekkor a vagy többszöröse n-nek, vagy pedig r maradékot ad n-el osztva: $a = qn + r$ A maradék definíciója miatt ekkor $0 < ||r|| < ||n||$, ahol r is I-beli a korábbi tétel miatt. Ez viszont ellentmond annak a feltevésnek, hogy n a legkisebb norma az ideálban. Tehát csak az az eset lehetséges, hogy a n többszöröse minden $a \in I$-re. Ami pont azt jelenti, hogy egy elem generálja az ideált. □

Most már minden eszköz rendelkezésünkre áll ahhoz, hogy bebizonyítsuk a Bézout azonosságot.

4.1. Tétel (**Bézout azonosság**). *Egy euklideszi gyűrűben $gcd(a, b)$ mindig felírható $ax + by$ alakban.*

Bizonyítás. Azt fogjuk bebizonyítani, hogy ha a és b benne van az euklideszi gyűrűben, akkor az $\{xa + yb\}$ halmaz tartalmazza a $gcd(a, b)$-t. Korábban beláttuk, hogy $\{xa + yb\}$ ideált alkot. Ebben a és b is benne van, mivel $a = 1a + 0b$, és $b = 0a + 1b$. A 4.6 lemma miatt minden euklideszi gyűrűbeli ideál zárt a gcd-re, tehát a $gcd(a, b)$ is benne van az $\{xa + yb\}$ ideálban. Ez pedig ekvivalens az állítás eredeti alakjával. □

Most, hogy beláttuk, hogy a és b legnagyobb közös osztója előáll $ax + by$ alakban, próbáljuk meg megtalálni az x és y együtthatókat. Idézzük fel az euklideszi algoritmus működését, ahogy fentebb (4.3) részleteztük.

$$\begin{aligned} r_1 &= a - bq_1 \\ r_2 &= b - r_1q_2 \\ r_3 &= r_1 - r_2q_3 \\ &\vdots \\ r_n &= r_{n-2} - r_{n-1}q_n \end{aligned} \tag{4.2}$$

Átrendezve az egyenlőségeket, kapjuk:

$$\begin{aligned} a &= bq_1 + r_1 \\ b &= r_1q_2 + r_2 \\ r_1 &= r_2q_3 + r_3 \\ &\vdots \\ r_{n-2} &= r_{n-1}q_n + r_n \end{aligned}$$

Fentebb azt is beláttuk, hogy az utolsó nemnulla maradék, r_n, pont a keresett legnagyobb közös osztót adja. Vagyis azt szeretnénk belátni, hogy $gcd(a, b) = r_n = xa + yb$. Ehhez az r_i maradékok sorozatát fogjuk kifejezni a következőképpen: az algoritmus kezdeti állapotában az $x = 1$, $y = 0$, mivel

$$a = 1a + 0b$$

A következő sor is egyértelmű:

$$b = 0a + 1b$$

r_1 együtthatóit a 4.2 sor szerint szintén könnyen megkapjuk:

$$r_1 = 1a + (-q_1)b$$

Ezt behelyettesítve majd átrendezve r_2 együtthatói:

$$\begin{aligned} r_2 &= b - r_1q_2 \\ &= b - (a - q_1b)q_2 \\ &= b - aq_2 + q_2q_1b \\ &= -q_2a + (1 + q_1q_2)b \end{aligned} \tag{4.3}$$

Ezek után rekurzívan felírhatjuk az r_{i+2} együtthatóit r_i és r_{i+1} segítségével. Tegyük fel, hogy r_i és r_{i+1} már fel van írva a megfelelő lineáris kombinációkkal:

$$\begin{aligned} r_i &= x_ia + y_ib \\ r_{i+1} &= x_{i+1}a + y_{i+1}b \end{aligned}$$

Ekkor a 4.3 mintájára:

$$\begin{aligned} r_{i+2} &= r_i - r_{i+1}q_{i+2} \\ &= x_i a + y_i b - (x_{i+1}a + y_{i+1}b)q_{i+2} \\ &\vdots \\ &= (x_i - x_{i+1}q_{i+2})a + (y_i - y_{i+1}q_{i+2})b \end{aligned}$$

