MINISTERUL EDUCAȚIEI AL REPUBLICII MOLDOVA

UNIVERSITATEA DE STAT DIN TIRASPOL

FACULTATEA FIZICĂ, MATEMATICĂ ȘI TEHNOLOGII INFORMAȚIONALE

CATEDRA ALGEBRĂ, GEOMETRIE ȘI TOPOLOGIE

Domeniul general de studiu:
Științe ale Educației

Specialitatea:
Matematică și Informatică

TEZĂ DE LICENȚĂ

Tema:

ELEMENTE DE TEORIE A DIVIZIBILITĂȚII MODELATE ÎNTR-UN SISTEM DE CALCUL ELECTRONIC

Autor:
studentul ciclului I, **Dumitru Uzun**

Conducător științific:
prof. univ., dr. conf. **Valeriu Bordan**

CHIȘINĂU, 2010

CUPRINS:

Web App: `http://duzun.teologie.net/math/`

INTRODUCERE

Matematica în formele cele mai simple a apărut odată cu apariţia omului (homo sapiens = omul înţelept), fără însă să existe ştiinţa matematicii. La început oamenii foloseau doar câteva numere naturale, care erau suficiente pentru rezolvarea problemelor cu care se confruntau zi de zi (de ex., câţi au plecat şi câţi s-au întors de la vânătoare).

Descoperirea scrisului a constituit un salt enorm pentru dezvoltarea omenirii, marcând apariţia civilizaţiilor. Dar totodată scrisul a însemnat şi un salt pentru matematică. În mod natural a apărut necesitatea sistemelor de numeraţie pentru a putea număra cantităţi din ce în ce mai mari. A apărut posibilitatea înscrierii numerelor, ceea ce permitea nu doar memorarea lor pe un suport material, fie nisip, lut sau papirus, dar şi operarea cu numerele în forma scrisă. Acest fapt a extins posibilităţile folosirii noţiunii de număr şi a matematicii de atunci în general.

Totuşi primele sisteme de numeraţie operau cu o mulţime finită de numere naturale (de ex. numerele romane[1]). Adică exista o valoare maximă reprezentabilă în acel sistem, ceea ce însemna că pe măsură ce apărea necesitatea de a calcula cantităţi mai mari, trebuia de completat sistemul vechi de numeraţie cu noi simboluri pentru noi valori. Din acest motiv omenirea a fost în căutarea unui sistem de numeraţie universal, care să poată reprezenta orice valoare, oricât de mare. Aşa un sistem a fost găsit – sistemul poziţional de numeraţie[2] de care ne folosim şi astăzi. Acest sistem, pe lângă faptul că poate reprezenta valori oricât de mari, este convenabil şi pentru efectuarea operaţiilor aritmetice cu numerele scrise, ceea ce aduce un automatism în efectuarea calculelor, în sensul că având un set mic de reguli, poţi efectua calcule cu numere oricât de mari folosind doar aceste reguli. Dar totuşi calculele sunt efectuate de către om, care este o sursă „bună" de erori. Odată cu dezvoltarea societăţii (comerţul, construcţiile, ş.a.), oamenii aveau nevoie de metode sau mijloace de calcul care să permită efectuarea calculelor **rapid** şi **corect**.

---

[1] Cifrele romane sunt: I=1, V=5, X=10, L=50, C=100, D=500, M=1000. Valoarea maximă în sistemului roman este: MMMDCCCLXXXVIII = 3888

[2] Se ştie că încă babilonienii foloseau un sistem poziţional de numeraţie sexazecimal pentru calculele calendaristice. Însă sistemul lor avea un neajuns serios – nu exista cifra zero, de aceea numerele 6 şi 60 se scriau la fel, valoare înţelegându-se din context.

Cu 4000 de ani în urmă a **apărut primul calculator** numit **abac**. Iar în combinaţie cu sistemul poziţional de numeraţie, abacul a uşurat cu mult utilizarea în calcule a tabelelor de adunare şi înmulţire.

Datorită apariţiei noţiunii de logaritm, descoperită de către John Neper în 1617, a fost posibilă inventarea **riglei de calcul** de către matematicianul englez William Oughtred în 1633, care permitea efectuarea operaţiilor de înmulţire şi împărţire cu numere foarte mici sau foarte mari.

Primul calculator mecanic care efectua calcule aritmetice independent de agentul umană a fost construit de către Blaisse Pascal în anul 1642. Calculatorul lui Pascal folosea rotiţele dinţate pentru efectuarea adunării şi scăderii. În 1671, Leibnitz a construit un calculator mecanic care putea să efectueze şi operaţia de înmulţire.

Matematica secolului XVII are pretenţii mult mai mari decât calculul celor patru operaţii aritmetice. Din acest motiv, calculatorul lui Pascal şi al lui Leibnitz n-au rezolvat principalele două probleme, cea a corectitudinii rezultatului şi a timpului de calcul, pentru că necesitau intervenţia continuă a agentului uman la fiecare pas al calculului. Se simţea nevoia automatizării întregului proces de calcul pentru diferite probleme complexe, nu doar pentru efectuarea celor patru operaţii aritmetice (problemă rezolvată de aceste două exemple de calculatoare mecanice).

În 1801, Joseph Jacquard a dezvoltat primul război de ţesut capabil să repete un model în mod automat. Paşii care trebuiau urmaţi în procesul de ţesut erau determinaţi de **modelul perforaţiilor executate pe cartele de hârtie**.

Inspirat de revoluţia industrială de la sfârşitul secolului al XVII-lea şi fiind stăpânit de ideea automatismului (în procesul de producţie, şi nu numai), matematicianul şi inginerul-inventator Charles Babbage în 1837 face prima descriere a Motorului Analitic – **primul calculator digital mecanic** de scop general care a anticipat toate aspectele calculatoarelor moderne. Fiind ales în anul 1828 Profesorul Lucasian de Matematică al Universităţii Cambridge (aceeaşi funcţie ocupată de Issac Newton), Charles Babbage în 1839 părăseşte catedra pentru a se devota pe deplin creării Motorului Analitic. Dar moare în 1871 înainte de a finisa Motorul său. Însă ideile lui au marcat o nouă eră în istoria dezvoltării omenirii – **era calculatoarelor**.

În 1890, Herman Hollerith a folosit ideea reprezentării informaţiei sub forma perforaţiilor în cartele de hârtie şi a realizat un calculator utilizat pentru înregistrarea şi prelucrarea datelor din recensământul din SUA, care a durat astfel doar 3 ani.

Maşinile electromagnetice şi-au făcut apariţia în anii 1920, astfel în această perioadă au fost perfecţionate maşinile cu cartele perforate.

În anul 1928 Taushek a descoperit principiul **tamburului magnetic** pentru înregistrarea informaţiei, principiu folosit şi azi la calculatoarele PC[3], pentru memoria externă cu dischete.

Profesorul Howard Aiken de la Universitatea Harvard împreună cu specialiştii firmei IBM Corporation, în 1940 a construit **prima maşină electromecanică complexă de calcul**, numită Mark 1. Această maşină folosea relee electromagnetice controlate electronic şi folosea sistemul de introducere, stocare şi prezentare a rezultatelor pe cartele perforate.

Primul calculator integral electronic a fost creat la cererea armatei SUA in perioada 1942-1945. La baza funcţionării lui stă tehnologia lămpilor cu vid pentru controlul circuitelor electrice.

În 1944 matematicianul John von Neumann a lansat ideea programului înregistrat, pentru care o maşină de calcul trebuie să fie dotată cu un dispozitiv de memorare a datelor şi comenzilor, care trebuie să lucreze cu o viteză mare şi trebuie să permită înregistrarea simplă şi rapidă a informaţiei. Astfel au apărut noţiunile de **program de prelucrare** a **algoritmului de rezolvare** a unei probleme, a secvenţelor de comenzi şi memorare de date. Calculatoarele moderne din familia 80x86 (şi nu numai) folosesc structura propusă de Neumann.

Corporaţiei IBM i se datorează deschiderea pieţei de **calculatoare personale** (PC - IBM compatibile). Acest pas a marcat pătrunderea calculatoarelor în viaţa oamenilor de rând.

Ca un fir roşu prin istoria apariţiei maşinilor de calcul trece ideea automatizării procesului de calcul pentru economii de timp şi obţinerea rezultatelor corecte. Frumos

[3] Personal Computer – Calculator Personal sau Microcalculator (are în calitate de Unitate Centrală de Procesare un microprocesor)

a enunţat această problemă primul programator, contesa Ada Augusta, în notele sale asupra maşinii lui Babbage:

„Acele munci care fac parte din diferite ramuri ale ştiinţelor matematice, deşi la prima vedere par a fi exclusiv din domeniul intelectului, pot totuşi să fie împărţite în două secţiuni distincte, dintre care una poate fi numită mecanică, pentru că este supusă unor legi precise şi invariabile, care sunt capabile de a fi exprimate prin intermediul operaţiunilor cu materia, în timp ce cealaltă, necesitând intervenţia raţionalului, ţine în mod mai special de domeniul înţelegerii. Admiţând acest lucru, am putea propune să executăm pe mijloace mecanice ramura mecanică a acestor munci, rezervând-o pentru intelectul pur pe cea care depinde de facultăţile raţionalului. Astfel, exactitatea rigidă a acelor legi care reglementează calcule numerice trebuie adesea să fi sugerat folosirea instrumentelor materiale, fie pentru a executa toate calculele de acest fel sau doar pentru a le scurta…" (citat tradus din engleză, sursa [3])

În zilele noastre există aşa un instrument – calculatorul. Sub diferite forme şi cu diferite caracteristici, calculatoarele moderne îl însoţesc pe om în aproape toate activităţile sale, nu doar în cercetare sau pentru a efectua calcule. Calculatoarele sunt prezente în cele mai multe dispozitive electrice pe care le foloseşte omul (telefon mobil, televizor, maşină de spălat, automobil etc.). Indiferent de funcţiile îndeplinite sau de structura fizică, toate calculatoarele moderne au un aspect comun – ele pot fi programate! Iar programele sunt scrise de oameni.

După cum a observat Ada Augusta, nu toate operaţiile şi concepţiile mentale pot fi efectuate de către calculator (programate). Un calculator idealizat este o maşină Turing-completă, cu deosebirea că este limitat în memorie. Astfel, conform Conjecturii Church-Turing, calculatorul poate rezolva orice problemă bazată pe o procedură algoritmică.

Calculatoarele electronice moderne sunt discrete, de aceea operează în mod natural cu numere întregi. Pentru a înţelege mai bine modul în care un calculator operează cu numerele şi pentru a putea alcătui algoritmi cât mai optimi, este absolut indispensabilă înţelegerea implementării numerelor sistematice în lumea calculatoarelor electronice.

Mi-am propus să creez un sistem web extensibil care să permită vizitatorilor să exploreze teoria împărţirii cu rest şi implementarea unor noţiuni din această teorie în sistemele de calcul, iar programatorilor să folosească şi să extindă acest sistem pentru alte aplicaţii. Sistemul constă din două părţi mari importante:

1. bibliotecile de funcţii şi clase scrise în limbajul PHP;
2. aplicaţia web care foloseşte aceste biblioteci prin funcţii API[4].

Am ales limbajul PHP din următoarele motive: securitate, flexibilitate, accesibilitate. Aplicaţiile web sunt foarte uşor de accesat, din orice sistem de operare cu posibilităţi de navigare web. Limbajul PHP este uşor de învăţat şi de utilizat, iar codul sursă de pe server nu este accesibili vizitatorilor.

Însă aplicaţiile PHP sunt limitate în timp de execuţie şi memorie operativă disponibilă, de aceea bibliotecile trebuie să folosească la maxim instrumentele oferite de limbaj pentru optimizarea algoritmilor. În acest scop se folosesc concepţiile şi principiile descrise în lucrarea de faţă.

