Domagoj Matijević i Ninoslav Truhar

Uvod u računarstvo

Odjel za matematiku
Sveučilište Josipa Jurja Strossmayera u Osijeku
Osijek, 2012.

Izdavač: Sveučilište J. J. Strossmayera u Osijeku, Odjel za matematiku

Recenzenti: doc.dr.sc. Ivica Nakić
prof.dr.sc. Goran Martinović

Lektor: Tatjana Ileš, prof.

Tehnička obrada: doc.dr.sc. Domagoj Matijević
prof.dr.sc. Ninoslav Truhar

Tisak: Grafika d.o.o., Osijek

CIP zapis dostupan u računalnom katalogu Gradske i sveučilišne knjižnice Osijek pod brojem 120830048.

ISBN 978-953-6931-51-4

Udžbenik se objavljuje uz suglasnost Senata Sveučilišta J.J. Strossmayera u Osijeku pod brojem 17/12.

Udžbenik se tiska uz novčanu potporu Ministarstva znanosti, obrazovanja i sporta.

Sadržaj

Predgovor

Računarstvo je znanstvena disciplina koja se bavi proučavanjem teoretskih osnova informacije i računanja, te njihovim implementacijama i primjenama u računalnim sustavima. Temeljno pitanje kojim se bavi računarstvo je: koji se računalni procesi mogu učinkovito automatizirati i provoditi? Kako bi riješili ovo naizgled jednostavno pitanje, računalni znanstvenici rade u brojnim komplementarnim područjima. Unutar računarstva proučava se sama priroda računalnih problema kako bi se utvrdilo koji problemi jesu ili nisu rješivi. Nadalje, uspoređuju se različiti algoritmi kako bi se utvrdilo pružaju li oni točna i učinkovita rješenja promatranih problema. U tu se svrhu konstruiraju programski jezici što omogućuje učinkovitiju implementaciju takvih algoritama. Konačno, dobivene se rezultate primjenjuje od znanosti preko industrije do svakodnevnog života.

U tom kontekstu ovim udžbenikom želimo odškrinuti vrata računarstva i studente Odjela za matematiku motivirati kako bi više pažnje posvetili ovom području. Ovaj je tekst stoga prvenstveno namijenjen studentima Odjela za matematiku Sveučilišta J. J. Strossmayera u Osijeku za potrebe predmeta *Uvod u računarstvo* **I021** koji se sluša u 1. semestru preddiplomskog studija.

Svrha je ovoga teksta, kao i predmeta *Uvod u računarstvo*, upoznati studente s osnovnim idejama i metodama računarstva, na početnoj, ali i naprednoj razini, koje su osnova i za druge računalne kolegije. Nakon kratkog povijesnog pregleda razvoja računala te osnova građe računala, naglasak u prvom dijelu ovoga sveučilišnog udžbenika stavljen je na reprezentaciju brojeva u memoriji te aritmetiku s brojevima u računalima. Smatramo da je poznavanje spomenutoga nužan preduvjet za razumijevanje pisanja programskoga koda. U drugom dijelu knjige, naglasak je stavljen na razumijevanje proceduralnog pisanja programskoga koda u `C/C++` programskom jeziku. Također je naveden niz primjera i zadataka koji čitatelju trebaju omogućiti lakše svladavanje tehnike proceduralnog programiranja.

Zašto `C/C++`, a ne neki od jezika poput Pythona ili JavaScripta koji se u posljednje vrijeme veoma često spominje kao pogodniji izbor programskog jezika za srednje škole. Oba autora ovoga udžbenika smatraju kako `C/C++`, unatoč tome što zasigurno nije jednostavan jezik za učenje programiranja, omogućava kvalitetniju kontrolu i shvaćanje onoga što se uistinu događa pri samome izvršavanju programskoga kôda. Također, ovaj je tekst, kao što smo već napomenuli, namijenjen prvenstveno studentima matematike za koje je `C/C++` programski jezik odličan izbor pri implementacijama različitih numeričkih algoritama koji ih očekuju tijekom studija.

Tekst je knjige organiziran na sljedeći način:

Poglavlje 1 (Povijesni razvoj računalnih sustava) sadrži kratki pregled povijesti računarstva, pregled prvih računalnih pomagala, opis nekih mehaničkih računala te modernih računala, uključujući pojavu mikroprocesora i osobnih računala.

Poglavlje 2 (Osnove računala) ukratko opisuje što je to računalo i kakva je njegova arhitektura.

Poglavlje 3 (Osnovni tipovi podataka) proučava različite pozicijske brojevne sustave s posebnim osvrtom na binarni, oktalni i heksadecimalni sustav. Također, u okviru se ovoga poglavlja proučava način prikaza racionalnih brojeva u računalu te pogreške koje pritom nastaju.

Poglavlje 4 (Planiranje i razvoj programa) sadrži osnovne pojmove o pisanju programa i kontrolne strukture te pregled programskih jezika.

Poglavlje 5 (Funkcije i potprogrami) opisuje način pisanja funkcije i potprograma (procedura), rekurzivnih funkcija, deklariranje lokalnih i globalnih varijabli te razumijevanje polja u računalima.

Poglavlje 6 (Uvod u algoritme) opisuje nekoliko algoritama za brzo i točno rješavanje nekih standardnih problema. Algoritmi su napisani u pseudokodu čija struktura omogućuje vrlo jednostavnu implementaciju u nekom od programskih jezika poput `C/C++`. Na temeljnoj se razini pokušava motivirati studente kako bi shvatili složenost izvršavanja algoritma, tj. broja operacija algoritma u odnosu na veličinu *inputa* na kojemu se algoritam izvršava.

Zahvale

Zahvaljujemo kolegama i studentima koji su savjetima, diskusijom ili strpljivim čitanjem ovoga teksta pridonijeli njegovu poboljšanju. Posebno zahvaljujemo Domagoju Ševerdiji na grafičkom oblikovanju naslovnice te Adamu Novagenu za ustupanje fotografije s naslovnice udžbenika (Quantum computing @ DeviantArt). Veliko hvala Petru Taleru i Ivanu Vazleru na marljivom prikupljanju programskih zadataka.

POGLAVLJE 1

Povijesni razvoj računalnih sustava

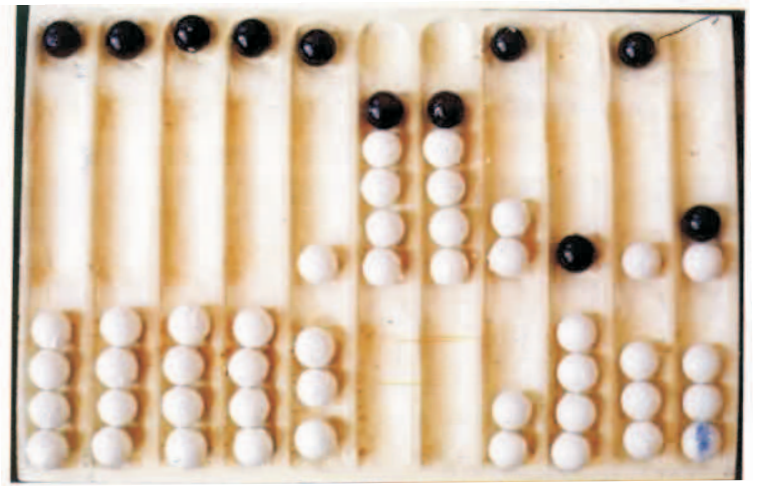
1.1 Prva računalna pomagala

Moglo bi se reći kako su prva računala zapravo bili ljudi. Elektronska su računala nazvana upravo po ljudima koji su obavljali poslove "računanja" i prije njihove pojave. Ti su poslovi obuhvaćali uzastopno izračunavanje navigacijskih tablica, pozicija planeta za astronomske potrebe ili matematičke izračune (npr. rješavanje linearnih sustava jednadžbi Jacobijevom metodom).

Lako je zamisliti kako na poslu, u kojemu se iz sata u sat i iz dana u dan računaju samo jednostavne računske operacije zbrajanja i oduzimanja te množenja i dijeljenja, može doći do pogreške. Stoga su se ljudi u najranijim razdobljima svoga postojanja trudili načiniti strojeve koji bi olakšali te "mukotrpne" poslove.

Najstarije poznato pomagalo pri izvođenju računskih operacija je **abakus** (slike 1.1 i 1.2).

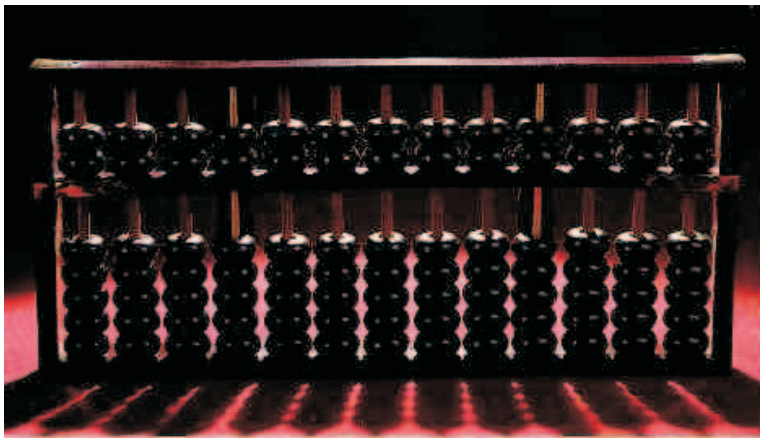
Zbrajanje i oduzimanje na abakusu može se izvoditi gotovo jednako brzo kao i na ručnom elektroničkom kalkulatoru, dok se množenje i dijeljenje izvodi sporije. Najstariji poznati abakus korišten je 300 godina p. n. e. u Babilonu, ali se i danas koristi i to najviše na Dalekom istoku (Kina).



Slika 1.1: Vrlo stara verzija abakusa. Slika preuzeta iz [4].