Tehát az $i+2$. iterációban az együtthatók:

$$\begin{aligned} x_{i+2} &= x_i - x_{i+1}q_{i+2} \\ y_{i+2} &= y_i - y_{i+1}q_{i+2} \end{aligned}$$

Megfigyelhetjük, hogy a együtthatói mindig a korábbi együtthatóiból fejezhetőek ki, és ez igaz b-re is. Így az utolsó iteráció pont megadja x-et és y-t, hogy $xa + yb = gcd(a, b)$. Mivel az x-ek nem függnek az y-októl és fordítva, $xa + yb = gcd(a, b)$ átrendezhető a következő módon, amennyiben b nem nulla:

$$y = \frac{gcd(a, b) - ax}{b}$$

Ez azért jó, mert ha kiszámoljuk a legnagyobb közös osztót és x-et a fenti módon, akkor y kiszámításához már nem kell minden köztes értékét meghatározni, hanem adódik a képletből.

Tehát most már implementálható a kiterjesztett euklideszi algoritmus, alkalmazva a korábbi `quotient_remainder` (4.2) függvényt. Egyből generikus formában célszerű megírni, kiegészítve a 4.4.3 szakaszbeli változatát.

```
template <EuclideanDomain E>
std::pair<E, E> extended_gcd(E a, E b) {
    E x0(1);
    E x1(0);
    while (b != E(0)) {
        // uj r es x kiszamolasa
        std::pair<E, E> qr = quotient_remainder(a, b);
        E x2 = x0 - qr.first * x1;
        // r es x frissitese
        x0 = x1;
        x1 = x2;
        a = b;
        b = qr.second;
    }
    return {x0, a};
}
```

4.8. Megjegyzés. *Ha a és b relatív prímek, akkor a Bézout azonosság szerint* $xa + yb = gcd(a, b) = 1$, *azaz* $xa \equiv 1 \mod b$. *Ekkor azt mondjuk, hogy a és b egymás multiplikatív inverzei mod b. Ha tehát alkalmazzuk a fenti kiterjesztett euklideszi algoritmusunkat, és a két visszaadott érték közül a második (azaz a gcd) pont 1, akkor tudjuk, hogy az első érték (az x) a multiplikatív inverze lesz modulo b.*

4.6. Speciális esetek, optimalizálás

1967-ben Josef Stein adott egy új gyorsabb verziót az euklideszi algoritmusra. Az optimálisabb futás az osztás helyett végezhető bitenkénti shift művelet hatékonyságán, illetve következő megfigyeléseken alapszik:

- $gcd(0, n) = gcd(n, 0) = gcd(n, n) = n$
- $gcd(2n, 2m) = 2 \cdot gcd(n, m)$
- $gcd(2n, 2m + 1) = gcd(n, 2m + 1)$
- $gcd(2n + 1, 2m) = gcd(2n + 1, m)$
- $gcd(2n + 1, 2(n + k) + 1) = gcd(2n + 1, k)$
- $gcd(2(n + k) + 1, 2n + 1) = gcd(2n + 1, k)$

Ezeket alkalmazva a következő algoritmust írta le:

```
int stein_gcd(int m, int n) {
    if (m < 0) m = -m;
    if (n < 0) n = -n;
    if (m == 0) return n;
    if (n == 0) return m;
    // m > 0 && n > 0
    int d_m = 0;
    while (even(m)) { m >>= 1; ++d_m;}
    int d_n = 0;
    while (even(n)) { n >>= 1; ++d_n;}
    // odd(m) && odd(n)
    while (m != n) {
        if (n > m) std::swap(n, m);
        m -= n;
        do m >>= 1; while (even(m));
    }
    // m == n
    return m << std::min(d_m, d_n);
}
```

Stein a fenti módon gyorsítani tudott egy már működő algoritmust. Kérdés, hogy ez valamilyen módon átvihető-e más matematikai struktúrákra is, ahogyan korábban már sikerült kiterjeszteni az algoritmust. A megfigyelések a 2-es szám kitüntetett szerepén alapultak. Tehát az algoritmus ezen változatának kiterjeszthethetősége azon múlik, hogy megtaláljuk az adott struktúrában azt, ami úgy viselkedik, mint az egészek körében a 2-es szám.