Această lucrare reprezintă un studiu al implementării noţiunilor teoriei împărţirii cu rest în sistemele de calcul din familia 80x86, care sunt cele mai răspândite în zilele noastre. O deosebită atenţie se acordă noţiunii de număr întreg în sistemele de calcul şi a operaţiilor cu aceste numere.

[4] Application Programmable Interface – Interfaţă Programabilă de Aplicaţie

CAPITOLUL I. TEOREMA DE BAZĂ A ARITMETICII

Introducere

În acest capitolul se studiază concepţiile de bază necesare pentru utilizarea noţiunii de număr întreg în sistemele de calcul electronice şi se prezintă unele principii folosite la crearea bibliotecilor de clase şi funcţii ale aplicaţiei web menţionate în introducere. Fiecare concept nou este însoţit de exemple şi descrieri. După caz, se compară noţiunea matematică cu cea a sistemelor de calcul, asemănările şi deosebirile dintre acestea şi specificul implementării noţiunii date pe calculator. Textul foloseşte alternativ noţiunile *Unitatea Centrală de Procesare (UCP)*[5] şi procesor, cifră binară şi *bit*[6], *cuvânt al procesorului* şi cuvânt, înţelegându-se aceeaşi noţiune.

I.1. Număr întreg (ℤ) – noţiunea matematică şi cea a sistemelor de calcul.

„Mulţimea numerelor întregi este creaţie a lui Dumnezeu,
iar restul matematicii este alcătuită de oameni”

Kroneker

Noţiunea de număr, ca şi majoritatea noţiunilor în matematică, a apărut din necesităţile practice ale omului. Primele numere descoperite de om au fost cele naturale, mai exact numerele: 1, 2, 3, … (zero încă nu exista). În antichitate, grecii au dat o interpretare şi pentru numerele negative, motivând necesitatea numerelor întregi şi astfel extinzând mulţimea numerelor cunoscute de către omenire.

La începutul secolului XX, savantul şi matematicianul Peano a axiomatizat sistemul numerelor naturale, iar sistemul numerelor întregi se obţine ca o construcţie din elemente numere naturale (orice număr întreg *z* poate fi scris ca diferenţa a două numere naturale *u*-*v*). Deci un număr întreg reprezintă o îmbinare între număr şi algoritm.

---

[5] Din eng. CPU – Central Processing Unit. Este microprocesorul central în PC-uri care execută toate operaţiile aritmetice şi logie. Conform Arhitecturii Von Newman, calculatorul trebuie sa aibă o Unitate Centrală de Procesare. Familia de calculatoare 80x86 foloseşte anume această arhitectură. Textul de faţă foloseşte noţiunea de UCP referindu-se la familia de microprocesorul 80x86.

[6] Un bit este o cifră binară reprezentată de o celulă de memorie care poate conţine una din două valori: 0 sau 1. (eng. *bit* = binary digit)

În sistemele de calcul pentru reprezentarea unui număr se foloseşte o anumită cantitate de memorie. Numerele pot fi reprezentate grafic, simbolic sau cantitativ.

- **Grafic** poate fi reprezentată practic orice notaţie folosită de către om, dar de regulă este destinată numai omului, calculatorul nefiind abil să opereze cu conţinutul semantic al acestor reprezentări.
- **Simbolic** se poate reprezenta orice număr format din litere, cifre şi alte simboluri ASCII[7], de ex. '256', '–45', 'unsprezece', '2.56E+3' etc. Ca regulă această reprezentare se foloseşte pentru afişarea numerelor sau pentru citirea de la consolă[8], dar nu şi pentru operarea cu valoarea lor.
- **Cantitativ** se pot reprezenta DOAR NUMERE NATURALE, în sensul că reprezentarea numărului în baza 2 coincide cu secvenţa de biţi din memorie, fără nici o codificare sau transformare. De aceea voi numi această reprezentare **binară**. Alte numere (precum cele întregi sau reale) sunt o combinaţie dintre un algoritm şi unul sau mai multe numere naturale. De fapt toată memoria de *n octeţi*[9] a unui calculator poate fi privită ca un număr întreg de $8\times n$ cifre binare.

Indiferent de modul cum interpretăm numerele sau de felul cum sunt reprezentate pe diferite dispozitive periferice, pentru UCP a unui calculator din familia 80x86 există un singur „fel" de numere, şi anume *numere naturale binare cu o lungime fixată*. Un astfel de număr se numeşte „**cuvânt al procesorului**" (*processor word*) sau simplu – „cuvânt". Un cuvânt al procesorului de lungimea *n* este un grup de *n* biţi (sau un număr din *n* cifre binare) care sunt procesaţi împreună. De lungimea cuvântului depinde arhitectura microprocesorului şi multe aspecte de funcţionare, de aceea lungimea cuvântului se mai numeşte „**mărimea procesorului**". Calculatoarele moderne ca regulă au UCP de 16, 32 sau 64 biţi, dar se întâlnesc şi alte mărimi.

Iată mulţimea valorilor unui cuvânt de lungimea *n*: $N_n=\{0, 1, 2, \ldots, 2^n-1\}$.

---

[7] American Standard Codification for Information Interchange – un tabel de 256 de simboluri codificate pe un octet.

[8] Sistem de dispozitive de intrare şi de ieşire (a datelor) conectat la un calculator.

[9] Un octet este o secvenţă de opt biţi consecutivi (de memorie). În familia de procesoare 80x86 este cantitatea minimă de informaţie procesată într-o singură instrucţiune.

Cel mai mare număr binar format din n cifre este 2^n-1. În tabelul de mai jos sunt prezentate valorile maxime reprezentabile pe cuvântul procesorului de lungimea n:

n	Binar	Hexazecimal	Zecimal
4	1111	F	15
8	11111111	FF	255
16	11111111 11111111	FF FF	65 535
32	11111111 11111111 11111111 11111111	FF FF FF FF	4 294 967 295
64	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111	FF FF FF FF FF FF FF FF	18 446 744 073 709 551 615

Aşadar, orice operaţie aritmetică (sau logică) efectuată de către microprocesor este limitată la operanzi de valori nu mai mari decât cele specificate în tabelul de mai sus (în funcţie de lungimea cuvântului). Mai mult ca atât, rezultatul operaţiei la fel nu trebuie să depăşească valoarea maximă reprezentabilă pe cuvântul procesorului, în caz contrar rezultatul fiind diferit de cel aşteptat. Să vedem cum UCP efectuează operaţiile aritmetice de bază:

Adunarea: Vom considera cuvântul de opt biţi (sau numere de maxim opt cifre binare). Valoarea maximă conţinută într-un cuvânt este $1111\ 1111_2 = 255_{10}$. Care va fi rezultatul operaţiei „255 + 1”? Corect matematic ar fi $256 = 2^8$, care este un număr binar de nouă cifre – $1\ 0000\ 0000_2$ – şi de aceea nu „încape” în cuvântul procesorului de capacitate opt biţi. Din acest motiv ultima cifră care este „1” (în ordinea efectuării operaţiei de adunare) se ignoră, rezultatul obţinut fiind „0000 0000” (valoarea *zero*), ceea ce ne îndreptăţeşte să scriem egalitatea: $255 + 1 = 0$. Pentru a face distincţie între egalitatea matematică şi egalitatea numerelor în calculator, pe ultima o voi nota prin „==”, care este operatorul de comparaţie în mai multe limbaje de programare precum C, PHP, JS, ş.a. Astfel $255 + 1 == 0$ şi $255 + 1 = 256$. Simbolul „==” la fel denotă o egalitate matematică a valorilor, însă se compară valorile care se conţin în cuvântul procesorului dat (sau variabila dată).

Cifra ignorată „1” se numeşte **bitul de transport** (*carry bit*), iar în situaţia descrisă mai sus operaţia se numeşte **cu transport**. Uşor se observă că la efectuarea operaţiei de adunare transportul poate fi doar 1 sau 0 (adică poate lipsi).

Înmulţirea şi împărţirea: O situaţie similară este şi în cazul operaţiei de înmulţire. La înmulţirea a două numere de câte n cifre fiecare (indiferent de baza aleasă) se poate

obţine un număr de $2n$ cifre în aceeaşi bază. De aceea UCP foloseşte două cuvinte pentru reprezentarea rezultatului înmulţirii a două numere de lungimea cuvântului. La fel pentru operaţia de împărţire foloseşte două cuvinte: unul pentru câtul şi altul pentru restul împărţirii. Însă în limbajele de programare de nivel mediu şi înalt este accesibil doar un cuvânt din cele două ale rezultatului operaţiilor multiplicative. De aceea pentru a obţine câtul şi restul împărţirii se efectuează operaţia de împărţire de două ori, pentru cât şi rest câte o dată. Iar la înmulţire cuvântul superior se ignoră (de ex., pe patru biţi: $1111_2 \times 1111_2 == 0001_2$ sau $15 \times 15 == 1$).

Scăderea: Scăderea numerelor naturale nu este operaţie algebrică, deoarece există numere naturale diferenţa cărora nu este un număr natural ($a - b < 0 \Leftrightarrow a < b$). Calculatorul, însă, foloseşte o altă interpretare a noţiunii de număr, astfel că scăderea este operaţie algebrică chiar şi în mulţimea numerelor naturale (ale cuvintelor procesorului). Acest fapt se datorează modului de efectuare a operaţiei de adunare, scăderea fiind operaţia inversă a adunării. Conform definiţiei operaţiei de scădere care se învaţă în şcoală, $a - b = c$ dacă şi numai dacă $c + b = a$. Folosind exemplul de mai sus, din egalitatea „$255 + 1 == 0$” obţinem: $0 - 1 == 255$, adică scăzând un număr mai mare din altul mai mic obţinem un număr pozitiv, deci natural. Dacă omitem descăzutul, obţinem o egalitate stranie din punct de vedere matematic: $-1 == 255$. Explicaţia este simplă: când dintr-un număr mai mic se scade unul mai mare (de ex. 0 – 1), la numărul mai mic se adaugă bitul de transport (care se ignoră la operaţia de adunare), apoi se efectuează scăderea.

$00000000_2 - 00000001_2 == \mathbf{1}\ 00000000_2 - 00000001_2 = 11111111_2$

Se poate uşor de verificat că mulţimea numerelor naturale împreună cu operaţiile de adunare şi înmulţire definite mai sus formează inel ($\mathbb{Z}_c$, unde $c = 2^n$, n – lungimea cuvântului procesorului).

Dacă considerăm numerele întregi ca o extindere a numerelor naturale pe baza operaţiei de scădere, putem identifica numerele naturale cu cele întregi, şi anume: orice număr întreg z poate fi scris ca diferenţa u – v a două numere naturale. Dar pentru că diferenţa u – v a două cuvinte ale UCP la fel este un număr natural, apare necesitatea definirii numerelor negative. Mulţimea valorilor unui cuvânt este 2^n. E natural ca jumătate din aceste valori să reprezinte numere negative şi jumătate –

numere pozitive şi zero. Putem construi mulţimea de valori întregi pentru o lungime dată a cuvântului procesorului astfel:

$Z_n = \{-2^{n-1}, -2^{n-1}+1, -2^{n-1}+2, \ldots, -1, 0, 1, \ldots, 2^{n-1}-1\}.$

De ex.:

$N_8 = \{0, 1, 2, \ldots, 255\},$

$Z_8 = \{-128, -127, \ldots, -1, 0, 1, \ldots, 127\},$

$ord(N_8) = ord(Z_8) = 2^8 = 256.$

Observăm că $ord(N_n) = ord(Z_n) = 2^n$, adică $N_n \cong Z_n$.

Fie aplicaţia $\omega: N_n \to Z_n$, definită astfel:

$\omega(x) = x$, dacă $x < 2^{n-1}$ şi $\omega(x) = x - 2^n$, dacă $x \geq 2^{n-1}$;

$\omega^{-1}(y) = y$, dacă $y \geq 0$ şi $\omega^{-1}(y) = y + 2^n$, dacă $y < 0$.