1.2 Mehanička računala

Škotski matematičar John Napier (1550.-1617.) otkriva logaritme i 1614. godine objavljuje prve logaritamske tablice u knjizi "Rabdologia". Otkriće logaritama bitno je pojednostavilo operacije množenja i dijeljenja, koje su u to vrijeme bile vrlo složene. Godine 1622. William Oughtred (1574.-1660.) i Edmund Gunter (1581.-1626.) izradili su cirkularno logaritamsko računalo. Od 1654. godine u uporabi su ravna logaritamska računala ili pomična računala, u nas popularno zvana "šiber". Popularnost pomičnih računala naglo je porasla nakon 1850. godine kada je francuski topnički časnik Victor Mayer Amédée Mannheim (1831.-1906.) računalu dodao klizni prozorčić s oznakama. Ta inačica pomičnog računala nazvana je *astrolab*, zbog česte primjene u astronomiji. U tom su se obliku pomična računala održala sve do 80-ih godina 20. stoljeća kad su ih u potpunosti zamijenili elektronički kalkulatori. Zanimljivo je istaknuti kako su NASA-ini inženjeri 60-ih godina prošloga stoljeća koristili "šibere" u Apollo programu koji je uspješno završio spuštanjem čovjeka na Mjesec.



Slika 1.2: Modernija verzija abakusa. Slika preuzeta iz [4].

Prvi poznati mehanički kalkulator izradio je 1623. godine njemački profesor Wilhelm Schickard (1592.-1635.), ali taj pronalazak dugo ostaje nepoznat zbog prerane profesorove smrti i tek je 1956. ponovno otkriven. Mnogo je poznatiji bio Pascalov kalkulator pa je slava oko izuma prvoga mehaničkog kalkulatora pripala velikom francuskom znanstveniku Blaiseu Pascalu.

Načelo djelovanja Pascalova kalkulatora temeljilo se na napravi za mjerenje prijeđenog puta kočije koju je u 2. stoljeću opisao Heron iz Aleksandrije. Prema istom načelu rade brojila prijeđenog puta u automobilima. Ni Pascalov niti Schickardov kalkulator nisu bili praktično primjenjivi zbog ograničenja tadašnje tehnologije koja nije omogućavala preciznu i pouzdanu izradu mehaničkih dijelova poput zupčanika, preciznih prijenosnih elemenata i sl.

Sljedeći veliki izumitelj koji se ogledao u izradi mehaničkog kalkulatora bio je Gottfried Wilhelm Leibniz (1646.-1716.). On je znatno pridonio napretku dvanaestak znanstvenih područja, od logike do lingvistike, a svoj kalkulator poznat kao "Leibnizov kotač" izradio je 1672. u Parizu.

Iako je taj kalkulator bio mnogo savršeniji od prethodnih i mogao je obavljati sve četiri osnovne aritmetičke operacije, nije bio pouzdan i upotrebljiv u praksi. Ograničenje je i opet bila tehnologija koja nije mogla slijediti Leibnizovu zamisao.

Mehanički su kalkulatori ostali gotovo nepromijenjeni sve do 1820. godine, kada Thomas De Colmar razvija komercijalno uspješan mehanički kalkulator koji je mogao izvoditi četiri osnovne računske operacije: zbrajanje, oduzimanje, množenje i dijeljenje. Različite inačice toga modela, poznatog pod engleskim nazivom *arithmometer*, proizvodile su se sljedećih stotinu godina.

Napretkom fine mehanike otvara se mogućnost izrade pouzdanih mehaničkih kalkulatora. Mehanički stolni kalkulatori doživljavaju najveći razvoj na prijelazu iz 19. u 20. stoljeće. Šveđanin Willgodt Theophil Odhner konstruira uspješan mehanički kalkulator i 1874. počinje proizvodnju i prodaju modela kojega su mnogi kasnije kopirali. Odhnerov kalkulator je u potpunosti zamijenio *arithmometar*. Amerikanac William S. Burroughs (1857.-1898.), pradjed kultnog pisca i pjesnika beat generacije Williama S. Burroughsa (1914.–1997.), godine 1886. konstruira vrlo uspješan model koji prodaje tvrtka *American Arithmometer Company*. Prvi mehanički kalkulator kojega je pokretao elektromotor proizvodi se od 1920. godine.

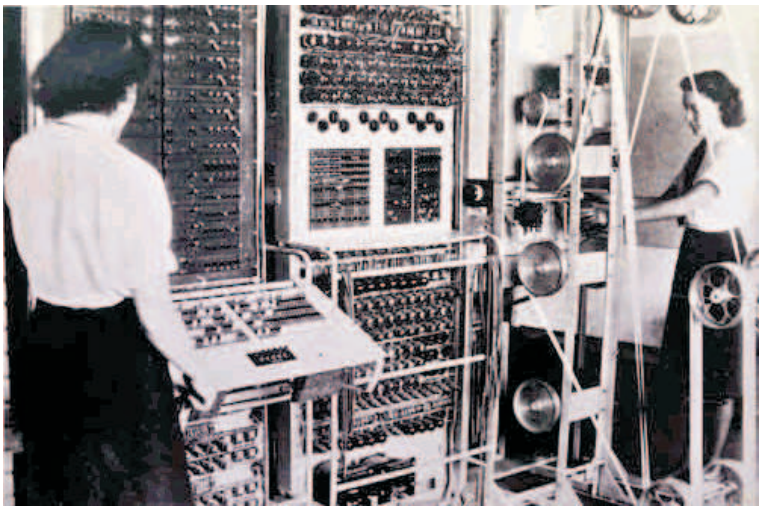
1.3 Moderna računala

Za vrijeme Drugog svjetskog rata količina potrebnoga računanja je toliko porasla da je primjerice 1944. u SAD-u količina proizvedenih topova ovisila o fizičkim mogućnostima ljudi koji su obavljali računanje ("ljudski kalkulator"). Stoga je SAD bio spreman financirati gotovo svaki projekt računala koji je obećavao razrješenje problema, pa se ubrzano počinju ostvarivati mnoge zamisli o računskim strojevima.

U isto se vrijeme u Engleskoj gradi elektronički programabilni kalkulator pod imenom **Colossus** (slika 1.3).

Prvo je računalo počelo raditi 1943., a napravljeno ih je ukupno deset. Računala su čuvana u najvećoj tajnosti. Na temelju toga iskustva, Englezi grade prvo posve elektroničko računalo (s elektroničkom memorijom) pod nazivom **Manchester Mark I**, koje je sadržavalo 2400 elektronskih cijevi. Komercijalno i praktično primjenjivo računalo pod nazivom EDSAC (engl. Electronic Delay Storage Automatic Calculator) izgradili su 1949. godine. U to su vrijeme Englezi tehnološki posve ravnopravni s Amerikancima. No na njihovu žalost, ubrzo napuštaju istraživanja i njihovo financiranje i do danas značajno zaostaju za Amerikancima.

Zanimljivo je spomenuti kako je engleski matematičar Alan Turing sa sveučilišta Cambridge u svom radu „Hypotetical Machine" opisao ustrojstvo modernoga računala (memoriju, CPU i dr.). Njegov je rad bio posve teorijske prirode



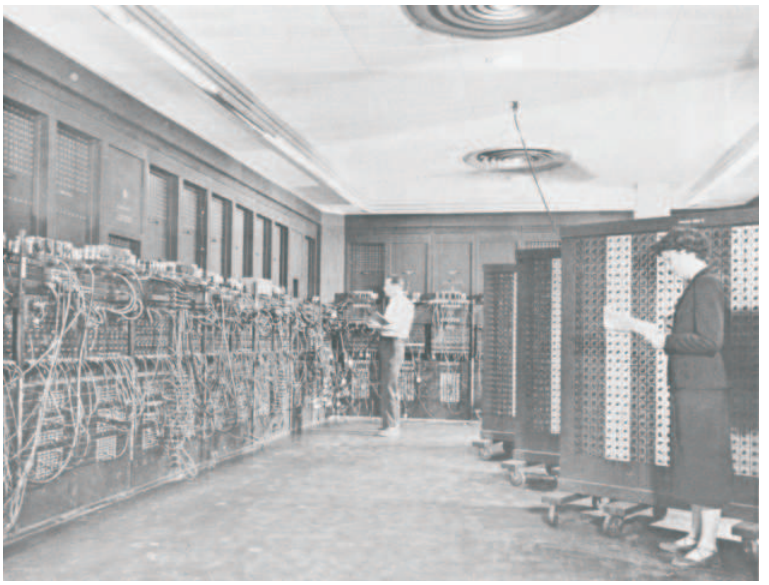
Slika 1.3: Stroj za dešifriranje Colossus. Slika preuzeta iz [4].

i Turing nikada nije pokazao zanimanje za gradnju računala prema navedenom načelu.

U SAD-u se istodobno rađa računalo pod nazivom ENIAC (engl. Electronic Numerator, Integrator, Analyzer and Computer), koje se općenito smatra prvim elektroničkim računalom (engl. computer). Razvijeno je na relativno malom i neuglednom američkom sveučilištu Moore School na prijedlog Johna W. Mauchlyja, a glavni inženjer projekta bio je John Presper Eckert, Jr. Tako se Mauchlyja i Eckerta smatra očevima modernih računala. Svakako valja spomenuti Johna Vincenta Atanasoffa koji je prvi osmislio ideju suvremene građe računala (način na koji je građena većina suvremenih računala i mikroprocesora), i velikog matematičara 20. stoljeća Johna von Neumanna, koji je teorijski obradio i ususstavio tu građu.

ENIAC (slika 1.4) je dovršen i pušten u rad u studenome 1945. godine, dakle tek nakon završetka Drugoga svjetskog rata. Sastojao se od 17468 elektroničkih cijevi, težio oko 30 tona i imao snagu 174 kW. Između ostaloga, ENIAC je služio za proračune prve hidrogenske bombe, a bio je u uporabi do 1955. godine.

Uskoro započinje razvoj sličnih računala, primjerice EDVAC-a, ILLIAC-a



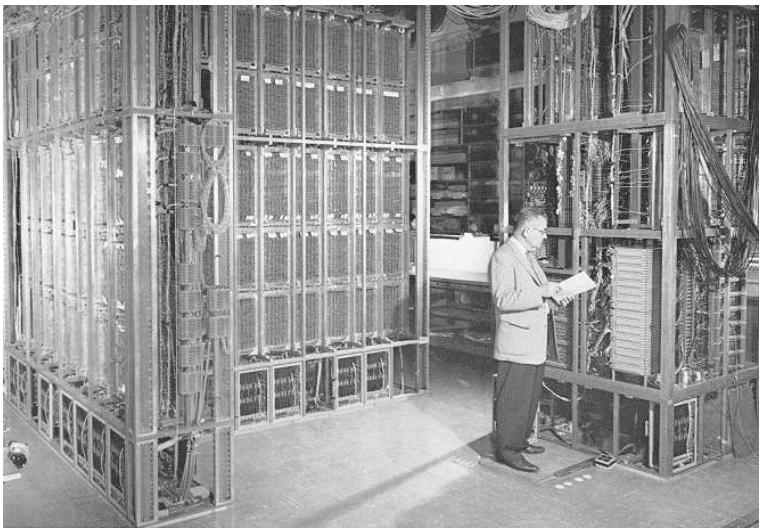
Slika 1.4: Prikaz računala ENIAC. Slika preuzeta iz [4].