Kiderült, hogy a polinomok körében ezt a szerepet az x tölti be: ha kiemelhető az x, mint például az $x^4 + x^3 + 2x$ polinomból ($x(x^3 + x^2 + 2)$), akkor „párosnak" tekintjük a polinomot. Ellenkező esetben „páratlannak" nevezzük, ezek pontosan a nemnulla konstans tagú polinomok.

Stein algoritmusának gyorsaságához az is szükséges volt, hogy a könnyen tudjunk 2-vel osztani, ahol ez egy „shift" volt. Mivel azonban a polinomokat ábrázolhatjuk csak az együtthatójukkal egy listában, tulajdonképpen itt is egy shift-re van szükség.

Stein megfigyelései a következőképpen vihetőek át polinomokra:

- $gcd(0, p) = gcd(p, 0) = gcd(p, p) = p$
- $gcd(xp, xq) = x \cdot gcd(p, q)$
- $gcd(xp, xq + c) = gcd(p, xq + c)$
- $gcd(xp + c, xq) = gcd(xp + c, q)$
- $deg(p) \geq deg(q) \implies gcd(xp + c, xq + d) = gcd(p - \frac{c}{d}q, xq + d)$
- $deg(p) < deg(q) \implies gcd(xp + c, xq + d) = gcd(xp + c, q - \frac{d}{c}q)$

2000-ben Andre Weilert tovább általánosította az algoritmust Gauss egészekre azzal a megfigyeléssel, hogy itt az $1+i$ felel meg a 2-nek [6], valamint 2004-ben Agarwal és Frandsen mutatott egy olyan gyűrűt, ami nem Euklideszi gyűrű, de mégis működik az algoritmus [7].

Az általánosíthatóság minden esetben azon múlik, hogy generalizálható a paritás fogalma a kettővel való oszthatóságról egy legkisebb prímmel való oszthatóságra (lehet hogy több legkisebb is van, például a Gauss egészek körében az $1 + i, 1 - i, -1 + i, -1 - i$). A legkisebb prímmel való osztásnak pedig az volt a szerepe, hogy ilyenkor mindig nulla vagy egység a maradék. Tehát az algoritmus nem amiatt működik, mert a számítógépek kettes számrendszerben számolnak.

4.7. Alkalmazás

Az euklideszi algoritmusnak számos fontos alkalmazása van, ezek közül megemlítünk párat a teljesség igénye nélkül:

- törtek egyszerűsítése, moduláris aritmetikában az osztás végrehajtása.
- lineáris diofantoszi egyenletek megoldása: olyan $x, y \in \mathbb{Z}$ számok keresése, amik kielégítik az $ax + by = c$ egyenletet, ahol $a, b, c \in \mathbb{Z}$.
- kriptográfia, nyílt kulcsú titkosítás: az RSA algoritmusban a publikus és a privát kulcsok kiszámítása. Röviden a következő történik:
  - alkalmasan kiválasztunk két nagy prímszámot: p, q.
  - kiszámoljuk az Euler-féle φ függvényét a két prím szorzatának: $\varphi(pq) = (p-1)(q-1)$.
  - veszünk egy random pb_k számot, amire $gcd(pb_k, \varphi(pq)) = 1$.
  - a pr_k privát kulcs a publikus kulcs multiplikatív inverze (4.8) lesz modulo $\varphi(pq)$. Ehhez a lépéshez használjuk a kiterjesztett euklideszi algoritmust.
- permutáció rendje: ahogy az 1.49 állításban szerepelt, egy permutáció rendje a diszjunkt ciklusokra való felbontásában a ciklusok hosszának a legkisebb közös többszöröse. Ismeretes továbbá a következő összefüggés: $\forall a, b \in \mathbb{Z}$: $ab = (a, b) \cdot [a, b]$. Indukcióval ennek a képletnek az alapján ki lehet számolni több szám legkisebb közös többszörösét is, legnagyobb közös osztót pedig tudunk számolni az euklideszi algoritmussal.
- csoportelem hatványának a rendje: ahogy az 1.9 állításban láttuk, ennek a képlete $o(a^k) = \frac{o(a)}{(o(a), k)}$. A nevező kiszámításához ismét használhatjuk az euklideszi algoritmust.