Funcţia ω este izomorfism şi reprezintă legătura dintre valoarea unui număr întreg pe un cuvânt (valoarea totdeauna este număr natural) şi interpretarea acesteia.

Mai sus am menţionat că $-1 == 255$. Apare întrebarea: Cum de identificat numerele negative? Observam că toate numerele negative sunt mai mari sau egale cu -2^{n-1}, adică au bitul superior 1. Pentru numerele întregi, prin convenţie, dacă bitul superior este 1, numărul se consideră negativ, altfel numărul se consideră nenegativ.

Să examinăm reprezentarea binară a Z_8:

Zecimal	Binar	Zecimal	Binar
0	0000 0000	128 == -128	1000 0000
1	0000 0001	129 == -127	1000 0001
…	…	…	…
126	0111 1110	254 == -2	1111 1110
127	0111 1111	255 == -1	1111 1111

Observăm că $-0 == 0$ şi $-2^{n-1} == 2^{n-1}$. De această proprietate se bucură doar aceste două valori.

Se poate demonstra că operaţiile de adunare şi scădere astfel definite pe cuvintele procesorului nu depind de interpretarea valorii acestora (fie că numărul se consideră întreg sau natural).

Însă operaţiile de înmulţire şi împărţire se efectuează diferit în funcţie de interpretarea valorilor. Astfel, daca se doreşte înmulţirea a două numere naturale, se alege o instrucţiune a UCP (în limbajul ASM este `mul`), iar dacă se doreşte înmulţirea

aceloraşi valori, dar ca numere întregi, se alege altă instrucţiune a UCP (`imul`). Analogic pentru împărţire (instrucţiunile `div` şi `idiv`).

În limbajele de programare de nivel mediu sau înalt de alegerea operaţiei corespunzătoare pentru numere întregi sau naturale se ocupă compilatorul[10] sau interpretorul, în funcţie de tipul de date al operanzilor. Deci tipul de date este o construcţie a limbajului respectiv, dar nu a UCP. Pentru UCP contează doar dimensiunea datelor şi instrucţiunea de efectuat, iar instrucţiunile simple (care se execută la un singur tact al ceasului procesorului) se efectuează doar asupra datelor de dimensiuni nu mai mari decât mărimea procesorului.

Indiferent de instrucţiunea aleasă pentru operaţia multiplicativă, în limbajele de nivel mediu şi înalt, operaţia se efectuează conform descrierii de mai sus. Şi anume: la înmulţire, cuvântul superior al produsului se trunchiază, iar la împărţire rezultatul e format din două cuvinte – unul pentru cât şi altul pentru rest.

În baza acestor reguli se poate afirma că în sistemele de calcul moderne nu există numere naturale/întregi autentice, dar există **clase de resturi** după modulul 2^n, unde n este lungimea cuvântului UCP.

[10] Un program special care transformă codul sursă scris într-un limbaj de programare accesibil omului în cod-maşină.

I.2. Numere sistematice. Între valoare şi reprezentare.

Pentru ca omul să poată înțelege și folosi o noțiune abstractă are nevoie de o experienţă de interacţiuni cu obiecte concrete din volumul noţiunii date. Apoi, în baza experienţei, poate opera mintal cu noțiunea abstractă formată/învăţată prin experienţă. De altfel așa au apărut majoritatea cuvintelor din orice limbă, prin abstractizarea unor obiecte des întâlnite în viaţa cotidiană. Spre exemplu, în limba strămoşilor noştri existau noţiunile „măr", „păr", „nuc", însă nu exista noţiunea „copac", care a apărut mai târziu.

Orice număr este o abstracţie. Iar noţiunea de număr este o abstracţie şi mai mare. De aceea, pentru ca omul să însuşească şi să poată folosi numerele are nevoie de o experienţă de interacţiuni cu fiecare număr. Numerele mici se însuşesc foarte uşor, fiind cel mai des întâlnite în viaţa de zi cu zi, ceea ce nu se poate spune despre numerele mari. Însă chiar dacă omul nu poate percepe o valoare mare, totuşi el poate efectua operaţii concrete cu această valoare şi o poate comunica şi altor oameni ca entitate de informaţie. Omul nu ar putea opera cu numere mari fără un sistem de numeraţie, datorită naturii abstracte a acestora şi a cantităţii de numere existente.

Se poate spune că un **sistem de numeraţie** este *un set de reguli şi de numere cunoscute care permite reprezentarea sau/şi efectuarea unor operaţii cu o mulţime mai mare de numere, finită sau infinită*. Numerele cunoscute se numesc cifre şi de regulă numărul lor este foarte mic (cel puţin una). Fiecare cifră, prin convenţie, poate fi reprezentată de un simbol (grafic, sonor, electronic etc.) în mod univoc. Operaţiile asupra cifrelor la fel trebuie să fie cunoscute/definite.

O trăsătură importantă a operaţiilor asupra numerelor este faptul că operaţiile nu depind de sistemul de numeraţie ales. Sistemul „are grijă" de reprezentarea sub o anumită formă a numărului, dar „nu atinge" valoarea acestuia. De exemplu, doi plus doi este patru în orice sistem de numeraţie. Însă în funcţie de sistemul ales, putem defini şi regula de efectuare a operaţiei date. Anume regula de efectuare a operaţiei depinde de sistemul ales, dar nu şi rezultatul operaţiei.

Cel mai răspândit sistem de numeraţie din zilele noastre este sistemul poziţional de numeraţie, în care valoarea fiecărei cifre este determinată de poziţia acesteia în

număr. Cantitatea de cifre dintr-un sistem poziţional de numeraţie se numeşte **baza sistemului** şi este proprietatea caracteristică a sistemului poziţional. Dată fiind natura acestui sistem, baza trebuie să fie nu mai mică decât doi.

Să notăm baza sistemului poziţional prin $\mathbb{b}$ și setul de cifre prin $\mathbb{C}_{\mathbb{b}} = \{0, 1, \ldots, \mathbb{b}-1\}$. Atunci $\forall \alpha \in \mathbb{N}$, α se reprezintă în baza $\mathbb{b}$ astfel:

$$\alpha = a_n\mathbb{b}^n + a_{n-1}\mathbb{b}^{n-1} + \ldots + a_1\mathbb{b} + a_0,\ a_i \in \mathbb{C}_{\mathbb{b}},\ i \in \{0, 1, \ldots, n\}. \quad (1)$$

Vom numi a_n prima cifră a numărului α, iar a_0 – ultima cifră. Numărul i este poziţia cifrei a_i. Poziţia n o vom numi poziţia superioară, iar 0 – poziţia inferioară.

Dacă notăm $\alpha = \alpha(\mathbb{b})$, obţinem o expresie de forma unui polinom peste $\mathbb{C}_{\mathbb{b}}$, pe care formal o putem nota astfel:

$$\alpha(\mathbb{b}) = a_n\mathbb{b}^n + a_{n-1}\mathbb{b}^{n-1} + \ldots + a_1\mathbb{b} + a_0,\ a_i \in \mathbb{C}_{\mathbb{b}}. \quad (2)$$

Desigur $\alpha(\mathbb{b})$ nu e polinom, deoarece $\mathbb{C}_{\mathbb{b}}$ nu e domeniu de integritate. Coeficienţii a_i în baza $\mathbb{b}$ sunt de aceeași natură cu „necunoscuta” $\mathbb{b}$, adică numere, însă $a_i < \mathbb{b}$. După cum am menţionat mai sus, valoarea numărului α nu depinde de bază, însă pentru a desemna anumite proprietăţi ale numărului α scris într-o anumită bază, vom nota $\alpha(\mathbb{b}) = \alpha_{\mathbb{b}}$. Numărul $\alpha_{\mathbb{b}}$ scris în baza $\mathbb{b}$ se numește **număr sistematic**. În mod obișnuit, numărul α în baza $\mathbb{b}$ se scrie astfel: $(\overline{a_n a_{n-1} \ldots a_1 a_0})_{\mathbb{b}}$.

Exemplu: $352_8 = (3\times8^2 + 5\times8 + 2)_{16} = (EA)_{16} = (14\times16 + 10)_{10} = 234_{10}$

Folosind un sistem poziţional de numeraţie, dacă sunt definite operaţiile de adunare şi înmulţire a cifrelor (sunt date tabelele de adunare şi de înmulţire), poate fi calculată suma și produsul oricăror numere sistematice.

Încă un avantaj important al numerelor sistematice este posibilitatea manipulării cifrelor numărului, folosirea criteriilor de divizibilitate specifice unei baze, definirea de alte operaţii asupra cifrelor distincte şi aplicarea algoritmilor asupra numărului pe baza cifrelor acestuia (de ex., înmulţirea cu $\mathbb{b}^n$ se obţine prin deplasarea cifrelor cu n poziţii la stânga).

Pentru a folosi avantajele numerelor sistematice în sistemele de calcul, considerăm următoarele principii:

Principiul 1 (recursivitatea reprezentării): Deoarece cifrele $\mathbb{C}_{\mathbb{b}}$ la fel sunt numere, acestea la fel pot fi scrise într-o anumită bază conform regulilor de mai sus și invers –

orice număr sistematic $\alpha_{\mathbb{b}}$ scris în baza $\mathbb{b}$ poate servi drept cifră pentru înscrierea altui număr (mai mare) într-o anumită bază, respectând restricţia de înscriere $\alpha_{\mathbb{b}} < \mathbb{b}_{\mathbb{b}}$, unde $\mathbb{b}_{\mathbb{b}} > \mathbb{b}^n$ este noua bază, iar numărul $\alpha_{\mathbb{b}}$ este cifră în baza $\mathbb{b}_{\mathbb{b}}$. Acest principiu poate fi aplicat recursiv ori de câte ori dorim.

Noua bază $\mathbb{b}_{\mathbb{b}}$ poate deveni destul de mare și incomodă la aplicarea în practică de către om din cauza numărului mare de simboluri necesare pentru fiecare cifră. Sistemele de calcul, însă, nu au nevoie de simboluri speciale pentru a opera cu numerele sistematice și principiul recursiv de reprezentare a numărului în altă bază poate fi aplicat cu succes.

În cazul când noua bază $\mathbb{b}_{\mathbb{b}}$ are forma $\mathbb{b}^x$, fiecare grup de x cifre consecutive ale $\alpha_{\mathbb{b}}$ formează o cifră a numărului $\alpha_{\mathbb{b}^x}$. De exemplu, pentru x = 4 și n = 4k+2 avem:

$$\alpha(\mathbb{b}^4) = (a_0+a_1\mathbb{b}+a_2\mathbb{b}^2+a_3\mathbb{b}^3) + (a_4+a_5\mathbb{b}+a_6\mathbb{b}^2+a_7\mathbb{b}^3)\times\mathbb{b}^4 + \ldots +$$
$$+ (a_{n-6}+a_{n-5}\mathbb{b}+a_{n-4}\mathbb{b}^2+a_{n-3}\mathbb{b}^3)\times(\mathbb{b}^4)^{k-1} + (a_{n-2}+a_{n-1}\mathbb{b}+a_n\mathbb{b}^2)\times(\mathbb{b}^4)^k$$

Dacă deschidem parantezele, obţinem reprezentarea $\alpha_{\mathbb{b}}$.

Este mai ușor de operat cu asemenea numere, deoarece poziţia cifrelor la diferite operaţii se calculează simplu.

Principiul 2 (păstrarea structurii): Dacă numărul sistematic $\nu_{\mathbb{b}}$ se obţine în rezultatul efectuării unei operaţii $\alpha_{\mathbb{b}}$ □ $\beta_{\mathbb{b}}$ și pe poziţia i a numărului $\nu_{\mathbb{b}}$ se obţine un număr $c_i \geq \mathbb{b}$ (să zicem, din „neatenţia" algoritmului), atunci din înscrierea numărului c_i în baza $\mathbb{b}$ ultima cifră se plasează pe poziţia i a $\nu_{\mathbb{b}}$, iar numărul format din restul cifrelor c_i se adună la cifra următoare c_{i+1}, respectând acest principiu şi pentru poziţia i+1.