(slika 1.5) i drugih. Sva su ona imala elektroničke cijevi i građena su najčešće u sklopu sveučilišta, a financirala ih je vlada SAD-a.

1.4 Poslovna primjena računala

Mauchly i Eckert u međuvremenu osnivaju vlastitu tvrtku za proizvodnju elektroničkih računala, ali ubrzo zapadaju u nepremostive financijske teškoće te su prisiljeni prodati tvrtku. Kupila ih je tvrtka Remington Rand, a Mauchly i Eckert ostaju glavni konstruktori i grade prvo komercijalno uspješno elektroničko računalo pod nazivom UNIVAC.

Za razliku od prvih primjeraka računala ostalih proizvođača koji su bili građeni za vojne potrebe, UNIVAC se prodavao poslovnim tvrtkama, a proizvelo ih se ukupno 46. Prvi primjerak računala UNIVAC I isporučen je u ožujku 1951. Uredu za popis stanovništva SAD-a.



Slika 1.5: ILLIAC II načinjen na sveučilištu u Illinoisu. Slika preuzeta iz [4].

Uz veliku promidžbu, računalo UNIVAC je 1952. godine upotrijebljeno za predviđanje rezultata predsjedničkih izbora u SAD-u. Na temelju samo 7% izbrojenih glasova, računalo je predvidjelo točan rezultat izbora. To je uvelike pridonijelo popularizaciji računala i pobudilo veliko zanimanje.

I dok su se IBM i UNIVAC bavili razvojem i prodajom velikih računala, koja su zbog visoke cijene mogle kupiti samo vlade pojedinih država ili velike tvrtke, Kenneth Harry Olsen 1957. godine osniva tvrtku DEC (Digital Equipment Corporation). Prepoznavši potrebu za malim i pristupačnim računalima, primjerenima jednostavnim problemima malih i srednjih tvrtki, Olsen se odlučuje za proizvodnju malih računala namijenjenih industrijskom upravljanju. Budući da je radio na projektu SAGE, osnovnu koncepciju i iskustvo prenio je na prvo računalo tvrtke DEC koje se pojavilo 1959. pod nazivom PDP-1.

Godine 1963. DEC proizvodi model PDP-8, prvo malo komercijalno uspješno računalo. Cijena toga računala iznosila je 18000 američkih dolara, pa su prvi puta i male tvrtke mogle kupovati i upotrebljavati računala. Godine 1977. DEC proizvodi 32-bitno malo računalo, pod nazivom VAX, koje postiže

velik poslovni uspjeh. Porodica VAX računala bila je tako uspješna da je u 5 godina prodano više od 10000 primjeraka. Računala su tako iz laboratorija i "posvećenih" prostora najvećih tvrtki ušla u široku uporabu. Gotovo neometana u području malih računala sve do potkraj 70-ih godina, tvrtka DEC doživljava nevjerojatan rast i 1983. godine ima 78000 zaposlenih.

1.5 Pojava mikroprocesora i osobnih računala

Razvojem tehnologije sve se više elektroničkih komponenti može smjestiti na pločicu poluvodiča. Godine 1970. su se napokon svi osnovni elementi računala mogli smjestiti na samo jednu pločicu poluvodiča veličine nekoliko desetaka kvadratnih milimetara, pa je tako nastao mikroprocesor. Jednostavno rečeno, to je računalo na jednoj pločici poluvodiča. Iste godine tvrtka INTEL tržištu nudi prvi mikroprocesor pod nazivom 4004.

Godine 1973. ista tvrtka proizvodi mikroprocesor 8080, prvi 8-bitni mikroprocesor, uvelike prihvaćen na tržištu jer omogućuje proizvodnju malih i vrlo jeftinih računala. INTEL je sljedeće desetljeće i pol bio gospodar tržišta mikroprocesorima s modelima 8086, 80286, 80386, 80486 te modelom Pentium.

Osobno računalo koje je doista otvorilo put osobnoj uporabi računala bilo je računalo tvrtke Apple (slika 1.6), posebno model Apple II. U Europi su tu ulogu imala računala Sinclair Spectrum i Commodore C64. Zbog svoje niske cijene bila su pristupačna gotovo svakome, što je omogućilo široku računarsku naobrazbu.

Računalo IBM PC koje se pojavilo 1981. prekretnica je kojom započinje vladavina osobnih računala. Od tada im neprestano raste snaga i snizuje se cijena.

Do sada računala dijelimo na 5 generacija. Svaku generaciju obilježava velik tehnološki napredak koji pomaže izgradnju manjih, jeftinijih i znatno učinkovitijih uređaja.

- ***1. generacija (1941-56)***

  Računala ove generacije karakteriziraju elektronske cijevi i magnetni bubnjevi koji se koriste kao memorija. Velikih su dimenzija (ispunjavaju čitave sobe). Zbrajanje i oduzimanje trajalo je **50-70 ms** (mili sekundi).

  Primjeri računala prve generacije su UNIVAC i ENIAC. Istaknimo kako je ENIAC (Electronic Numerator, Integrator, Analyzer and Computer) zauzimao 90 m^2, a za zbrajanje mu treba **200** μs (mikro sekundi), odnosno

Slika 1.6: Apple I je prodavan kao računalo tipa "uradi sam". Slika preuzeta iz [4].

2000 instrukcija u sekundi. Problem su mu česta oštećenja elektronskih cijevi uzrokovana zagrijavanjem.

- ***2. generacija (1957-64)***

  Osnovna obilježja 2. generacije su zamjena katodnih cijevi tranzistorima. Koristi se hibridna tehnika, na jednom modulu smješteno je više tranzistora, a kao unutarnja memorija koristi se magnetska jezgra.

  Iako je upotreba tranzistora omogućila proizvodnju manjih, bržih i jeftinijih računala i dalje postoji problem zagrijavanja i mogućih oštećenja. I dalje se za unos koriste bušene kartice, a za ispis pisači.

  Pojavljuju se i programski jezici, a brzina računala 2. generacije iznosi oko 1 MIPS-a, tj. milijun instrukcija u sekundi, što je oko 500 puta više od ENIAC-a.

- ***3. generacija (1965-71)***

Treću generaciju obilježava upotreba integriranih krugova. Razvoj tehnologije omogućio je proizvodnju minijaturnih tranzistora smještenih na jednom poluvodiču koji je nazvan "silikonski čip". To je omogućilo značajno povećanje brzine i učinkovitosti računala.

Ovu generaciju također obilježava i nov način unošenja i ispisivanja podataka. Umjesto do tada korištenih bušenih kartica, korisnici s računalom komuniciraju preko tipkovnice i monitora. Razvijaju se i **operacijski sustavi** (OS), što omogućuje primjenu različitih aplikacija istovremeno, pri čemu središnji program (OS) nadzire korištenje memorije. Brzina računala dostiže 10 MIPS-a, manja su i jeftinija te postaju dostupna širokom broju korisnika.

- ***4. generacija (1972-89)***

  Četvrtu generaciju obilježavaju mikroprocesori.

  Mikroprocesor se sastoji od velikog broja (preko tisuću) integriranih krugova smještenih na jednom "silikonskom čipu". Ono što je u prvoj generaciji računala bilo smješteno u čitavu sobu, sada se nalazilo na pločici veličine ljudskog dlana.

  INTEL je 1971. godine razvio čip 4004 na kojemu su bile smještene sve računalne komponente, od centralne procesorske jedinice i memorije do ulazno-izlazne jedinice. Godine 1981. IBM uvodi u uporabu prvo "kućno računalo", a 1984. Apple predstavlja *Macintosh*. Svakim danom ta "kućna računala" postaju sve moćnija, a mogućnošću njihova povezivanja pojavila se i mogućnost razvoja interneta.

  Četvrta je generacija omogućila razvoj grafičkih sučelja (Graphical User Interface – GUI), korištenje "miša", ... Dolazi do razvoja programskih jezika , `C/C++`, Pascal, ...

- ***Današnja računala***

  Današnja računala pripadaju tzv. petoj generaciji koju obilježava umjetna inteligencija.

  Visoko integrirana računala, čija je moć računanja od nekoliko 1000 MIPS-a, omogućila su razvoj umjetne inteligencije (npr. prepoznavanje glasa) koja se i dalje snažno razvija. Intenzivno se radi na primjenama kvantne, molekularne i nano tehnologije. Glavni cilj nove generacije računala je mogućnost glasovnog unošenje podataka i samoorganiziranja računala.

POGLAVLJE 2

Osnove računala

2.1 Što je računalo?

Računalo je stroj koji, u skladu s uputama definiranima u programu, može obrađivati podatke te izvoditi osnovne računske i logičke operacije. Termin **program** podrazumijeva skup instrukcija složenih određenim redoslijedom u skladu s kojima računalo izvodi gore navedene operacije s ciljem izvođenja određenog zadatka.

Na računalo možemo gledati s dvije osnovne razine. Prva se odnosi na "vidljivi" dio računala, a druga na njegovu primjenu. Vidljivi dio se najčešće naziva **arhitektura računala** ili **hardware**, a podrazumijeva dijelove računala poput monitora, pisača, tipkovnice, miša te drugih elektronskih dijelova računala.

Osnovne operacije koje računalo izvodi možemo podijeliti na:

- **Unos podataka** (input)

  Pri unosu podataka, računalo prihvaća podatke predstavljene na način na koje ih ono može razumjeti. U ovom se kontekstu podatak (data) odnosi na bilo kakav neorganizirani, sirovi, materijal sastavljen od riječi, brojeva, slika ili njihovih kombinacija.