Összegzés

Ebben a fejezetben megvizsgáltuk az euklideszi algoritmus kezdeti alakját, optimalizáltuk, illetve beláttuk, hogy helyesen működik. Ezek után általánosítani tudtuk más matematiai struktúrákra –alkalmazva generikus programozás szemléletét–, majd áttekintettük a kiterjesztett változatát, valamint azokat a speciális eseteket, amikor még hatékonyabban kiszámítható a legnagyobb közös osztó. Végül pedig láttunk pár alkalmazást is. Ezennel befejezzük az euklideszi algoritmus tárgyalását.

5. fejezet

További lehetőségek, kitekintés

A generikus programozás jelen van a konkurens programozásban is. A C++17-es szabvánnyal bevezetésre került a Parallel STL, amely lehetővé teszi a Standard Template Library algoritmusainak párhuzamos implementációját egy execution policy (végrehajtási mód) paraméter segítségével. Ezzel specifikálhatjuk, hogy a végrehajtás szekvenciálisan (`std::execution::seq`), párhuzamosan (`std::execution::par`), vagy pedig párhuzamos és rendezetlen módon (`std::execution::par_unseq`) történjen. Azonban a párhuzamos végrehajtási mód használata gyakran hibához vezethet, ha nem tartunk be bizonyos szabályokat, mint például a kommutativitást és az asszociativitást. A következő példa ezt szemlélteti [17].

Az STL-t használva így definiálnánk tipikusan egy olyan függvényt, ami kiszámítja a számok négyzetösszegét egy intervallumban:

```
auto sqrsum = [](auto s, auto val) {
    return s + val * val;
};
auto sum = std::accumulate(v.begin(), v.end(), 0ULL, sqrsum);
```

Az `std::accumulate` Parallel STL-beli változata az `std::reduce` [19] lenne, amivel így nézne ki a fenti program:

```
    auto sqrsum = [](auto s, auto val) {
        return s + val * val;
    };
    auto sum = std::reduce(std::execution::par, v.begin(), v.end(), 0ULL, sqrsum);
```

Első ránézésre jónak tűnik az implementáció, ha kipróbáljuk pár száz elemű vektorra, akkor mindkét program ugyanazt adja vissza. Mivel azonban kis példákra feleslegesen költséges lenne több szálat indítani, ezekre valójában nem kapcsol be a párhuzamos végrehajtás, így a probléma csak akkor jön elő, amikor jóval több számra próbáljuk tesztelni. És ekkor hibás

eredményt kapunk, aminek az oka, hogy az $f(s, v) = s + v^2$ függvény nem kommutatív [18].

Szerencsére bevezették az `std::transform_reduce` [20] függvényt, ami már helyesen fog működni több szálon is:

```
auto sum = std::transform_reduce(
    std::execution::par, v.begin(), v.end(), 0ULL,
    /* Reduce =*/ std::plus<>(),
    /* Transform =*/ [](auto v) { return v * v; });
```

Azonban számos még ismeretlen esetben eredményezhet hibát, ha nem figyelünk a kommutativitásra és asszociativitásra. Egyelőre még nem született teljes körű megoldás arra a problémára, hogy hogyan lehetne fordítási időben ellenőrizni az említett matematikai feltételeket, így további kutatási lehetőséget biztosít a jövőre nézve.

Összefoglalás

Ebben a szakdolgozatban betekintést nyerhettünk a matematika és az informatika kapcsolatába a generikus programozás paradigmáján keresztül, amelynek fontos alapköve az absztrakt algebra. E két fogalom ismertetése után végigkövettük, hogy hogyan lehet az egyiptomi szorzást és az euklideszi algoritmust általánosabbá tenni, ami által lehetőségünk nyílt több különböző alkalmazásra. Azt mondhatjuk tehát, hogy a generikus programozás egyfajta hidat képez a két diszciplína között azzal, hogy rámutat a matematika önmagában is szép eredményeinek alkalmazhatóságára, illetve arra, hogy miként absztrahálhatunk matematikai fogalmak segítségével már létező algoritmusokat, elkerülve ezzel a kód duplikációt, és növelve a felhasználhatóságot.