Regula aceasta este valabilă pentru orice operaţie în rezultatul căreia pe o anumită poziţie se obţin cifre mai mari sau egale cu baza. Astfel operaţiile de adunare şi de înmulţire a două numere sistematice pot fi efectuate asemănător cu operaţiile omoloage pentru polinoame, aplicând principiul de mai sus.

Prin analogie cu polinoamele, în înscrierea (2) a numărului sistematic $\alpha_{\mathbb{b}}$ vom considera $a_n \neq 0$ și notăm grad($\alpha_{\mathbb{b}}$) = n.

Iată câteva proprietăţi ale numerelor sistematice:

1°. $\square\alpha_{\mathbb{b}}, \beta_{\mathbb{b}}$, grad($\alpha_{\mathbb{b}}$)= m, grad($\beta_{\mathbb{b}}$)= n, grad($\alpha_{\mathbb{b}} + \beta_{\mathbb{b}}$) = k, (**k = max{m, n}**) ∨ (**k = max{m, n} + 1**);

2°. $\square\alpha_{\mathbb{b}}, \beta_{\mathbb{b}}$, grad($\alpha_{\mathbb{b}}$)= m, grad($\beta_{\mathbb{b}}$)= n, grad($\alpha_{\mathbb{b}} \times \beta_{\mathbb{b}}$) = k, **k = m + n.**

Proprietăţile de mai sus rezultă direct din regulile de adunare și de înmulţire a două numere sistematice.

Se cunosc mai multe reguli de conversie a numerelor sistematice dintr-o bază în alta. Dacă conversia este efectuată de către om, de regulă se folosește baza zece ca intermediară, dat fiind faptul că cei mai mulţi dintre oameni cunosc bine numerele zecimale şi operaţiile aritmetice cu acestea. În general, însă, nu e nevoie de o bază intermediară.

La esenţa conversiei bazei numerelor sistematice stă relaţia dintre cifrele numărului în baza nouă şi însăși noua bază: $a_i' < \mathbb{b}'$.

Cu ajutorul unor operaţii de împărţire succesive, se pot obţine cifrele numărului sistematic în baza nouă. Și anume, împărţind numărului α la baza nouă $\mathbb{b}'$, obţinem ultima cifră a_0' a numărului α în baza $\mathbb{b}'$, iar câtul împărţirii $q_1 = [\alpha / \mathbb{b}']$ reprezintă numărul α în baza $\mathbb{b}'$ fără ultima cifră. Repetând procedeul pentru numărul q_1, obţinem a_1' şi q_2, apoi a_2' şi q_3, ş.a.m.d. În sfârşit se obţine $q_{m+1} = 0$ și reprezentarea numărului $\alpha(\mathbb{b}') = a_n'\mathbb{b}'^n + a_{n-1}'\mathbb{b}'^{n-1} + \ldots + a_1'\mathbb{b}' + a_0'$.

Observaţie: Pentru conversia unei baze mai mare în alta mai mică, poate fi aplicat *principiul 2* de mai sus.

I.3. Numerele sistematice în sistemele de calcul electronice.

Sistemele de calcul electronice moderne folosesc impulsul electric pentru codificarea informaţiei. Din mai multe motive tehnice şi tehnologice, informaţia se codifică binar. Valoarea 1 corespunde impulsului electric, iar valoarea 0 corespunde lipsei impulsului electric. Analogic se procedează la codificarea datelor memorate: o celulă de memorie se numește bit și poate avea două stări –închis pentru 1 și deschis pentru 0. De aceea sistemul de numeraţie folosit de către astfel de sisteme de calcul este cel binar.

Sistemul poziţional de baza doi, pe lângă faptul că este minim, mai are ceva special faţă de celelalte baze, și anume poate folosi ca cifre valorile booleene ADEVĂ și FALS, corespunzătoare valorilor numerice 1 și 0, respectiv. Folosind avantajul înscrierii poziţionale a numerelor sistematice, putem aplica operaţiile logice asupra numerelor cifră cu cifră.

Să considerăm numerele $\alpha = 155_{10}$ și $\beta = 132_{10}$ pe un cuvânt al procesorului de 8 biţi. Vom efectua următoarele operaţii logice asupra numerelor α și β cifră cu cifră (bit cu bit): `și (&), sau (|), sau exclusiv (^), negaţia (~)`.

Notaţia	Valoarea binară	Valoarea zecimală
α	**1001 1011**	**155**
β	**1000 0100**	**132**
α & β	1000 0000	128
α | β	1001 1111	159
α ^ β	0001 1111	31
~α	0110 0100	100
~β	0111 1011	123

Operaţiile bit cu bit au echivalente logice. Prin convenţie (dar şi regulă a unor limbaje de programare de tipul C, Modulo-2, PHP, ş.a.), în operaţiile logice valoarea 0 se evaluează FALS, iar orice valoare diferită de 0 se evaluează ADEVĂR.

Operaţii cu cifrele numerelor binare:

UCP a unui sistem de calcul electronic efectuează operaţii logice şi aritmetice cu numere binare de lungimea cuvântului procesorului. În afară de operaţiile aritmetice, UCP mai efectuează şi alte operaţii de manipulare a datelor. Însă nu toate instrucţiunile procesorului sunt disponibile în limbajele de programare de nivel mediu sau înalt. În afară de instrucţiunile enumerate mai sus, limbajele de programare de nivel mediu și înalt mai implementează un tip de instrucţiuni pentru manipularea numerelor sistematice în baza doi: deplasările logice și aritmetice.

Deplasările logice sunt de două tipuri: la stânga și la dreapta. O deplasare la stânga cu n poziţii (binare) adaugă la dreapta numărului n cifre de 0, iar primele n cifre ale numărului (cifrele superioare) se trunchiază, păstrându-se astfel lungimea cuvântului. Deplasarea logică la dreapta este analogică.

De exemplu, să deplasăm numărul 211_{10} cu 3 poziţii la stânga:

$\underline{111}\mathbf{01001}_2$ □ 3 == $\mathbf{01001}\underline{000}_2$

Deplasările aritmetice sunt asemănătoare cu cele logice, cu unica deosebire că deplasarea aritmetică la dreapta păstrează semnul numerelor întregi, adică nu adaugă totdeauna cifra 0 la stânga numărului, ci bitul de semn, care poate fi 0 sau 1.

$\mathbf{1}0001010_2$ □ 4 == $\mathbf{11111}000_2$, $(-118$ □ $4 = -3)_{10}$

$\mathbf{0}1110110_2$ □ 4 == $\mathbf{00000}111_2$, $(\ 118$ □ $4 = \ \ 3)_{10}$

Observăm că n deplasări la stânga sunt echivalente cu înmulţirea numărului cu 2^n, iar n deplasări la dreapta – cu împărţirea la 2^n. Din punct de vedere aritmetic, deplasările aritmetice se deosebesc de cele logice prin faptul că păstrează semnul numerelor negative și deci pot fi folosite pentru operaţia de înmulţire și împărţire cu 2^n, în conformitate cu reprezentarea numerelor negative.

Operaţiile multiplicative aritmetice consumă cele mai multe cicluri ale UCP din familia de procesoare 80x86. În special operaţia de împărţire consumă cel mai mult timp al UCP. Deplasările, însă, sunt dintre cele mai rapide instrucţiuni. Astfel, de fiecare dată când este posibil, se recomandă folosirea deplasărilor în locul operaţiilor multiplicative.

Înmulţirea cu ajutorul deplasărilor ușor se efectuează mai ales când avem constante. De ex., pentru a înmulţi un număr α cu 10, procedăm astfel:

$\alpha \times 10 = 2\alpha + 2^3\alpha = (\alpha \ll 1) + (\alpha \ll 3)$

Chiar dacă se efectuează trei operaţii în loc de o înmulţire, totuşi acestea trei împreună (două deplasări şi o adunare) se execută mai repede într-un sistem din familia 80x86.

Cu ajutorul operaţiilor logice bit cu bit şi a operaţiilor de deplasare putem manipula cifrele numărului sistematic în baza doi.

Citirea unei cifre binare din cuvânt:

În unele situaţii este nevoie de citit o cifră a numărului binar, adică de determinat dacă cifra de pe poziţia `i` este `0` sau `1`. Cu acest scop se foloseşte un număr auxiliar `m` numit **mască** şi operaţia `&` (şi bit cu bit). Masca are pe poziţia `i` cifra `1`, iar restul cifrelor egale cu `0`.

```
01101101        01101101
00100000 &      00000010 &
--------        --------
00100000        00000000
TRUE            FALSE
```

Masca `m` uşor poate fi obţinut aplicând deplasarea la stânga. În concluzie, pentru a obţine cifra de pe poziţia `n` a numărului α, procedăm astfel: α `& (1 << n)`.

Scrierea unei cifre binare în cuvânt:

1. Dacă se dorește scrierea <u>cifrei 0</u> pe o anumită poziţie a numărului binar, se poate folosi același procedeu ca și la citirea unui bit, însă în acest caz masca are toţi biţii `1`, cu excepţia bitului de pe poziţia unde se scrie `0`:

```
01101101        01101101
11011111 &      11111101 &
--------        --------
01001101        01101101
```

Masca poate fi obţinută printr-o deplasare şi o negaţie:

`~(1 << 5) == ~00100000 == 11011111`

2. Scrierea <u>cifrei 1</u> pe o poziţie dorită a numărului binar se face cu ajutorul operaţiei `|` (sau bit cu bit):

```
01101101
00010000 |
--------
01111101
```

3. Algoritmul ce urmează scrie o cifră arbitrară b pe o poziţie arbitrară n a numărului binar α_2:

```
α ≔ α & ~(1 □ n); // scriem 0 pe poziția n
α ≔ α ^  (b □ n); // scriem b pe poziția n. 0 ^ b == b
```

Obţinerea unei secvenţe de cifre binare consecutive:

Deseori este nevoie de obţinut un număr cu o secvenţă de biţi 0 sau 1 consecutivi de o lungime dată, pe o poziţie dată. Dacă avem o secvenţă de biţi 1, prin negaţie se obţine aceeaşi secvenţă de biţi 0 (~00**111**000 = 11**000**111). Deci e suficient să găsim un algoritm pentru generarea unei secvenţe de n cifre consecutive de 1.

Observăm că $(2^n-1)_{10} = (2^{n-1}+2^{n-2}+2^{n-3}+\ldots+4+2+1)_{10} = 111\ldots111_2$, iar $2^n = 1$ □ n. De aici avem algoritmul de generare a numărului α format din n cifre de 1: α ≔ (1 □ n) – 1.

Dacă e specificată poziţia p pe care se doreşte de obţinut secvenţa, se aplică aceste instrucţiuni: α ≔ ((1□n)-1) □ p.

Proprietăţi ale negaţiei bit cu bit:

1°. **~0 == –1**

Ex. pe 8 biţi: ~00000000 = 11111111 = **1** 00000000 – 1 = -1

2°. **~α == ~0–α**

Ex. pe 8 biţi: ~10101011 = 11111111 - 10101011 = 01010100

3°. **~(α + β) == ~α + ~β + 1**

Într-adevăr: ~(α+β) == (~0-α-β)+(1–1) == ~0-α + ~0-β + 1

4°. **~(α × β) == –(~α)×(~β) – α – β.**

În baza celor expuse mai sus, se poate spune că pentru operaţii cu numere într-un sistem de calcul electronic cel mai convenabil este să considerăm numerele în sistemul binar de numeraţie. Însă sistemul binar este incomod pentru om, înscrierea numerelor în această bază fiind prea lungă. Iar conversia din baza doi în baza zece „ascunde” structura numărului binar. De aceea programatorii cel mai des folosesc numere sistematice în baze puteri ale lui doi, cu ar fi 8 sau 16. Aceste numere se

bucură de proprietăţile reprezentării recursive a numerelor sistematice de forma 2^n ($8 = 2^3$, $16 = 2^4$).