- **Obrada** (processing)

Pri obradi računalo izvodi aritmetičke ili usporedne (logičke) operacije s predstavljenim podacima. Te operacije su jako jednostavne. U biti, većina stvari koje mikroprocesor (srce računala) stvarno izvodi temelji se na operaciji zbrajanja ili uspoređivanja dvaju brojeva. Međutim, budući da je brzina kojom izvodi te operacije izuzetno velika (milijuni u sekundi), to računalu omogućava širok spektar upotrebe. Druga činjenica koja je bila presudna za razvoj računarstva upravo je pouzdanost u obradi podataka. Čak i najjeftinija računala mogu godinama izvoditi nekoliko milijuna operacija u sekundi bez greške uzrokovane radom hardverskih komponenti. (Većina "pogrešaka računala" uzrokovano je logikom u programiranju ili početnim pogreškama u podacima koji se unose u računalo, o čemu ćemo detaljno govoriti kasnije).

- **Izlaz** (output)

  Treća operacija ima za cilj predstavljanje rezultata obrade na način razumljiv ljudima. Obrađeni podaci prosljeđuju se kao informacija.

- **Pohranjivanje** (storage)

  Računalo pohranjivanjem sprema rezultate kako bi se oni mogli kasnije koristiti. Količina podataka koju može spremiti i dobaviti jeftinije računalo mogla bi se usporediti s enciklopedijom tiskanom u 32 knjige.

2.2 Struktura računala

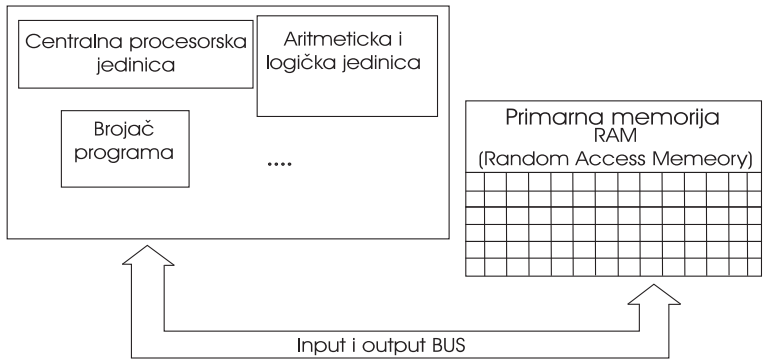
Većina računalnih sustava sadrži tri osnovne strukture:

- memoriju velike brzine
- centralnu jedinica za obradu
- ulazno-izlazne sklopove

Slika 2.1 prikazuje osnovnu shemu strukture računala.

2.2.1 Memorija velike brzine

Memorija o kojoj je ovdje riječ ugrađena je u samo računalo. To je ona memorija koju procesor (CPU) koristi za neposredno dobavljanje i pohranjivanje podataka, a zovemo ju i **glavna memorija** ili **fizička memorija**. Dok je računalo uključeno i obrađuje podatke, ti se podaci i programi nalaze u radnoj



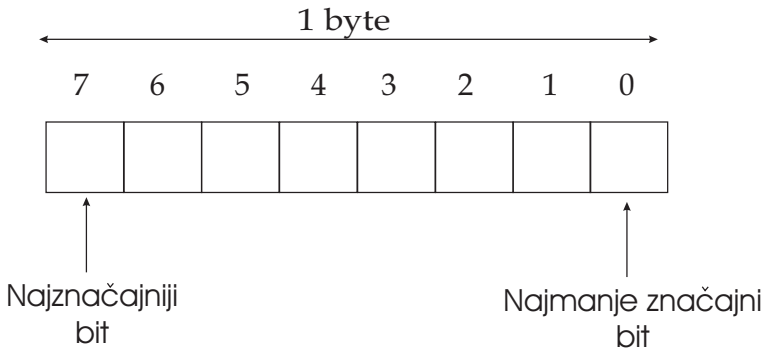
Slika 2.1: Osnovna struktura računala

memoriji. U većini se slučajeva memorija sastoji od čipova načinjenih od metalnog oksida na silikonskoj pločici. Takvu vrstu memorije nazivamo **RAM** što je skraćenica za *Random Access Memory*.

Memorija računala podijeljena je na logičke jedinice jednake veličine od kojih najjednostavniju nazivamo **byte** (čitamo bajt). Svaki se byte sastoji od 8 uzastopnih **bit**ova, ili **binarnih znamenki**. Pohranjivanje bita u memoriju računala podrazumijeva postojanje uređaja kao što je prekidač, koji može biti u jednom od dva stanja, upaljen ili ugašen. Jedno se stanje koristi za predstavljanje nule, a drugo za jedinicu. U primjenama često koristimo oznake 1B = jedan byte i 1b = jedan bit.

Memorijske lokacije možemo zamisliti kao niz pretinaca od kojih svaki ima svoju adresu. I ne samo da svaka memorijska jedinica ima svoju adresu, već se i unutar pojedinog bytea zna točan redoslijed bitova prema važnosti. Krajnje lijevi bit u byteu je bit najveće važnosti, a onaj krajnje desni je najmanje značajan bit. Kao bitnu posljedicu označavanja (adresiranja) byteova i bitova u njima, čitavu glavnu memoriju možemo predstaviti jednim dugačkim redom. Dijelovi toga reda mogu se iskoristiti za pohranjivanje uzorka bitova koji je dulji od jednog bytea. Obično ga spremamo u *string* koji je cjelobrojni višekratnik osnovne memorijske ćelije.

Osim toga, ovakav način organiziranja memorije omogućava izravan pristup sadržaju svake memorijske lokacije, bez obzira gdje se ona nalazi (kaotično – *random*). Stoga se ovakve memorije zovu RAM-ovi (*Random Access Memory*).



Slika 2.2: Shematski prikaz bytea

1 nibble	4 bita
1 byte	8 bita
1 riječ	2 bytea = 16 bita
1 duga riječ	4 bytea = 32 bita
1 kvadratna riječ	8 bytea = 64 bita
1 oktalna riječ	16 bytea = 128 bita

Tablica 2.1: Uobičajena imena za skupove bitova

Osim tih vrsta memorije, glavnu memoriju čine i ROM-ovi, memorije koje služe samo za čitanje. Kapacitet memorije mjeri se ukupnim brojem byteova koji se mogu u nju pohraniti. Današnja računala imaju 256 MB, 512 MB ili 1 GB i više kapaciteta glavne memorije.

Budući da se memorijski sustavi izrađuju tako da je ukupan broj lokacija potencija broja 2, veličina memorije kod starijih sustava često se mjerila jedinicom od 1024 bita, što odgovara broju 2^{10}. Kako je 1024 približno 1000, društveno je prihvaćen prefiks **kilo** kao dodatak osnovnoj jedinici. Stoga je termin kilobyte (KB) označavao 1024 bita, a za stroj sa 4096 adresnih lokacija govorilo se da ima 4 KB memorije. Kako je memorija postajala sve veća, u upotrebu dolaze termini megabyte (MB) ($1\text{MB} = 2^{20}\text{B} = 1048576\text{B} \approx 10^6\text{B}$) i gigabyte (GB) ($1\text{GB} = 2^{30}\text{B} = 1073741824\text{B} \approx 10^9\text{B}$).

Poželjno je da RAM bude što većeg kapaciteta kako bi se pohranilo što više

podataka, a što nadalje omogućava brži pristup većoj količini podataka pa je računalo s većim RAM-om pogodnije (brže) za korištenje.

Druga vrsta memorije je tzv. **ROM memorija** (*Read Only Memory*) ispisna memorija, memorija u koju se podatak može upisati samo jednom. Nakon upisa, podatak se može čitati onoliko puta koliko se želi, ali ne i mijenjati, brisati ili upisivati novi podatak. Zato je primjena ove memorije ograničena na pohranu podataka koji su uvijek jednaki i nepromijenjeni.

2.2.2 Centralna jedinica za obradu

Centralna jedinica za obradu (CPU – *Central Processing Unit*) je "mozak računala". U prethodnom dijelu pozornost je bila posvećena pohranjivanju podataka i organizaciji memorije u računalu. Jednom kada su podaci pohranjeni, stroj mora biti sposoban manipulirati tim podacima u skladu s naredbama algoritma. Dakle, stroj mora imati ugrađenu sposobnost izvođenja operacija i koordiniranja redoslijeda izvođenja operacija.

Centralna je jedinica za obradu (CPU) ili obično procesor, sastavljena od dva osnovna dijela: aritmetičko-logičke jedinice (ALU) u kojoj su sklopovi za manipulaciju podacima, i upravljačke jedinice koja koordinira aktivnostima unutar računala. Za privremeno čuvanje podataka CPU koristi registre opće i posebne namjene. Registri opće namjene služe kao privremeni spremnici podataka koji se obrađuju unutar CPU-a. U tim se registrima čuvaju vrijednosti koje ulaze u aritmetičko-logičku jedinicu i čuvaju se rezultati koji se iz AL jedinice tamo pošalju.

2.2.3 Ulazno-izlazni sklopovi

Ulazno-izlazni sklopovi (I/O sklopovi ili *Input-output* sklopovi) služe za povezivanje računala s okolinom. Pod okolinom se podrazumijeva sve ono što se nalazi izvan računala. Programe i podatke koji se u računalu obrađuju potrebno je na neki način dostaviti računalu, a isto je tako potrebno dobivene rezultate dostaviti korisniku.

Ulazni sklopovi su građeni tako da omogućuju priključivanje vanjskih jedinica pomoću kojih je moguće iz okoline podatke predavati računalu. Podaci tako ulaze u računalo pa se takvi sklopovi nazivaju ulaznim sklopovima. Izlazni sklopovi omogućuju priključivanje vanjskih jedinica pomoću kojih je podatke iz računala moguće predavati okolini. Podaci na taj način izlaze iz računala pa se takvi sklopovi nazivaju izlaznim sklopovima. U sklopu računala postoji više

ulazno-izlaznih sklopova koji omogućuju priključivanje najrazličitijih vanjskih jedinica, kao što su: tipkovnica, monitor, magnetski disk, zvučnici, itd.

Kako bi mogli izmjenjivati podatke, CPU i RAM povezani su skupom žica BUS-om. Preko BUS-a, CPU može dohvatiti ili pročitati podatke iz RAM-a prosljeđivanjem adrese odgovarajuće memorijske lokacije i signala za čitanje.