Irodalom

[1] Alexander A. Stepanov és Daniel E. Rose. *From Mathematics to Generic Programming*. Addison-Wesley Professional, 2014.

[2] Kiss Emil. *Bevezetés az algebrába*. Typotex Kiadó, 2007. URL: https://dtk.tankonyvtar.hu/handle/123456789/13048?show=full.

[3] David R. Musser és Alexander A. Stepanov. „Generic programming". (1988). URL: http://stepanovpapers.com/genprog.pdf (elérés dátuma 2023. 06. 01.).

[4] Eli Bendersky. „The Expression Problem and its solutions". (2016). URL: https://eli.thegreenplace.net/2016/the-expression-problem-and-its-solutions/ (elérés dátuma 2023. 06. 01.).

[5] Philip Wadler. „The Expression Problem". (1998). URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (elérés dátuma 2023. 06. 01.).

[6] André Weilert. „Asymptotically Fast GCD Computation in $\mathbb{Z}[i]$". (2000). URL: https://link.springer.com/chapter/10.1007/10722028_40 (elérés dátuma 2023. 06. 01.).

[7] Saurabh Agarwal és Gudmund Skovbjerg Frandsen. „Binary GCD Like Algorithms for Some Complex Quadratic Rings". (2004). URL: https://www.semanticscholar.org/paper/Binary-GCD-Like-Algorithms-for-Some-Complex-Rings-Agarwal-Frandsen/2f4ba8b6320f2d849cd165117660c3e63b6d1126 (elérés dátuma 2023. 06. 01.).

[8] *Boost C++ libraries*. URL: https://www.boost.org/ (elérés dátuma 2023. 06. 01.).

[9] Boris Schäling. *The Boost C++ Libraries*. 2023. URL: https://theboostcpplibraries.com/.

[10] Paul E. Black. *Tail recursion*. 2008. URL: https://www.nist.gov/dads/HTML/tailRecursion.html (elérés dátuma 2023. 06. 01.).

[11] *std::regular*. URL: https://en.cppreference.com/w/cpp/concepts/regular (elérés dátuma 2023. 06. 01.).

[12] *std::integral*. URL: https://en.cppreference.com/w/cpp/concepts/integral (elérés dátuma 2023. 06. 01.).

[13] Nicomachus of Gerasa. *Introduction to arithmetic*. Kr. u. 60-120. URL: https://archive.org/details/NicomachusIntroToArithmetic.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013. URL: https://www.stroustrup.com/4th.html.

[15] Aho, Sethi és Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 1986, 2006.

[16] David Vandevoorde és Nicolai M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2002. URL: http://www.tmplbook.com/.

[17] Benjámin Barth, Richárd Szalay és Zoltán Porkoláb. „Towards Safer PARALLEL STL Usage". (2022). URL: https://www.researchgate.net/publication/65801452_Towards_Safer_Parallel_STL_Usage (elérés dátuma 2023. 06. 01.).

[18] N. M. Josuttis. *C++17: The biggest traps - [C++ on Sea 2019]*. 2019. URL: http://youtube.com/watch?v=mAZyaAo3M70&t=3975 (elérés dátuma 2023. 06. 01.).

[19] *std::reduce*. URL: https://en.cppreference.com/w/cpp/algorithm/reduce (elérés dátuma 2023. 06. 01.).

[20] *std::transform_reduce*. URL: https://en.cppreference.com/w/cpp/algorithm/transform_reduce (elérés dátuma 2023. 06. 01.).

[21] *std::iota*. URL: https://en.cppreference.com/w/cpp/algorithm/iota (elérés dátuma 2023. 06. 01.).

[22] *Explicit képlet a Fibonacci-sorozatra*. URL: https://ewkiss.web.elte.hu/html/bboard/12t.n/Fibonacci.pdf (elérés dátuma 2023. 06. 01.).