Astfel se obţin numere într-o bază apropiată de cea familiară nouă (10), dar din care ușor se vede structura numărului bina pe care îl reprezintă, și anume: în baza 8 fiecare grup de trei cifre binare consecutive formează o cifră octală, iar în baza 16 – fiecare grup de patru cifre binare constituie reprezintă o cifră hexazecimală.

Prezentăm tabelul de conversie din baza 2 în baza 8 și 16:

Binar	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binar	Hexazecimal	Zecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Exemplu de conversie:

$01101101_2 = 01\_101\_101_2 = 155_8$

$01101101_2 = 0110\_1101_2 = 6D_{16}$

Cu ajutorul acestor tabele conversia se efectuează fără calcule aritmetice. Se fac doar niște manipulări cu înscrierea sistematică a numărului. Deoarece tabelele conţin câte un set mic de cifre (8 și 16, respectiv), sunt ușor de memorizat (la fel cum am memorizat cândva cifrele zecimale).

Nu numai pentru om este comodă reprezentarea numerelor binare în baze de forma 2^n. Procesoarele din familia 80x86 implementează un şir de instrucţiuni pentru manipularea cifrelor binare (mai sus au fost expuse cele de bază). Cea mai mică unitate de informaţie pe care o poate prelucra într-o instrucţiune un astfel de procesor este octetul – opt biţi consecutivi, care conţin un număr binar de opt cifre. Un octet poate fi

afișat la consolă folosind exact două cifre hexazecimale: `0110_1101 → 6D`. Astfel, pentru a obține reprezentarea unui număr binar de k octeți în baza 16 (e valabil $\square 2^n$), se fac 2k conversii consecutive, separate. Deci algoritmul are complexitate liniară. Conversia din baza 16 în 2 se face analogic.

Desigur, pot fi utilizate și alte reprezentări, cum ar fi cea zecimală, care ne este mult mai familiară. Însă conversia în alte baze necesită calcule aritmetice, ceea ce consumă timp de execuție a UCP. Conversia unui număr binar într-o bază arbitrară cu ajutorul operațiilor aritmetice se face prin împărțirea numărului la noua baza până numărul devine zero. Dacă numărul are o lungime mai mare decât lungimea cuvântului procesorului, la fiecare împărțire se efectuează un număr de operații direct proporțional cu lungimea numărului. Astfel de algoritm are o complexitate pătratică. Conversia în sens opus se efectuează în mod analogic, fiind de aceeași complexitate.

Indiferent de natura operațiilor efectuate cu numerele în sistemele de calcul electronice, folosind proprietățile numerelor binare putem alcătui algoritmi optimi. De multe ori, în condiții concrete, deosebirea între alegerea algoritmului optim sau unuia neoptim este echivalentă cu posibilitatea sau imposibilitatea realizării sarcinii.

I.4. Inele ale claselor de resturi după modulul 2^n sau tipul INT în sistemele de calcul

În paragraful I.1 am arătat că în sistemele de calcul electronice moderne nu există numere autentice, dar există **clase de resturi** după modulul 2^n, unde n este mărimea procesorului. La fel am arătat că adunarea şi înmulţirea cuvintelor procesorului ca numere naturale coincid cu adunarea şi înmulţirea claselor de resturi după modulul 2^n. Aşadar mulţimea valorilor cuvântului unui procesor împreună cu operaţiile de adunare şi înmulţire a cuvintelor formează **inelul claselor de resturi** după modulul 2^n ($\mathbb{Z}_c$, unde $c = 2^n$). Legătura dintre numerele întregi într-un sistem de clacul și clasele de resturi după un modul ne permite să operăm cu structura algebrică cunoscută de inel al claselor de resturi pentru a studia proprietăţile numerelor şi a operaţiilor UCP.

Direct din această legătură rezultă următoarele aspecte:

1. atât timp cât operanzii şi rezultatul operaţiei sunt mai mici decât modulul, rezultatul obţinut este acelaşi ca şi pentru numere în matematică. În caz contrar, rezultatul obţinut este un reprezentant al aceleiași clase de resturi, însă mai mic ca modulul;
2. deoarece o clasă de resturi conţine o infinitate de elemente, inclusiv numere negative, unele clase de resturi reprezentate de valori pozitive pot fi considerate negative.

După cum știm, în calculator există doar valori numere naturale, celelalte numere fiind o îmbinare între careva numere naturale și careva algoritmi. Dacă privim numerele întregi dintr-un sistem de calcul cu UCP de mărimea n ca clase de resturi după modulul 2^n, în baza proprietăţilor claselor de resturi uşor putem explica din punct de vedere matematic operaţiile cu numerele întregi și reprezentarea lor ca valori numere naturale în memorie.

Egalitatea valorilor a două cuvinte ale procesorului se identifică cu congruenţa după modul a numerelor întregi sau naturale corespunzătoare acestor valori. De exemplu, pe un procesor de 8 biţi, egalitatea $255 == -1$ se explică prin congruenţa $255 \equiv -1\ (256)$, unde $256 = 2^8$. În acest exemplu ambele cuvinte conţin aceeași valoare 255, însă sunt interpretate diferit: prima ca număr natural (255), iar a doua ca număr întreg (-1).

Pentru simplitate voi omite modulul în notaţia congruenţelor după modulul 2^n, corespunzător mărimii procesorului n.

Să deducem regula de obţinere a opusului unui număr întreg:

$-1 \equiv 2^n-1$

Numerele sistematice de forma $(\mathbb{b}^n-1)_{\mathbb{b}}$ sunt formate din n cifre cu valoarea ($\mathbb{b}$–1). Pentru $\mathbb{b}$ = 2, acest număr este format din n cifre de 1, adică pentru n cifre binare avem: 2^n-1 = ~0. Astfel **–1 == ~0** pentru orice lungime a cuvântului procesorului. Folosind această proprietate şi proprietatea negaţiei ~α == ~0 – α, avem:

$-\alpha = 1 - 1 - \alpha \equiv \sim 0 - \alpha + 1 \equiv \sim\alpha + 1.$

Am obţinut **–α ≡ ~α + 1**.

Daca considerăm α o cifră a unui număr destul de mare, în baza principiului recursivităţii reprezentării numerelor sistematice, relaţia de mai sus poate fi folosită pentru a obţine opusul unui număr oricât de mare.

În multe limbajele de programare procedurale există mai multe tipuri de date pentru numerele întregi și naturale, ele deosebindu-se doar prin mărime. De exemplu, în limbajul C++ există tipuri de date pentru numere întregi și naturale de 8, 16, 32 și 64 de biţi. În ordinea enumerată, tipurile de date pentru numerele întregi sunt: `signed char, short, long, long long`.

Din punct de vedere matematic, fiecare din aceste tipuri de date reprezintă câte un inel al claselor de resturi după modulul 256, 65536, 4294967295, și 18446744073709551615, respectiv. Ceea ce înseamnă că variabilele de aceste tipuri pot opera cu valori nu mai mari decât modulul corespunzător tipului variabilei.

Dacă în unele limbaje nu sunt disponibile toate aceste tipuri de date, cele mai multe limbaje procedurale au cel puţin tipul `INT`, care are mărimea procesorului și reprezintă un cuvânt al acestuia interpretat ca număr întreg. Astfel, pentru a opera cu numere și mai mari decât cele disponibile într-un limbaj sau altul, pot fi compuși algoritmi care folosesc principiile de reprezentare a numerelor sistematice. În calitate de cifre se pot considera grupări a câte n biţi, astfel ca n să nu fie mai mare decât jumătate din lungimea cuvântului procesorului, ca să nu se piardă din valoare în rezultatul efectuării operaţiilor aritmetice. De exemplu, fiecare octet poate fi considerat ca cifră. Pentru UCP sunt cunoscute toate operaţiile aritmetice pentru

cifrele-octeți. Dacă se cunosc operațiile aritmetice cu cifrele, aceste operații pot fi extinse și asupra numerelor sistematice care folosesc aceste cifre. Mărimea numerelor astfel obținute poate fi una fixată/statică sau poate fi flexibilă/dinamică.

I.5. Noţiunea de divizibilitate în ℤ. Teorema împărţirii cu rest.

Noţiunea de divizibilitate se defineşte prin noţiunea de înmulţire. În sistemele de calcul electronice, însă, pentru a verifica divizibilitatea numerelor se aplică un criteriu de divizibilitate (de regulă în baza doi), iar în caz general se compară cu zero restul împărţirii. Deci la verificarea divizibilităţii a două numere se utilizează operaţia de împărţire, pe care UCP o efectuează puţin diferit pentru numerele întregi faţă de convenţia matematică.

Pentru calcularea câtului și restului împărţirii, UCP execută o singură instrucţiune, însă voi nota această instrucţiune prin două simboluri: „/” se va referi la cât și simbolul „%” – la rest (în conformitate cu notaţia din limbajul C).

Conform Teoremei Împărţirii cu Rest, restul totdeauna este nenegativ:

$\forall a, b \in \mathbb{Z}, b \neq 0, \exists! q \in \mathbb{Z}, \exists! r \in \mathbb{N}, r < |b|: a = q \times b + r$

Pentru a respecta teorema suntem nevoiţi să avem două reguli diferite pentru împărţire – una pentru numere naturale și una pentru numere întregi.

Împărţirea numerelor naturale se consideră definită:

$\forall a, b \in \mathbb{N}, q = a / b, r = a \% b: q, r \in \mathbb{N}$.

Pe baza împărţirii numerelor naturale, se defineşte împărţirea numerelor întregi:

Dacă $a < 0$ și $b < 0$, atunci $q = |a| / |b|, r = |a| \% |b|$.

Dacă $a > 0$ și $b < 0$, atunci $q = -(|a| / |b|), r = |a| \% |b|$.

Dacă $a < 0$ și $b > 0$, atunci avem două cazuri:

1: $|a| \% |b| = 0 \Rightarrow q = -(|a| / |b|), r = 0;$

2: $|a| \% |b| \neq 0 \Rightarrow q = -(|a| / |b| + 1), r = b - |a| \% |b|.$

În celelalte cazuri a și b coincid cu numere naturale, la fel şi operaţia de împărţire asupra lor.

Regula de împărţire a numerelor întregi pare a fi complicată, însă garantează că restul niciodată nu e negativ. Unde ceva se câștigă, altceva se pierde…

UCP la fel are două instrucţiuni diferite pentru operaţia de împărţire – una pentru numere naturale și alta pentru numere întregi – însă deosebirea între acestea este în interpretarea valorii, nu în regula operaţiei. Iar regula e una, și anume: instrucţiunea de împărţirea a numerelor nenegative coincide cu împărţirea numerelor naturale. Iar

dacă unul din operanzi este negativ, valoarea absolută a câtului și a restului se calculează la fel ca și pentru numerele pozitive, iar semnele se aleg conform egalităţilor de mai jos:

Fie a, b, q, r ∈ ℕ: a = q×b + r.

```
  a  /   b   =  q,        a  %   b   =  r;
(–a) / (–b)  =  q,      (–a) % (–b)  = –r;
(–a) /   b   = –q,      (–a) %   b   = –r;
  a  / (–b)  = –q,        a  % (–b)  =  r;
```

Tabelele de repartizare a semnelor pentru câtul și restul împărţirii UCP:

Câtul

	a/b	b +	b –
a	+	+	–
	–	–	+

Restul

	a%b	b +	b –
a	+	+	+
	–	–	–

În baza celor expuse mai sus, se poate de formulat teorema împărţirii cu rest pentru sistemele de calcul electronice:

∀a, b∈ ℤ, b ≠ 0, ∃!q, r∈ ℤ, |r|<|b|, abq≥0, ar≥0: a= q×b + r.