Na isti se način podaci mogu upisati u memoriju postavljanjem na BUS adrese lokacije odredišta, postavljanjem podatka i signala za pisanje. Slijedom ovih uputa, zadatak zbrajanja dvaju pohranjenih vrijednosti iz RAM-a uključuje mnogo više od provođenja same operacije zbrajanja. Proces uključuje koordiniranu akciju upravljačke jedinice koja upravlja prijenosom informacija između radne memorije i registra unutar CPU-a, i aritmetičko-logičke jedinice koja izvodi operacije zbrajanja u trenutku kada joj upravljačka jedinica da znak da ih izvede. Cjelokupan postupak zbrajanja dvaju vrijednosti pohranjenih u memoriju mogao bi se predstaviti procedurom od 5 koraka:

- Korak 1: Uzmi prvu vrijednost koja se zbraja i pohrani ju u registar.
- Korak 2: Uzmi drugu vrijednost koja se zbraja i pohrani ju u drugi registar.
- Korak 3: Aktiviraj sklopove za zbrajanje s vrijednostima koje su pohranjene u registrima tijekom prva dva koraka.
- Korak 4: Rezultat operacije pohrani u memoriju.
- Korak 5: Stani.

Ukratko, podaci se moraju prenijeti iz memorije u CPU, a rezultat se na kraju ponovno pohranjuje u memoriju.

Primjer 2.1 *Koliko 3 GB ima byta?*

Rješenje:

$$3\text{GB} = 3 \cdot 1024^3\text{B} = 3221225472\text{B}.$$

Zadatak 2.1 Računalo ima memoriju od 32 MB. Kolika je njegova stvarna memorija izražena u byteima?

Zadatak 2.2 Koliko iznosi 15 EB, preračunato u byte?

POGLAVLJE 3

Osnovni tipovi podataka

Brojevni sustavi mogu biti **pozicijski** i **nepozicijski**.

Najpoznatiji nepozicijski brojevni sustav, koji se i danas upotrebljava, je sustav rimskih brojeva, dok su većina brojevnih sustava koji se i danas koriste, upravo pozicijski sustavi.

Čovjek najčešće koristi dekadski brojevni sustav čije su znamenke 0, 1, . . . i 9.

3.1 Cijeli brojevi

Cijele brojeve $\mathbb{Z} = \{\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$ u području računarstva često nazivamo *integer* brojevima. Svaki cijeli broj možemo zapisati kao:

$$d_{n-1}d_{n-2}\ldots d_1 d_{0(10)} = d_{n-1}10^{n-1} + d_{n-2}10^{n-2} + \ldots + d_1 10^1 + d_0.$$

Gornji raspis nazivamo raspisom broja po bazi (u ovom slučaju radi se o bazi 10). Općenito, svaki broj $N = d_{n-1}d_{n-2}\ldots d_1 d_{0(B)}$ zapisan u nekoj bazi B možemo na jedinstven način zapisati u bazi 10 kao:

$$N_{(10)} = \sum_{i=0}^{n-1} d_i B^i.$$

Primjer 3.1

a) $768401_{(10)} = 7 \cdot 10^5 + 6 \cdot 10^4 + 8 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10 + 1 = 768401_{(10)}$

b) $768401_{(9)} = 7 \cdot 9^5 + 6 \cdot 9^4 + 8 \cdot 9^3 + 4 \cdot 9^2 + 0 \cdot 9 + 1 = 458866_{(10)}$

Primjer 3.2 *Izračunajte koliko je* $768401_{(8)}$ *u bazi 10?*

Rješenje: *Primijetite da problem nije valjano zadan, budući da znamenke u bazi 8 dolaze iz skupa* $\{0, 1, 2, 3, 4, 5, 6, 7\}$.

3.1.1 Binarni brojevni sustav

Posebno mjesto u svijetu računala zauzima binarni brojevni sustav i stoga ćemo mu posvetiti posebnu pažnju. U binarnom brojevnom sustavu brojevi su zadani u bazi $B = 2$, a znamenke dolaze is skupa $\{0, 1\}$. Svaki binarni broj možemo na sljedeći način raspisati po bazi:

$$b_{n-1}b_{n-2}\ldots b_1 b_{0(2)} = \sum_{i=0}^{n-1} b_i 2^i = N_{(10)}.$$

Primjer 3.3

$$\begin{aligned} 1001101_{(2)} &= 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 1 \\ &= 64 + 8 + 4 + 1 \\ &= 77_{(10)} \end{aligned}$$

Primjer 3.4 *Dan je niz od 4 bytea. Što predstavlja taj niz ako ga čitamo kao*
a) individualne byteove
b) dvije riječi, svaka s po 2 bytea
c) dugu riječ od 4 bytea?

01100011	*01100101*	*01000100*	*01000000*

Rješenje:
a) čitajući s lijeva na desno, dobivamo:

$$\begin{aligned}
01000000_{(2)} &= 0\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 \\
&= 64_{(10)} \\
01000100_{(2)} &= 0\cdot 2^7 + 1\cdot 2^6 + 0\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 \\
&= 68_{(10)} \\
01100101_{(2)} &= 0\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\
&= 101_{(10)} \\
01100011_{(2)} &= 0\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 \\
&= 99_{(10)}
\end{aligned}$$

b)

$$\begin{aligned}
0100010001000000_{(2)} &= "0^{15}1^{14}0^{13}0^{12}0^{11}1^{10}0^9 0^8 0^7 1^6 0^5 0^4 0^3 0^2 0^1 0^0" \\
&= 1\cdot 2^{14} + 1\cdot 2^{10} + 1\cdot 2^6 = 17427_{(10)} \\
0110001101100101_{(2)} &= "0^{15}1^{14}1^{13}0^{12}0^{11}0^{10}1^9 1^8 0^7 1^6 1^5 0^4 0^3 1^2 0^1 1^0" \\
= 1\cdot 2^{14} + 1\cdot 2^{13} &+ 1\cdot 2^9 + 1\cdot 2^8 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^2 + 1\cdot 2^0 \\
&= 24445_{(10)}
\end{aligned}$$

c) jedna duga riječ

$$\begin{aligned}
01100011011001010100010001000000_{(2)} &= "0^{31}1^{30}1^{29}0^{28}0^{27}0^{26}1^{25}1^{24}0^{23}1^{22} \\
1^{21}0^{20}0^{19}1^{18}0^{17}1^{16}0^{15}1^{14}0^{13}0^{12}0^{11}1^{10}0^9 0^8 0^7 1^6 0^5 0^4 0^3 0^2 0^1 0^0" \\
&= 1667580992_{(10)}
\end{aligned}$$

Pri pretvaranju binarnih brojeva u dekadske koristimo ideju cjelobrojnog dijeljenja s brojem 2 i pamćenja ostatka.

Primjer 3.5 *Pretvorite dekadski 232 u binarni broj.*

232	0
116	0
58	0
29	1
14	0
7	1
3	1
1	1
0	

Primijetite da se u desnom stupcu nalaze ostatci dijeljenja s 2 odgovarajućeg broja iz lijevog stupca. Na kraju, desni stupac čitamo odozdo prema gore i dobivamo odgovarajući binarni broj. Dakle, $232_{(10)} = 11101000_{(2)}$.

Pokušajmo opravdati valjanost gornjeg postupka. U tu svrhu raspišimo neki binarni broj $N_{(2)} = b_{n-1}b_{n-2}\ldots b_1 b_{0(2)}$ u raspisu po bazi:

$$N = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_2 2^2 + b_1 2 + b_0$$

Primijetite da je broj N djeljiv s 2 onda i samo onda ukoliko je broj $b_0 \in \{0, 1\}$ djeljiv s 2. Drugim riječima, ukoliko je N djeljiv s 2, tada mora vrijediti da je $b_0 = 0$. I suprotno, ukoliko N nije djeljiv s 2, tada mora vrijediti da je $b_0 = 1$. Ako ponovimo gornji postupak na broju

$$\lfloor N/2 \rfloor^1 = N_1 = b_{n-1}2^{n-2} + b_{n-2}2^{n-3} + \ldots + b_2 2^1 + b_1$$

dobit ćemo vrijednost znamenke b_1. Ako nastavimo postupak dijeljenja s 2 točno $n-1$ puta dobit ćemo vrijednosti svih binarnih znamenaka kao ostatke dijeljenja s 2.

Zašto računala 'vole' binarne brojeve? Registri u računalima skupina su povezanih *bistabila*. Bistabil je osnovni elektronički sklop (dio je memorije računala) u koji se može pohraniti jedna binarna znamenka (tj. 1 bit informacije), a može se nalaziti u dva stanja:

ima struje	nema struje
1	0

[1] S $\lfloor x \rfloor$ označavamo funkciju koja vraća najmanje cijelo od neke vrijednosti x. Npr. $\lfloor 2.7 \rfloor = 2$.

Točnije, kada je na bistabilu niži napon (oko 0V), tada bistabil 'prikazuje' znamenku 0, a kada je na njemu napon viši (oko 5V), tada 'prikazuje' znamenku 1. Obično se povezuje 8, 16, 32 ili 64 bistabila pa takav registar ima kapacitet isto toliko bitova.

3.1.2 Heksadecimalni i oktalni brojevni sustavi

Osim binarnog sustava, u svijetu računarstva najčešće je u upotrebi **oktalni sustav** s bazom 8 i **heksadecimalni sustav** s bazom 16. Sljedeća tablica prikazuje brojevne sustave i odgovarajuće znamenke.

SUSTAV	ZNAMENKE
Dekadski	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
Binarni	$0, 1$
Oktalni	$0, 1, 2, 3, 4, 5, 6, 7$
Heksadecimalni	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

Tablica 3.1: Znamenke brojevnih sustava

Kako se može vidjeti u tablici 3.2, svakom binarnom nizu (broju u binarnom sustavu) odgovara jedan i samo jedan broj u jednom od preostalih sustava. Tablica sadrži zapis brojeva $0, \ldots, 15$ u dekadskom (DEC), oktalnom (OCT), heksadecimalnom (HEX) i binarnom (BIN) sustavu.

Ekvivalencija ima za posljedicu da broj zapisan u jednom sustavu možemo pretvoriti u broj zapisan u nekom drugom sustavu, pri čemu je takav zapis jedinstven.

Pretvaranje brojeva iz binarnog u heksadecimalni sustav i obratno je jednostavno. Na primjer, ako je broj zapisan u heksadecimalnom sustavu, njegov binarni ekvivalent određujemo tako da svaku heksadecimalnu znamenku prikažemo kao 4 binarne znamenke (4 bita). Prisjetite se da s 4 bita možemo kodirati 16 različitih kombinacija (koliko je i znamenaka u heksadecimalnom sustavu).

Primjer 3.6 *Odredite binarni zapis heksadecimalnog broja* AF3B1.

Rješenje: *Uzimajući u obzir vrijednosti iz tablice 3.2 dobivamo da je* $\text{AF3B1}_{(16)} = 1010111100111011000 1_{(2)}$

Za pretvaranje binarnih brojeva u heksadecimalne potrebno je najprije binarni broj podijeliti u skupine od po 4 znamenke (počevši s desna na lijevo). Ako

DEC	OCT	HEX	BIN
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111

Tablica 3.2: Ekvivalencija zapisa u binarnom, oktalnom, dekadskom i heksadecimalnom sustavu

A	F	3	B	1
1010	1111	0011	1011	0001

nakon tog postupka posljednja skupina ima manje od 4 znamenke, potrebno joj je dodati nule na lijevo tako da i ona ima 4 znamenke. Nakon toga, svaku od tih skupina treba zapisati pomoću heksadecimalnih znamenki: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E i F.

Primjer 3.7 *Odredite heksadecimalni zapis binarnog broja*

10101111101010

***Rješenje**: Podijelimo najprije binarni broj u skupine od po 4 znamenke:*

10 1011 1110 1010

Budući da smo nakon podjele u posljednjoj skupini dobili broj 10*, dodavanjem dviju nula, broj koji ćemo pretvarati u heksadecimalni ima oblik:*

$$0010 \quad 1011 \qquad 1110 \qquad 1010$$

Uzimajući u obzir vrijednosti iz tablice 3.2, dobivamo

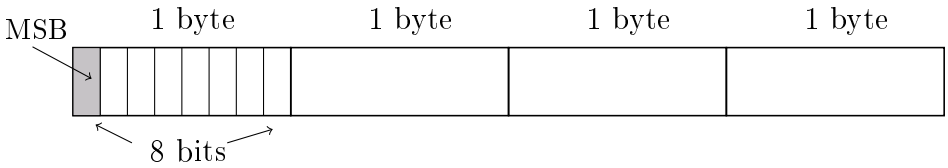
$$\begin{array}{cccc} 0\,0\,1\,0 & 1\,0\,1\,1 & 1\,1\,1\,0 & 1\,0\,1\,0 \\ 2 & B & E & A \end{array}$$

Tako je $10101111101010_{(2)} = 2\mathrm{BEA}_{(16)}$.

Zadatak za vježbu 3.1 Objasnite kako jednostavno možemo pretvarati brojeve iz binarnog u oktalni sustav i obratno.

3.1.3 Prikaz cijelih brojeva u računalu

Memorija računala sastoji se od bitova. Bit (Binary digit) može biti ili 1 ili 0. Najmanji memorijski prostor kojemu možemo pristupati (alocirati) je 1 byte. 1 byte se sastoji od 8 bitova. Uobičajeno se na modernim računalima za cijele brojeve alocira 4 bytea (ako je arhitektura računala zasnovana na 32 bita) ili 8 bytea (ako je arhitektura računala zasnovana na 64 bita) u memoriji računala, ukoliko korisnik ne odredi drugačije. Prvi bit je rezerviran za predznak broja i naziva se *most-significant bit* (MSB) (vidi sliku 3.1). Ako je MSB 0, broj je pozitivan, a ako je MSB 1, broj je negativan.



Slika 3.1: Cijeli broj spremljen u 4 bytea. Prvi bit (MSB bit ili 'most significant bit') rezerviran je za predznak broja.

Primjer 3.8 *a)* $232 = \underbrace{0}_{MSB}0000000\,|\,00000000\,|\,00000000\,|\,11101000$

$$b)\ -137 = \underbrace{1}_{MSB}0000000\,|\,00000000\,|\,00000000\,|\,10001001$$

Postavlja se pitanje koliki je raspon cijelih brojeva koje možemo spremiti u 32-bitni zapis na gore opisani način. Pokušajmo najprije shvatiti koji raspon brojeva možemo spremiti u 1 byte. Budući se 1 byte sastoji od 8 bitova, a svaki bit može poprimiti dvije vrijednosti, 1 ili 0, postoji $2^8 = 256$ različitih kombinacija jedinica i nula. Stoga će raspon brojeva koje možemo spremiti u 1 byte biti od 0 do 255.

U našem zapisu cijele brojeve spremamo u 4 bytea, tj. 32 bita, s tim da je prvi bit rezerviran za predznak. Koristeći isti način razmišljanja kao maloprije, imamo 2^{31} različitih kombinacija za spremanje pozitivnih brojeva te 2^{31} različitih kombinacija za spremanje negativnih brojeva u rasponu od 0 do $2^{31} - 1$.

Tehnika dvojnog komplementa. Primijetite da u gornjem zapisu imamo dvije nule:

$$\begin{aligned} -0 &= 1\,0000000\,|\,00000000\,|\,00000000\,|\,00000000 \\ 0 &= 0\,0000000\,|\,00000000\,|\,00000000\,|\,00000000 \end{aligned}$$

Kako bi izbjegla dvije nule, ali i pojednostavila aritmetičke operacije s cijelim brojevima (kao što će biti kasnije objašnjeno), većina modernih računala koristi tzv. **tehniku dvojnog komplementa** (*two's complement*) za zapisivanje negativnih brojeva. Pozitivni cijeli brojevi su i dalje reprezentirani u memoriji na gore objašnjen način. Međutim, ukoliko od pozitivnog broja želimo napraviti negativni, ili obrnuto, postupak prevođenja u zapisu dvojnog komplementa je sljedeći:

1. korak: sve se nule prebacuju u jedinice;
2. korak: sve se jedinice prebacuju u nule;
3. korak: i svemu se još doda jedinica.

Primjer 3.9 *Primjer broja 1 i -1 u zapisu dvojnog komplementa.*

$$\begin{aligned} 1 &= 0\,0000000\,|\,00000000\,|\,00000000\,|\,00000001 \\ -1 &= 1\,1111111\,|\,11111111\,|\,11111111\,|\,11111111 \end{aligned}$$

Primijetite kako 1 na mjestu prvog bita (MSB) i dalje predstavlja negativan broj.

U ovakvom zapisu nula i dalje ostaje 0 0000000|00000000|00000000|00000000. Primjetite da se nula ne da prevesti u -0.

$$\begin{array}{rl} & 1\,1111111|11111111|11111111|11111111 \\ 0\,0000000|00000000|00000000|00000000 \ \longrightarrow & + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 \\ & \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\ & \text{OVERFLOW} \end{array}$$

Problem koji se pojavio pri prevođenju nule u -0 je u tome što će se dobiveni broj nalaziti izvan intervala $\left[-2^{31}-1, +2^{31}-1\right]$. Tada kažemo da je došlo do *overflowa*.

Ono što je prije bila -0, kod dvojnog se komplementa koristi za najveći negativan broj, tj.

$$1\,0000000|00000000|00000000|00000000 = -2^{31}.$$

Stoga je raspon brojeva koje možemo spremiti u 32 bita koristeći ovakav zapis od -2^{31} do $2^{31}-1$.

Primjer 3.10 *Kako glasi reprezentacija broja -31 u zapisu dvojnog komplementa? Zbog jednostavnosti pretpostavite da je ovaj cijeli broj zapisan u 1 byte.*

Rješenje: Broj 31 možemo prikazati u memoriji računala kao 00011111. Slijedeći prva dva koraka, tj. $0 \to 1$ i $1 \to 0$, dobivamo broj 11100000, te na kraju dobivenom broju dodajemo 1, tj.

$$\begin{array}{rl} 11100000 & + \\ \qquad\quad 1 & \\ \hline 11100001 & \end{array}$$

Tražena reprezentacija broja -31 glasi 11100001. Primijetite da je predznak dobivenoga broja 1 što odgovara $-$.

Zadaci za provjeru znanja

Pretvaranja između brojevnih sustava Tablice za pretvaranje oktalnih i heksadecimalnih brojeva u binarne:

oktalno	**binarno**
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

heksadecimalno	**binarno**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Zadatak 3.1 Pretvorite sljedeće dekadske brojeve u binarne:

a) 24

b) 128

c) 36

d) 101

e) 1234

f) 11111

Zadatak 3.2 Pretvorite sljedeće binarne brojeve u dekadske:

a) $101_{(2)}$

b) $1110110_{(2)}$

c) $10010100_{(2)}$

d) $10010011_{(2)}$

e) $111111111_{(2)}$

f) $10010001010_{(2)}$

Zadatak 3.3 Pretvorite sljedeće brojeve u zadane brojevne sustave:

a) $1224_{(10)} =?_{(16)}$

b) $4E23_{(16)} =?_{(10)}$

c) $122_{(10)} =?_{(8)}$

d) $732_{(8)} =?_{(10)}$

e) $BABE_{(16)} =?_{(8)}$

f) $55124_{(8)} =?_{(16)}$

Zadatak 3.4 Pretvorite sljedeće dekadske brojeve u zadane baze:

a) 306 u baze $2, 5, 9$

b) 5021 u baze $3, 6, 11$

c) 31708 u baze $4, 7, 12$

Zadatak 3.5 Bez pretvaranja iz baze u bazu, izračunajte:

a) Koliko znamenki ima broj $1F5CCA3_{(16)}$ u bazi 2?

b) Koliko znamenki ima broj $734123933_{(10)}$ u bazi 2?

c) Koliko znamenki ima broj $991124344_{(10)}$ u bazi 5?

d) Koliko znamenki ima broj $(5^{1250})_{(10)}$ u bazi 16?

Zadatak 3.6 Kako se u računalu zapisuje broj:

a) $9_{(10)}$

b) $456_{(10)}$?

Zadatak 3.7 Kolika je vrijednost, u dekadskom sustavu, broju koji u tehnici dvojnog komplementa ima oblik 11100100? Pretpostavite da je ovaj cijeli broj zapisan u 1 byte te da je prvi od osam bitova MSB.