Această teoremă „permite” restului să fie negativ. Însă cazul când restul este nul coincide pentru ambele teoreme (a = q×b). Astfel noţiunea de divizibilitate pentru sistemele de calcul coincide cu cea matematică.

Pentru utilizarea în practică, instrucţiunea de împărţire a UCP este mai convenabilă decât operaţia matematică de împărţire. Cred că anume din acest motiv designer-ii microprocesoarelor au ales anume acest mod de efectuare a împărţirii.

CAPITOLUL II. APLICAŢII ÎN PROGRAMARE

II.1. Criterii de divizibilitate în baza 2

În școală, după însuşirea noţiunii de divizibilitate, se învaţă unele criterii de divizibilitate pentru unele numere mici. Aceste criterii se bazează pe înscrierea sistematică a numerelor în baza zece (operează cu cifrele zecimale), deci pot fi aplicate doar în această bază. Ele sunt bune pentru oameni, care folosesc sistemul zecimal de numeraţie, însă sunt absolut inutile pentru calculatoarele binare.

Calculatoarele moderne (familia 80x86) folosesc sistemul binar de numeraţie pentru reprezentarea numerelor naturale și întregi în memorie, astfel pot folosi criterii de divizibilitate a numerelor în baza doi și alte avantaje ale reprezentării binare a numerelor.

Pentru ce oamenii folosesc criterii de divizibilitate? Oare nu este mai simplu să împărţim un număr la altul și să aflăm restul decât să ţinem minte atâtea criterii? Nu. Desigur cu ajutorul împărţirii putem verifica dacă orice două numere sunt divizibile, însă pentru numere mari este mai greu de efectuat împărţirea decât de aplicat un criteriu (dacă e posibil).

Analogică este situaţia şi în lumea calculatoarelor: dintre toate instrucţiunile aritmetice (și nu numai) ale UCP din familia 80x86, împărţirea se execută cel mai lent. De aceea sunt binevenite unele criterii de divizibilitate în sistemul binar care permit evitarea împărţirii.

a. Criteriul de divizibilitate cu 2:

Analogic cu criteriul de divizibilitate cu 10 în baza 10 este criteriul de divizibilitate cu 2 în baza 2 (și pentru orice bază), adică ultima cifră trebuie să fie 0. Acest criteriu rezultă din reprezentarea numerelor sistematice. Ultimul bit al numărului poate fi verificat astfel:

```
(α & 1 == 0) sau (α □ 1 □ 1 == α).
```
(3)

Prima expresie verifică ultima cifră aplicând masca `1` și este preferabilă pentru performanţă. A doua expresie anulează ultima cifră a numărului α apoi compară numărul obţinut cu α iniţial şi dacă sunt egale, ultima cifră e zero. Să observăm că

$(\alpha \,\&\, 1 == \alpha \,\%\, 2)$ și $(\alpha \gg 1 \ll 1 == [\alpha / 2] * 2 = \alpha - \alpha \,\%\, 2)$. Adică egalitatea a doua din (3) este echivalentă cu prima.

b. Criteriul de divizibilitate cu 2^k:

Criteriul de divizibilitate cu 2 poate fi extins și pentru 2^k foarte simplu, și anume: următoarele expresii sunt criterii de divizibilitate cu 2^k:

$$(\alpha \,\&\, (2^k-1) == 0) \text{ sau } (\alpha \gg k \ll k == \alpha). \qquad (4)$$

Din înscrierea sistematică în baza 2 se observă ușor că:

$(\alpha \,\&\, (2^k-1) == \alpha \,\%\, 2^k)$ și $(\alpha \gg k \ll k == [\alpha / 2^k] * 2^k)$.

În mod natural apar următoarele întrebări: având un număr dat β, cum de verificat dacă $\beta = 2^k$ și cum de aflat $k = \log_2\beta$ fără a apela funcţia logaritm?

Să comparăm reprezentarea binară a numerelor $\beta = 2^5$ și $\beta - 1$:

$2^5 = 00100000_2$,

$2^5-1 = 00011111_2$.

Dacă aplicăm conjuncţia bit cu bit, în rezultat obţinem: $\beta \,\&\, (\beta - 1) == 0$. Această egalitate are loc numai pentru numerele de forma $\beta = 2^k$, în baza egalităţii:

$2^k-1 = 2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots + 2 + 1$.

Dacă notăm cu 0×2^s termenii nuli din $\beta = 2^k$, obţinem:

$$\beta \,\&\, (\beta - 1) = (1\times2^k + 0\times2^{k-1} + 0\times2^{k-2} + 0\times2^{k-3} + \ldots + 0\times2 + 0\times1) \,\&$$
$$\&\, (0\times2^k + 1\times2^{k-1} + 1\times2^{k-2} + 1\times2^{k-3} + \ldots + 1\times2 + 1\times1) =$$
$$= (1\&0)\times2^k+(0\&1)\times2^{k-1}+(0\&1)\times2^{k-2}+ \ldots +(0\&1)\times2 + (0\&1)\times1 = 0.$$

Pe fiecare poziţie se obţine conjuncţia $(1\&0)$. Dacă β ar avea altă formă, neapărat s-ar obţine pe careva poziţii şi conjuncţia $(1\&1)$, iar expresia de mai sus ar fi diferită de 0.

Observăm că $0 == 2^n$, pentru n lungimea cuvântului procesorului (sau mărimea în biţi a tipului de date INT). De aceea $0 \,\&\, (0-1) = 0$. De fapt $0 \,\&\, x = 0$, $\forall x \in \mathbb{Z}$, însă în contextul divizibilităţii acest caz trebuie examinat aparte.

Astfel am răspuns la prima întrebare:

$$\beta == 2^k \Leftrightarrow \beta \,\&\, (\beta - 1) == 0. \qquad (5)$$

Să presupunem că $\beta = 2^k$ și să găsim un algoritm pentru calcularea $k = \log_2\beta$ fără ajutorul funcţiei log.

Calcularea k se reduce la determinarea poziţiei bitului 1 în număr, care poate fi calculată cu ajutorul măștilor și a deplasărilor. Fie $b == 2^k – numărul cercetat.

```
$m = 1;          // masca
$k = 0;          // numărătorul pozițiilor binare
while($b & $m == 0) {
  $m = $m << 1; // trecem la următoarea poziție
  $k++;          // incrementăm poziția
}
```

La sfârșitul ciclului **while**, $k conţine valoarea căutată. Observăm că pentru $b == 0, se obţine un ciclu infinit. Acest caz trebuie prelucrat cu o condiţie suplimentară.

Algoritmul de mai sus poate fi scris și într-o formă mai concisă:

```
for($k=–1; $b; $b>>=1, $k++);
```
(6)

Această formă modifică variabila iniţială $b, ceea ce nu reprezintă o problemă în cazul când codul de mai sus apare în corpul unei funcţii. În schimb se prelucrează și cazul $b == 0 □ $k == –1, ceea ce este destul de convenabil. Apare însă o altă problemă: pentru unica valoare $b == 2^{n-1}, pentru care $b == –$b, se obţine un ciclu infinit. Numărul n este lungimea cuvântului procesorului. Problema constă în faptul că în PHP deplasările sunt aritmetice și dacă $b < 0, atunci ($b □ VAL) < 0, pentru orice număr de deplasări VAL. Cazul este unic și în general se prelucrează cu condiţia suplimentară la început: **if**($b<0) $k = NR_BIT_IN_INT; **else** …

În practica de programare aceste criterii au o largă întrebuinţare. De exemplu, al doilea criteriu poate fi aplicat în programarea dinamică pentru extinderea dinamică a memoriei alocate pentru o listă de elemente de lungime arbitrară. În urma unor cercetări costisitoare ale Microsoft, s-a constatat că pentru extinderea dinamică a memoriei alocate pentru un obiect/tablou când numărul de elemente crește liniar este cel mai eficient de realocat 160% din memoria curent alocat pentru acest obiect. Astfel bibliotecile standard din diferite medii de programare folosesc aceste rezultate pentru dirijarea procesului de alocare dinamică de memorie folosind două variabile diferite asociate fiecărui obiect cu alocare dinamică de memorie pentru un număr arbitrar de elemente: una pentru numărul de elemente și alta pentru capacitatea alocată. Când

numărul de elemente se apropie de capacitatea alocată, algoritmul mărește capacitatea obiectului cu factorul 1,6.

Există o soluţie mai simplă pentru problema realocării dinamice de memorie pentru un număr arbitrar de elemente. S-ar putea de găsit un algoritm de calculare a următoarei valori a capacităţii în funcţie de numărul de elemente, fără necesitatea de a păstra o variabilă destinată monitorizării capacităţii. Algoritmul trebuie să fie foarte simplu și rapid. Pentru simplitate, în calitate de factor se ia numărul 2, care este aproximativ egal cu 1,6. Astfel capacitatea obiectului totdeauna reprezintă o putere a lui 2 (0, 1, 2, 4, 8, 16, ...) și la fiecare realocare de memorie capacitatea pur și simplu se dublează. Rămâne de verificat când este nevoie de mărit capacitate. Aici vin în ajutor algoritmii descriși la criteriul de divizibilitate cu 2^k. Dacă numărul de elemente n crește cu o unitate la fiecare pas, n va trece prin toate valorile de forma 2^k, consecutiv. Când $n = 2^k$ (se depistează cu ajutorul (5)), capacitatea se dublează (2×n) și problema este rezolvată simplu și eficient. Dacă numărul de elemente crește cu un pas $s > 1$, dar constant, realocarea se poate face pentru numerele de forma $s \times 2^k$ și în acest caz problema se rezolvă simplu. Dacă numărul de elemente crește cu un pas arbitrar, știind numărul vechi de elemente și numărul nou de elemente pentru care se cere memorie, cu ajutorul algoritmului (6) se depistează capacităţile corespunzătoare fiecărui număr de elemente și dacă diferă, se face realocarea de memorie.

Criteriile de divizibilitate în baza 2^k pot fi aplicate cu succes la soluţionarea eficientă și a multor altor probleme des întâlnite în practica de programare.

II.2. Cel mai mare divizor comun şi cel mai mic multiplu comun a două numere întregi.

După cum am constatat în capitolul precedent, noţiunea de divizibilitate pentru sistemele de calcul coincide cu cea matematică, ceea ce ne îndreptăţeşte să aplicăm Algoritmul lui Euclid la determinarea celui mai mare divizor comun (CMMDC) a două sau mai multe numere întregi pe calculator.

Dacă se cunoaște CMMDC a două numere, ușor se poate calcula și cel mai mic multiplu comun (CMMMC) al acestor numere. Voi nota CMMDC al numerelor α și β prin (α, β), iar CMMMC – [α, β]. Iată o formulă simplă pentru calcularea CMMMC a două numere, dacă se cunoaște CMMDC al lor: `[α, β] = (α×β) / (α, β)`.

În acest paragraf vom analiza comparativ câţiva algoritmi pentru determinarea CMMDC a două numere întregi. Fiecare algoritm va fi prezentat în câte o funcţie[11] PHP. Toţi algoritmii au fost testaţi la consumul de timp pe un server Apache, cu UCP de marca Intel Pentium D 3.00GHz, sistem de operare pe 32 biţi. La testare s-a aplicat algoritmul testa de 100, de 1000, apoi de 10000 ori pe aceleași perechi de numere. Evident, timpul de execuţie nu depinde de numărul de repetări ale algoritmului, dar pentru a diminua influenţa altor factori care ar putea influenţa rezultatele testului, am ales anume această schemă.

1) Algoritmul Greedy – iterativ:

```
function divCom($a, $b)
{
   $r = 1;
   if($a < $b) $min = $a; else $min = $b;
   for($i=$min; $i>1; $i--)
      if(($a%$i==0) && ($b%$i==0) && ($r%$i!=0)) $r *= $i;
   return $r;
}
```

Acest algoritm este diferit de algoritmul lui Euclid. Algoritmul folosește structura aritmetică a numerelor naturale în baza teoremei de bază a aritmeticii.