Zadatak 3.8 Tehnikom dvojnog komplementa u 8-bitnom zapisu prikažite brojeve:

a) $-12_{(10)}$

b) $-120_{(8)}$

c) $-4B_{(16)}$

3.2 Računske operacije u binarnom, oktalnom i heksadecimalnom sustavu

U ovom ćemo poglavlju napraviti kratak pregled zbrajanja i oduzimanja brojeva danih u raznim brojevnim sustavima (kako smo već napomenuli, računalo množenje i dijeljenje izvodi kao niz uzastopnih zbrajanja, odnosno, oduzimanja). Operacije zbrajanja i oduzimanja u binarnom, oktalnom i heksadecimalnom sustavu izvode se analogno, kao i u dekadskom, pri čemu vrijede i ista svojstva (komutativnost, asocijativnost, ...).

3.2.1 Zbrajanje i oduzimanje u binarnom sustavu

Zbrajanje u binarnom sustavu ima jednaka svojstva kao i u dekadskom, a osnovna razlika je u tome da rezultat zbrajanja prikazujemo pomoću dviju znamenki (0, 1).

Sljedeća tablica prikazuje nekoliko jednostavnih zbrojeva binarnih brojeva. Primijetite, ako brojevi koje zbrajamo imaju dvije ili više znamenaka, za zbroj $1+1$ pisati ćemo 0 i prenijeti (pamtiti) 1, jer smo "ispunili bazu" $(1+1=2)$ (vidi tablicu 3.3). Primjerice, u dekadskom sustavu to odgovara zbroju $2+8$.

$0+0$	$=$	0
$0+1$	$=$	1
$1+0$	$=$	1
$1+1$	$=$	10

Tablica 3.3: Tablica zbrajanja binarnih znamenki.

Primjer 3.11 *Izračunajte zbroj dvaju binarnih brojeva* 101110 *i* 110101.

$$
\begin{array}{r}
10111\mathbf{0} \quad + \\
\underline{11010\mathbf{1}} \\
\mathbf{1} \rightarrow 0 \text{ plus } 1 \text{ je } 1 \\
1011\mathbf{1}0 \quad + \\
\underline{1101\mathbf{0}1} \\
\mathbf{1\,1} \rightarrow 1 \text{ plus } 0 \text{ je } 1
\end{array}
$$

$$
\begin{array}{rl}
101\mathbf{1}10 & + \\
\underline{110\mathbf{1}01} & \\
\quad\mathbf{011} & \rightarrow 1 \text{ plus } 1 \text{ je } 0 \text{ i } 1 \text{ “prenosimo”} \\
101\mathbf{1}10 & + \\
\underline{11\mathbf{0}101} & \\
\quad\mathbf{0011} & \rightarrow 1 \text{ plus } 0 \text{ je } 1 \text{ i plus } 1 \text{ (koji smo prenijeli) je } 0; \\
 & \text{a } 1 \text{ “prenosimo”} \\
10\mathbf{1}110 & + \\
\underline{11\mathbf{0}101} & \\
\quad\mathbf{00011} & \rightarrow 0 \text{ plus } 1 \text{ je } 1 \text{ i plus } 1 \text{ (koji smo prenijeli) je } 0; \\
 & \text{a } 1 \text{ “prenosimo”} \\
1\mathbf{0}1110 & + \\
\underline{1\mathbf{1}0101} & \\
\mathbf{100011} & \rightarrow 1 \text{ plus } 1 \text{ je } 0 \text{ i plus } 1 \text{ (koji smo prenijeli) je } 1; \\
 & \text{i } 1 \text{ “prenosimo”}
\end{array}
$$

Primijetite da smo u posljednjem redu trebali prenijeti jednu jedinicu, ali to smo zbog jednostavnosti uradili u istom redu. Tako je dodatna jedinica stavljena ispred do tada dobivenoga rezultata: **1**100011.

Oduzimanje se također izvodi analogno kao i u dekadskom sustavu, samo što u slučaju kada oduzimamo veću znamenku od manje umjesto “posuđivanja” desetke, “posuđujemo” 2.

Primjer 3.12 *Izračunajte razliku binarnih brojeva* 1001 *i* 11.

$$
\begin{array}{rl}
100\mathbf{1} & - \\
\underline{\quad 1\mathbf{1}} & \\
\quad\mathbf{0} & \rightarrow 1 \text{ minus } 1 \text{ je } 0 \\
10\mathbf{0}1 & - \\
\underline{\quad\mathbf{1}1} & \\
\quad\mathbf{10} & \rightarrow 0 \text{ je manja od } 1 \text{ pa moramo “posuditi” } 2, \text{ tako da je } 0 \text{ plus } 2 \\
 & \text{jednako } 2 \text{ pa } 2 \text{ manje } 1 \text{ jednako } 1 \text{ i pišemo } 1
\end{array}
$$

1
1**0**01 −
11
1 1 0 → 0 koju promatramo postaje 1 jer smo u gornjem redu posudili 2 i oduzeli 1 pa je ukupno ostalo 1, tako da je 1 minus 0 jednako 1 i pišemo 1. I dalje imamo posuđeno 1

0
1001 −
11
0 1 1 0 → posljednja znamenka 1 se poništava jer smo ju već ranije "posudili", pa pišemo 0

3.2.2 Aritmetičke operacije u zapisu dvojnog komplementa

Zakon zbrajanja: brojeve zbrajamo tako da ih tretiramo kao brojeve bez predznaka, tj. najznačajniji bit (lijevi bit), odnosno, bit s predznakom smatramo dijelom broja. Ako nakon zbrajanja ostane dio koji "bismo trebali prenijeti" nakon lijevog bita, to zanemarujemo.

Primjer 3.13 *Zbrojite brojeve* 11000111 *i* 11011101 *dane u zapisu dvojnog komplementa.*

11000111 +
11011101

0 — 1 i 1 su 10 pa "pišemo" 0 i 1 "pamtimo" (prenosimo)

11000111 +
11011101

00 — 1 i 0 su 1 plus ono što smo zapamtili je 10, pa pišemo 0 i 1 prenosimo

11000111 +
11011101

100 — 1 i 1 su 10 plus 1 koji smo prenijeli je 11, pa pišemo 1 i 1 prenosimo

1100**0**111	+	
1101**1**101		
0100		0 i 1 su 1 plus ono što pamtimo je 10, pa pišemo 0 i 1 prenosimo
110**0**0111	+	
110**1**1101		
00100		0 i 1 su 0 plus ono što pamtimo je 10, pa pišemo 0 i 1 prenosimo
11**0**00111	+	
11**0**11101		
100100		0 i 0 su 0 plus ono što pamtimo je 1, pa pišemo 1 i ništa ne prenosimo
11**1**000111	+	
11**1**011101		
0100100		1 i 1 su 10 pa pišemo 0 i 1 prenosimo
11000111	+	
11011101		
10100100		1 i 1 su 10 plus ono što smo prenijeli je 11, pa pišemo 1 i zanemarujemo ono što treba prenijeti

Napomena 3.1 *Dekadski zapis dobivenoga broja iznosi* 92*, što je upravo vrijednost zbroja brojeva* $11000111_{(2)} = 57_{(10)}$ *i* $11011101_{(2)} = 35_{(10)}$*.*

Detekcija overflowa. Posljednji bit kojega prenosimo u računskoj operaciji zbrajanja u sebi sadrži informaciju o *overflowu* u računanju, tj. broju prevelikom da bi se reprezentirao u zadanom broju bitova (u našem se primjeru radi o osam bitova). Ako su posljednja dva bita koja prenosimo oba 1 ili oba 0, s rezultatom je sve u redu. Međutim, ako su posljednja dva bita koja prenosimo oblika 1 i 0 ili 0 i 1, dogodio se *overflow.*

Primjer 3.14 *Odredite rezultat zbrajanja broja* $0111_{(2)} = 7_{(10)}$ *i* $0011_{(2)} = 3_{(10)}$*, ako su brojevi prikazani tehnikom dvojnog komplementa u zapisu duljine 4 bita.*

011**1**	+	
001**1**		
0		1 prenosimo

01**1**1	+	
00**1**1		
10		1 prenosimo
0**1**11	+	
0**0**11		
010		1 prenosimo
0111	+	
0011		
1010		0 prenosimo

Napomena 3.2 $1010_{(2)} = -6$, *pa smo kao rezultat zbrajanja dobili* $7+3 = -6$. *Ovakvu grešku, međutim, možemo lako uočiti budući su posljednja dva bita koja smo prenosili bili oblika 1 i 0.*

Kod oduzimanja se *overflow* detektira na potpuno isti način, tj. ako su posljednja dva bita koja se prenose različita.

Zakon oduzimanja: brojeve oduzimamo tako što ih također smatramo brojevima bez predznaka, tj. najznačajniji bit (lijevi bit), odnosno bit s predznakom smatramo dijelom broja. Ako nam za oduzimanje treba 1 iza lijevog bita, to "posudimo" i nakon operacije oduzimanja tu jedinicu zanemarujemo.

Primjer 3.15 *Odredite rezultat oduzimanja broja* 11011101 *od broja* 00111001, *ako su brojevi dani u zapisu dvojnog komplementa.*

0011100**1**	−	
1101110**1**		
0		1 minus 1 su 0 pa "pišemo" 0
0011100**1**	−	
1101110**1**		
00		0 minus 0 su 0 pa "pišemo" 0
0011**0**01	−	
1101**1**01		
100		0 je manje od 1, pa posuđujemo 2 od gornjeg broja. 2 minus 1 su 1 pa "pišemo" 1

00111001	–	
11011101		
11100		**1** je dobiven tako da smo uzimajući u obzir posuđenu jedinicu, umjesto 1 dobili 0, a kako je 0 manja od 1 moramo ponovno posuditi 2 u gornjem broju. Tako imamo 2 minus 1 je 1 i pišemo 1
00**1**11001	–	
11**0**11101		
11100		ta nova jedinica dobivena je potpuno analogno kao u gornjem slučaju, pa i dalje imamo posuđeno 2
00**1**11001	–	
11**0**11101		
011100		budući da smo posudili 2, gornja jedinica postaje 0 pa imamo 0 minus 0 je 0 i pišemo 0.
0**0**111001	–	
1**1**011101		
1011100		0 je manje od 1 pa posuđujemo 2 od gornjeg broja. 2 minus 1 su 1 pa "pišemo" 1
00111001	–	
11011101		
01011100		0 postaje 1, pa kako je 1 minus 1 jednako 0 "pišemo" 0.