Variabila `$r` acumulează divizorii comuni ai `$a` și `$b`. Divizorii comuni sunt căutaţi printre toate numerele naturale mai mici sau egale decât `$a` și `$b`. Condiţia

[11] O funcţie într-un limbaj de programare este un subprogram care efectuează ceva (calcule sau alte instrucțiuni) și poate returna o valoare, deci este un algoritm care poate fi parte componentă a altui algoritm de un nivel mai mic de granulare. Textul prezent folosește în multe locuri termenul de „algoritm” cu referire la o funcţie.

`$r % $i ≠ 0` filtrează divizorii care deja se conţin în `$r`. Această condiţie necesită ca numerele să fie parcurse în ordine inversă, pentru a obţine în `$r` cei mai mari divizori.

Algoritmul se numește „Greedy”, deoarece caută divizorii într-un set de numere și astfel consumă mult timp de execuţie. Acest algoritm are două mari neajunsuri:

1. Parcurge toate numerele de la min(`$a`, `$b`) până la 2;

2. Caută divizorii începând cu numerele mai mari.

Evident că este suficient de căutat divizorii comuni doar printre numerele prime și de verificat la ce putere acestea sunt divizori comuni.

O optimizare ar fi căutarea divizorilor comuni începând cu numerele mai mici. Dacă `$min` este un număr compus, atunci primul divizor comun se află pe o poziţie nu mai mare ca rădăcină din `$min`. Deci divizorii comuni ai `$a` și `$b` în caz general sunt mai aproape de începutul liste de numere parcurse.

Ambele optimizări ar fi posibile dacă am avea o listă de numere prime suficient de mare. Însă nu e garantat că algoritmul se va executa mai rapid, în special în limbaje de programare de tipul PHP, deoarece timpul de acces la lista de numere prime poate fi destul de mare ca să nu obţinem nici un câştig de timp.

O optimizare semnificativă ar fi să utilizăm Algoritmul lui Euclid de aflare a CMMDC a două numere. Esenţa Algoritmului lui Euclid este relaţia (a, b) = (b, r), unde `r = a % b` și `r < b`. În caz general numărul a poate fi mai mic decât b, însă aceasta doar mai adaugă un pas la procedeu.

E suficient să descriem algoritmul pentru numere nenegative, iar pentru cele negative se aplică același algoritm asupra valorilor absolute ale numerelor.

Pot fi compuse funcţii de câteva feluri: din punctul de vedere al structurii funcţiei, putem folosi algoritmi recursivi sau iterativi, iar din punctul de vedere al operaţiilor utilizate putem folosi algoritmi multiplicativi sau aditivi. Astfel avem patru cazuri posibile:

2) Algoritmul iterativ-multiplicativ

```
function divCom($a, $b)
{
   while($b) {
     $r = $a % $b;
     $a = $b;
     $b = $r;
   }
   return $a;
}
```

Funcţia iterativ-multiplicativă repetă întocmai modelul matematic al Algoritmului lui Euclid. Dintre toate funcţiile prezentate în paragraful curent aceasta este cea mai optimă din toate punctele de vedere (viteză, consum de memorie și valorile acceptate). Numărul de instrucţiuni este minim. Rezultatele testelor de viteză sunt prezentate într-un tabel mai jos. Consumul de memorie se reduce doar la trei variabile de tip `INT` (`$a, $b` și `$r`). Având în vedere modul de efectuare a instrucţiunii de împărţire de către UCP, această funcţie acceptă ca valori orice pereche de numere întregi.

O mică optimizare poate fi obţinută cu excluderea primului pas al iteraţiei în cazul `$a < $b`, prin schimbul cu locul al acestor două variabile:

```
if($a < $b) { $r = $a; $a = $b; $b = $r; }
```

Însă câştigul de timp este doar diferenţa dintre o operaţia de împărţire şi una de atribuire, ceea ce în general este nesemnificativ.

3) Algoritmul recursiv-multiplicativ:

```
function divCom($a, $b)
{
   if($b == 0) return $a;
   return divCom ($b, $a % $b);
}
```

Funcţiile recursive în general se deosebesc prin eleganţă și simplitate (de multe ori aparentă). Mulţi autori de manuale nu recomandă utilizarea funcţiilor recursive decât în cazuri excepţionale, deoarece greșelile și erorile legate de acestea sunt adesea depistate foarte greu.

Problema însă mai are și alt aspect – consumul resurselor calculatorului, memorie și timp. Fiecare apel recursiv al funcţiei alocă memorie pentru încă un set de variabile folosite de către aceasta și memorie pentru adresa de revenire a *indicatorului*

de instrucţiuni[12] la adresa precedentă. PHP este un limbaj de tip interpretor și, spre deosebire de limbajele de tip compilator, fiecare apel de funcţie consumă mai multe resurse şi fiecare declaraţie de variabilă consumă extra-memorie. Din acest punct de vedere, funcţia recursiv-multiplicativă rămâne cu un pas în urmă faţă de sora iterativă.

4) Algoritmul iterativ-aditiv(`$a > 0 && $b > 0`)

```
function divCom($a, $b)
{
   while($a != $b)
     if($a>$b) $a=$a-$b;
     else      $b=$b-$a;
   return $a;
}
```

La prima vedere s-ar părea că această funcţie trebuie să fie mai rapidă decât echivalentul multiplicativ, deoarece operaţia de scădere necesită mai puţine cicluri ale UCP. Aparenţa însă este falsă. Aceasta o demonstrează și rezultatele testărilor efectuate. Cauza este numărul mare de operaţii efectuate. În anumite cazuri, în loc de o împărţire se efectuează câteva mii (milioane, miliarde…) de operaţii de scădere, comparaţie şi salt condiţionat. Totuşi pentru numere de ordinul miilor funcţia este mai rapidă decât cea recursiv-multiplicativă, ceea ce demonstrează încă o dată ineficienţa recursiei în PHP.

Un alt punct slab al acestei funcţii este domeniul valorilor admisibile (DVA). Ușor se observă că pentru `$a < 0` sau `$b < 0` algoritmul reprezintă o iteraţie infinită, deci numerele negative se exclud din DVA. Această problemă se rezolvă ușor cu ajutorul următoarelor instrucţiuni plasate la începutul funcţiei:

```
$a = abs($a); $b = abs($b);
```

Însă perechile de numere ce conţin un singur `0` rămân în afara DVA. Ele pot fi adăugate la DVA prin secvenţa: `if(!$a ^ !$b) return 1;`

Funcţia iterativ-aditivă este o bună alternativă pentru cea recursiv-multiplicativă în cazul când din anumite motive nu putem folosi instrucţiunea de împărţire asupra valorilor date.

---

[12] În limbajele de tip compilator, indicatorul de instrucţiuni este un registru al UCP (IP = Instruction Pointer), care indică adresa de memorie a instrucţiunii curente şi care auto-avansează pe măsură ce se execută instrucţiunile din segmentul de cod (CS). Un apel de funcţie se efectuează printr-un salt al IP la adresa funcţiei şi apoi revenirea IP la adresa precedentă.

5) **Algoritmul recursiv-aditiv** (`$a > 0 && $b > 0`)

```
function divCom($a, $b)
{
   if($a > $b) return divCom($a - $b, $b);
   if($a < $b) return divCom($a, $b - $a);
   return $a;
}
```

Această funcţie moşteneşte toate dezavantajele surorilor recursiv-multiplicativă și iterativ aditivă. Totuși pot fi găsite două avantaje și pentru această funcţie:

Pentru valorile nepozitive, funcţia nu intră într-un ciclu infinit, deoarece fiind recursie, umple repede memoria operativă disponibilă şi se opreşte execuţia cu o eroare, care dacă nu e fatală (în alte limbaje), poate fi prelucrată de funcţia apelantă. În orice caz, utilizatorul nu va fi nevoit să aștepte rezultatul, spre deosebire de funcţia iterativ-aditivă.

Un alt avantaj este forma grafică frumoasă a codului funcţiei, astfel că se memorizează ușor ☺.

Rezultatele testărilor algoritmilor de aflare a CMMDC a două numere:

(Numerele testate: 15657, 88638)

Repetări / Algoritmul	100	1000	10000
Greedy – iterativ	0.330	3.340	34.100
recursiv-aditiv	0.0035	0.036	0.362
recursiv-multiplicativ	0.001	0.011	0.104
iterativ-aditiv	0.0007	0.0065	0.066
iterativ-multiplicativ	0.0004	0.0035	0.033

6) **Algoritmul CMMMC:**

Cu ajutorul uneia din funcţiile analizate mai sus, putem scrie funcţia pentru aflarea celui mai mic multiplu comun a doua numere:

```
function mulCom($a, $b)
{
  return (int)( $a*$b / divCom($a, $b) );
}
```

Evident (α, β, γ) = ((α, β), γ) și [α, β, γ] = [[α, β], γ]. Aceste proprietăţi rezultă din reprezentarea în forma canonică a CMMDC și a CMMMC a trei numere și pot fi aplicate la o cantitate oricât de mare de numere. Funcţiile pentru calcularea CMMDC şi a CMMMC a unei liste arbitrare de numere sunt prezentate în anexe. Ambele funcţii au la bază algoritmul de aflare a CMMDC a două numere.

II.3. Forma canonică a numerelor naturale (ℕ).

De multe ori reprezentarea în forma canonică a numerelor naturale ușurează cu mult rezolvarea unor probleme aritmetice. Procesul de obținere a acestei reprezentări se numește *factorizarea numărului*. Acest proces este destul de complicat din punct de vedere computațional și necesită determinarea primalității unor numere – problemă mai simplă, dar totuși și ea de un nivel înalt de dificultate. Există mai mulți algoritmi de determinare a primalității. Unii sunt deterministi, iar alții probabilistici. Algoritmii deterministi ca regulă necesită mai mult timp de lucru, însă dau un răspuns exact: numărul dat n este prim sau compus. Algoritmii probabilistici, pe de altă parte, consumă mai puțin timp de execuție și pentru unele numere determină cu exactitate că sunt compuse, iar celelalte rămân ca posibil prime. Probabilitatea primalității poate fi măsurată, iar cu un număr ales de iterații poate fi obținută o probabilitate arbitrar de mică.

În cadrul aplicației web dezvoltate, folosesc un algoritm determinist de factorizare a numerelor întregi mici (până la mărimea unui număr întreg pe 32/64 biți). Algoritmul este determinist, pentru că se bazează pe un algoritm determinist de testare a primalității numerelor. La factorizare se parcurge o listă de numere prime cunoscute și se verifică dacă numărul cercetat se împarte la acestea și de câte ori. Lista se extinde automat după necesitate. Pentru extinderea listei de numere prime se folosește un algoritm bazat la fel pe lista cunoscută de numere prime.

Sunt mulți algoritmi mai sofisticați de determinare a primalității numerelor întregi, însă pentru scopul aplicație, algoritmii descriși sunt suficienți.

Astfel, în cadrul aplicație poate fi obținută reprezentarea în forma canonică a oricărui număr întreg de mărimea procesorului și invers – din forma canonică a numărului se poate de obținut însuși numărul.

II.4. Descrierea aplicaţiei PHP

Aplicaţia are două părţi mari:

a. bibliotecile de funcţii și clase scrise în limbajul PHP;
b. aplicaţia web care foloseşte aceste biblioteci prin funcţii API.

A doua parte este strâns dependentă de prima. Iar prima a fost creată integral în baza concepţiilor descrise în această lucrare.

Elementele cheie ale bibliotecilor PHP sunt următoarele:

1. Clasa **`PkInts`** (Packed Integers):

Instanţele[13] acestei clase sunt niște containere de numere întregi păstrate în forma binară într-un șir „continuu” de octeţi. Fiecare număr întreg se reprezintă pe 1, 2, 4 sau 8 octeţi, după necesitate. Clasa introduce metode pentru accesare numerelor în diferite feluri (ca numere întregi sau naturale) și a fiecărui octet în parte, metode pentru manipularea listei de numere: scrierea/citirea listei în/din fișier, selectarea unei submulţimi de numere, sortarea listei, căutarea unui element, atașarea și eliminarea unui element, deplasarea cu un număr de biţi a listei, proprietăţi de accesare secvenţială, ș.a.

În PHP sunt destule instrumente pentru lucrul cu listele, în baza tipului de date **`array`**, însă au un minus semnificativ: pentru fiecare element al listei se alocă extra-spaţiu de memorie (~60 octeţi), deoarece lista poate conţine elemente de orice tip în mod arbitrar. Astfel numărul de elemente pe care le poate conţine un **`array`** în PHP este destul de mic (de ordinul miilor, în funcţie de memoria disponibilă). Clasa `PkInts` se specializează pe liste de numere întregi, astfel poate prelucra liste de până la câteva milioane elemente (limită în funcţie de timpul de execuţie, nu de memoria disponibilă). Un dezavantaj al clasei este timpul de acces al elementelor. Însă acest dezavantaj este compensat de timpul de scriere/citire în fișier, care poate fi ignorat (10^6 numere de 64biţi ocupă un șir de 8Mo, care se scrie/citește fără transformări în/din fișier).

[13] O instanță a clasei este un obiect al clasei respective, reprezentat de un sau mai multe (sau nici una) variabile de tipul clasei.

2. Clasa **`Prims`**:

Clasa `Prims` este un *Singleton*[14]. Folosește clasa `PkInts` pentru a prelucra și a păstra pe server o listă de numere prime, care la necesitate se folosesc în unele metode ale clasei `Prims` (și nu mai). Folosind metodele `Prims`, lista de numere prime se extinde în mod automat după necesitate, calculându-se numere prime din ce în ce mai mari. Încărcarea în memorie a listei la execuţia aplicaţiei nu se observă, chiar dacă lista este foarte mare (după măsurile PHP).

La momentul de faţă, lista conţine 270000 numere prime, ultimul prim fiind 3800201 (al 270000-lea). Aceasta este mai mult decât suficient pentru testul de primalitate a numerelor întregi reprezentate pe 32 biţi în baza listei de numere prime. Și aceasta nu e limita! Lista poate fi extinsă printr-o comandă API la server. În 30 de secunde (timpul oferit de serverul gazdă actual http://duzun.teologie.net/), la listă se pot adăuga în jur de 10000 numere prime noi.

De asemenea clasa conţine metode pentru determinarea CMMDC şi a CMMMC a unei liste arbitrare de numere întregi, testarea primalităţii unui număr întreg dat, găsirea celui mai apropiat prim mai mare sau mai mic decât numărul dat (se folosește căutarea prin metoda înjumătăţirii de complexitate O(log n), oferită de clasa `PkInts`), descompunerea numărului în forma canonică, ș.a.

3. Clasa **`LongNum`** (Long Numbers):

Este o extindere a clasei `Pkints` care introduce metode pentru efectuarea operaţiilor aritmetice cu numere întregi de lungime arbitrară (multi-precizie) fără pierderea valorii (ca în cazul numerelor de lungime fixată). Clasa garantează că lungimea numerelor se modifică în funcţie de valoarea conţinută astfel ca rezultatul operaţiilor aritmetice să coincidă cu valoarea matematică (nu a claselor de resturi după modulul 2^n).

Clasa la fel introduce metode de citire și scriere a numerelor sistematice reprezentate în orice bază de la 2 până la 36 (se poate de extins). În calitate de cifre se folosesc cifrele zecimale şi literele alfabetului englez. Cel mai mult timp de execuţie se consumă la convertirea numerelor (reprezentate binar în memorie) într-o bază

[14] Clase care pot avea o singură instanţă.

arbitrară, diferită de 2^k. Operaţiile aritmetice, însă, se efectuează foarte rapid. Fiind o extensie a clasei `PkInts`, numerele reprezentate binar pot fi păstrate în fișiere. Astfel practic nu se pierde timp pentru convertirea din reprezentarea sistematică în cea binară. Iar dacă baza este de forma 2^k, clasa folosește algoritmi optimi de convertire în/din baza respectivă, făcând uz de principiile descrise în lucrare de faţă.

În afară de aceste clase, biblioteca mai conţine şi alte funcţii PHP pentru operarea cu numerele întregi. De asemenea sunt funcţii JavaScript (JS) care fac uz de tehnologia AJAX (folosită și de Google) pentru comunicarea dinamică dintre server și browser. Funcţiile JS folosesc interfaţa API a bibliotecii pentru a comanda cu operaţiile care se execută pe server.

Însăşi aplicaţia constă din module independente care folosesc biblioteca pentru operaţii cu numerele. Modulele pot fi adăugate sau eliminate fără a defecta funcţionarea aplicaţiei în întregime.

Codul bibliotecilor folosesc instrumentele și posibilităţile oferite de POO[15], astfel fiind comode în utilizare de către programatori. Iată, de exemplu, cum se efectuează adunarea unui număr întreg `$i` la o instanţă a clasei `LongNum` – `$o`, cu ajutorul proprietăţilor de accesare secvenţială ale clasei `PkInts`:

```
function add_int_i($o, $i) {
   $carry = 0; // transportul initial lipseste
   $o->current = add_carry($o->reset, $i, $carry);
   $g = ($i & INT_SIGN) ? -1 : 0;
   $v = $o->next;
   while($v !== false && (($carry ^ $g) & 1)) {
      $o->current = add_carry($v, $g, $carry);
      $v = $ o->next;
   } return $carry;
}
```

Proprietăţile `reset`, `current` și `next` sunt moștenite de la `PkInts`, și sunt folosite pentru parcurgerea listei. Alte variabile contor nu se utilizează.

[15] Programarea Orientată pe Obiecte

CONCLUZII

Nu putem nega faptul că în zilele noaste calculatoarele au o influenţă foarte mare asupra tuturor aspectelor vieţii noastre. Calculatoarele au schimbat modul nostru de a percepe realitatea şi de a interacţiona cu realitatea și cu oamenii din jur. Aceasta se simte în mod accentuat în ultimele 2-3 decenii. Cercetarea și dezvoltarea de mai departe a matematicii şi a ştiinţei în general este în strânsă legătură cu dezvoltarea sistemelor de calcul. Pană nu demult capacitatea calculatoarelor personale (memoria și viteza de operare) se dubla în fiecare 1-2 ani. Acum însă se observă o creștere mai lentă în ceea ce privește viteza de operare a calculatoarelor, deoarece s-a atins limita fizică a capacităţii circuitelor integrate. Pentru îmbunătăţirea performanţelor calculatorului se merge pe alte căi. O cale este mărirea numărului de procesoare (UCP cu 2, 4 sau 8 nuclee) ale unui calculator, iar alta este folosirea mai multor calculatoare interconectate într-o reţea foarte rapidă numită cluster, astfel ca toate să lucreze pentru soluţionarea aceleiaşi probleme. Aceste căi, însă, necesită soft mai sofisticat care să folosească avantajele interconectării mai multor procesoare/nuclee.

O altă cale de dezvoltare este folosirea algoritmilor cât mai optimi pentru fiecare problemă în parte. Această cale este actuală începând cu apariţia sistemelor de calcul și până în zilele noastre. Însă pentru a crea și a folosi cu succes algoritmi optimi este nevoie de cunoştinţe matematice şi de înţelegerea implementării noţiuni de număr în sistemul de calcul dat (sistemul de calcul și se numește „de calcul”, pentru că operează cu numere). O piedică în înţelegerea modului de operare a calculatorului electronic este faptul că nu suntem obişnuiţi cu sistemul binar de numeraţie şi cunoaştem prea puţine proprietăţi ale numerelor şi operaţiilor binare.

În lucrarea de faţă am încercat să analizez principiile de bază de implementare a teoriei divizibilităţii în sistemele de calcul din familia 80x86. De asemenea am descris unele principii și metode computaţionale care permit extinderea noţiunii de număr în sistemele de calcul. Aplicând aceste principii și metode am reușit să creez o structură programatică de număr întreg de lungime variabilă, în funcţie de valoarea conţinută. Ca dovadă a succesului concepţiilor descrise serveşte aplicaţia web despre care s-a vorbit mai sus. Fără aceste noţiuni ar fi imposibil de realizat aplicaţia dată în forma în

care este, considerând limitele de memorie şi de timp de execuţie ale aplicaţiilor PHP (~128Mb RAM, 30-60sec).

Bibliotecile PHP ale aplicaţiei pot fi extinse, la fel ca şi aplicaţia în întregime. Clasa `PkInts` poate fi extinsă într-o clasă care ar reprezenta polinoamele peste inelul numerelor întregi. Clasa `Prims` poate fi completată cu metode pentru rezolvarea congruenţelor şi mai apoi pentru rezolvarea ecuaţiilor liniare cu două necunoscute cu ajutorul congruenţelor. Lista de numere prime poate fi utilizată şi în alte aplicaţii web chiar de pe alte servere-gazdă, prin intermediul interfeţei API, astfel oferind un serviciu web de nivel aplicaţie. În acest fel poate fi utilizată toată biblioteca pentru crearea de noi aplicaţii.

Aplicaţia bibliotecilor, cât și a teoriei divizibilităţii în sistemele de calcul este largă, iar aici este descrisă doar o mică parte a posibilităţilor. Însă prima condiţie este studierea şi înţelegerea în esenţă a fiecărei noţiuni în domeniul în care se utilizează.

BIBLIOGRAFIE

1. Achour M., Betz F., Dovgal A., Lopes N, Magnusson H., Richter G., Seguy D., Vrana J., ş.a., *PHP: Manual PHP*, 2010 (http://php.net/manual/)
2. Hyde R., *The Art of Assembly Language Programming*, California 1996. 1426 p. (http://oopweb.com/Assembly/Documents/ArtOfAssembly/Volume/toc.html)
3. Menabrea L. F., *Sketch of The Analythical Engine invented by Charles Babbage with notes by translator Ada Lovelace*, from the *Bibliothèque Universelle de Genève*, October, 1842, No. 82 (http://www.fourmilab.ch/babbage/sketch.html)
4. Rustem P., *Analiza şi sinteza sistemelor numerice*, Galaţi, 2002. 180 p.
5. Valuţă I., *Schiţe de istorie a matematicii*. Chişinău, 2004.

ANEXĂ

CMMDC şi CMMMC a unei liste arbitrare de numere întregi

```
/*! Cel mai mare divizor comun.
 *
 * Sintaxa: CMMDC(mixed $arr[, mixed $arr1[, mixed $arr2...]])
 */
function CMMDC($arr)
{
  if(is_array($arr)) {
     // daca toate sunt pozitive si exista 1,
     // nu se vor mai parcurge celelalte elemente
     sort($arr, SORT_NUMERIC);
     $r = array_pop($arr);
     foreach($arr as &$v) {
        if($r == 1 || $r == -1) break;
        $r = divCom($r, CMMDC($v));
     }
     $arr = $r;
  }
  if(func_num_args()==1) return $arr;
  for($i=1; $i < func_num_args(); $i++) {
     if($arr == 1 || $arr == -1) break;
     $v = func_get_arg($i);
     $arr = divCom($arr, CMMDC($v));
  }
  return $arr;
}

/*! Cel mai mic multiplu comun.
 *
 * Sintaxa: CMMMC(mixed $arr[, mixed $arr1[, mixed $arr2...]])
 */
function CMMMC($arr)
{
  if(is_array($arr)) {
     $r = array_pop($arr);
     foreach($arr as &$v) $r = mulCom($r, CMMMC($v));
     $arr = $r;
  }
  if(func_num_args()==1) return $arr;
  for($i=1; $i < func_num_args(); $i++) {
     $v = func_get_arg($i);
     $arr = mulCom($arr, CMMMC($v));
  }
  return $arr;
}
```