3.2.3 Zbrajanje i oduzimanje u heksadecimalnom sustavu

Kako se vidi u prethodnom poglavlju, kod računanja u binarnom sustavu, zbog velikog broja nula i jedinica, lako može doći do zabune. Zato je jednostavnije takve računske operacije izvoditi u heksadecimalnom sustavu, a onda ih pretvoriti u binarni (ako je potreban binarni oblik rezultata). Operacije se u heksadecimalnom sustavu izvode potpuno analogno kao i u dekadskom, samo treba imati na umu da smo u sustavu s bazom 16, pa kada "posuđujemo jedan", to znači da smo posudili 16 (a ne 10 kao u dekadskom sustavu).

Primjer 3.16 *Izračunajte zbroj brojeva u heksadecimalnoj bazi* 1A23 *i* 7C28.

1 A 2 **3** +
7 C 2 **8**

B → 3 plus 8 je 11, a znamenka za 11 je B

1 A **2** 3 +
7 C **2** 8
4 B → 2 plus 2 je 4, a oznaka za 4 je 4
1 **A** 2 3 +
7 **C** 2 8
6 4 B → računajući u dekadskom sustavu: $A = 10$ plus $C = 12$ što
daje 22, a 22 je u heksadecimalnom zapisu 16 (1·16 + 6),
zato pišemo 6 i jedan "prenosimo"
1 A 2 3 +
7 C 2 8
9 6 4 B → 1 + 7 su 8 i plus 1 koji smo prenijeli je 9
zato pišemo 9.

Primjer 3.17 *Izračunajte razliku brojeva danih u heksadecimalnoj bazi* 1F00A *i* B2.

1 F 0 0 **A** −
B **2**
8 → 10 minus 2 je 8, a znamenka za 8 je 8
1 F 0 **0** A −
B 2
5 8 → 0 je manja od 11, zato moramo posuditi 16
pa imamo 16 minus 11 je 5, zato pišemo 5
F
1 F **0** 0 A −
B 2
F 5 8 → Promatrana 0 postaje F, jer smo bazu (16) umanjili za 1
koji smo posudili, a 15 minus 0 je 15, zato pišemo F
i opet smo posudili 1 (nismo mogli oduzeti 1 od 0)
i dalje 1 ostaje posuđen

E
1**F**00A –
B2
EF,58 → F minus 1 (koji smo prenijeli) je E, (15 minus 1 je 14),
zato pišemo E
1F00A –
B2
1EF58 → 1 minus 0 je 1, pa pišemo 1

Zadaci za provjeru znanja

Zadatak 3.9 Odredite decimalnu vrijednost razlike brojeva 11110000 i 00010111 ako su dani konvencijom predznaka i ako je osnovna jedinica byte.

Zadatak 3.10 Što je dvojni komplement broja $-65_{(10)}$?

Zadatak 3.11 Odredite dvojni komplement brojeva $-35_{(10)}$ i $-56_{(10)}$.

Zadatak 3.12 Odredite pozitivnu decimalnu vrijednost dvojnog komplementa broja $11100100_{(2)}$.

Zadatak 3.13 Odredite decimalni zapis binarnog broja $10001100_{(2)}$ ako ga promatramo kao:

a) binarni broj bez predznaka,

b) dvojni komplement.

Zadatak 3.14 Odredite decimalnu vrijednost rezultata zbrajanja brojeva

a) 11000111 i 11011101;

b) 1100111 i 1110001,

ako su oba broja zapisana u obliku dvojnog komplementa.

Zadatak 3.15 Odredite rezultat oduzimanja brojeva zapisanih u obliku dvojnog komplementa:

a) 11011101 - 00111001;

b) 00111001 - 11011101,

ako su oba broja zapisana u obliku dvojnog komplementa.

Zadatak 3.16 Odredite rezultat zbrajanja brojeva 10011011 i 11100011 pri čemu su oba broja zapisana u obliku dvojnog komplementa. Je li došlo do *overflowa*? Zašto?

Zadatak 3.17 Odredite razliku brojeva 10001011 i 10101 pri čemu su oba broja zapisana u obliku dvojnog komplementa. Je li došlo do *overflowa*? Zašto?

Zadatak 3.18 Koliko je bitova potrebno kako bi se predstavilo 10 dekadskih znamenki? A 17?

Zadatak 3.19 Izračunajte:

a) $101111_{(2)} + 1010001_{(2)} = ?_{(2)}$

b) $53601_{(8)} + 73266_{(8)} = ?_{(8)}$

c) $7A7A_{(16)} + 7E7A_{(16)} = ?_{(16)}$

d) $40012_{(5)} + 33213_{(5)} = ?_{(5)}$

Zadatak 3.20 Izračunajte:

a) $101101_{(2)} - 10101_{(2)} = ?_{(2)}$

b) $53601_{(8)} - 7266_{(8)} = ?_{(8)}$

c) $BA72A_{(16)} - C14A_{(16)} = ?_{(16)}$

d) $32010_{(6)} - 21513_{(6)} = ?_{(6)}$

Zadatak 3.21 Izračunajte:

a) $11101_{(2)} \cdot 101_{(2)} = ?_{(2)}$

b) $1011011_{(2)} \cdot 1011_{(2)} = ?_{(2)}$

Zadatak 3.22 Izračunajte razlike sljedećih dekadskih brojeva u 8-bitnom zapisu koristeći tehniku dvojnog komplementa: $7 - 5$, $50 - 61$, $120 - 128$.

Zadatak 3.23 Odredite binarne ekvivalente sljedećih dekadskih brojeva:

$$43, 77, 115, 331, 1010.$$

Provjerite rezultat ponovnom pretvorbom u dekadski sustav.

Zadatak 3.24 Odredite dekadske ekvivalente sljedećih binarnih brojeva:

$$1001, 101001, 11001101, 1110011, 10010110.$$

Provjerite rezultat ponovnom pretvorbom u binarni sustav.

Zadatak 3.25 Koja je vrijednost svake znamenke binarnog broja $11010001_{(2)}$? Kako glasi decimalni zapis toga broja?

Zadatak 3.26 Odredite oktalni zapis sljedećih dekadskih brojeva:

$$126, 377, 485, 641.$$

Provjerite rezultat ponovnom pretvorbom u dekadski sustav.

Zadatak 3.27 Odredite dekadski ekvivalent oktalnih brojeva 147 i 555.

Zadatak 3.28 Pretvorite brojeve 5049 i 260137 iz dekadskog zapisa u heksadecimalni.

Zadatak 3.29 Brojeve 1AC7 i DCEF9 prebacite iz heksadecimalnog u dekadski brojevni sustav.

Zadatak 3.30 Načinite sljedeće pretvorbe:

a) $711_{(8)} = ?_{(16)}$

b) $AC1_{(16)} = ?_{(8)}$

c) $10101011_{(2)} = ?_{(8)} \qquad (= ?_{(16)})$

d) $ABC23_{(16)} = ?_{(2)} \qquad (= ?_{(8)})$

Zadatak 3.31 Izvedite sljedeće aritmetičke operacije:

a) $11011101_{(2)} + 10101001_{(2)}$

b) $11100010_{(2)} - 10000111_{(2)}$

c) $267E2_{(16)} + 4C10B_{(16)}$

d) $1F00A_{(16)} - B21_{(16)}$

e) $17_{(8)} + 376_{(8)}$

f) $255_{(8)} - 66_{(8)}$.

Zadatak 3.32 Izračunajte sljedeću razliku:

$$12A3E_{(16)} - 234_{(8)} - 11001101_{(2)}.$$

Rezultat izrazite u binarnom i dekadskom brojevnom sustavu.

3.3 Racionalni brojevi

Svaki racionalni broj iz skupa $\mathbb{Q} = \{\frac{m}{n} : m \in \mathbb{Z}, n \in \mathbb{N}\}$ možemo zapisati u decimalnom zapisu i raspisati po dekadskoj bazi:

$$\begin{aligned} R &= d_{p-1}d_{p-2}\ldots d_1d_0.d_{-1}d_{-2}\ldots d_{-q(10)} \\ &= d_{p-1}10^{p-1} + \ldots + d_1 10^1 + d_0 10^0 + d_{-1}10^{-1} + d_{-2}10^{-2} + \ldots + d_{-q}10^{-q} \end{aligned}$$

Na sličan način racionalne brojeve možemo zapisati u binarnom zapisu s točkom:

$$\begin{aligned} R &= b_{p-1}b_{p-2}\ldots b_1b_0.b_{-1}b_{-2}\ldots b_{-q(2)} \\ &= b_{p-1}2^{p-1} + b_{p-2}2^{p-2} + \ldots + b_1 2^1 + b_0 + b_{-1}2^{-1} + b_{-2}2^{-2} + \ldots + b_{-q}2^{-q} \end{aligned} \tag{3.1}$$

Pretvaranje racionalnih brojeva u binarne. Osnovna ideja pretvaranja racionalnih brojeva u binarni oblik je napisati racionalni broj u decimalnom obliku $d_{p-1}d_{p-2}\ldots d_1d_0.d_{-1}d_{-2}\ldots d_{-q}$, te zasebno pretvarati broj $d_{p-1}d_{p-2}\ldots d_1d_0$ i broj $0.d_{-1}d_{-2}\ldots d_{-q}$ u binarne brojeve. Pri pretvaranju broja $d_{p-1}d_{p-2}\ldots d_1d_0$ koristimo ideju cjelobrojnog dijeljenja s 2 i pamćenja ostatka, kao što smo već objasnili, dok kod pretvaranja broja $0.d_{-1}d_{-2}\ldots d_{-q}$ koristimo ideju množenja s brojem 2 i pamćenja cijela. Promotrimo sljedeći primjer:

Primjer 3.18 *Napišite decimalne brojeve 3.75 i 0.47 u binarnom obliku.*

a) $3.75 = 3 + 0.75$. *Dekadski broj 3 je oblika* 11_2 *u binarnom obliku, dok broj* 0.75 *možemo pretvoriti u binarni oblik množenjem s 2 i pamćenjem cijela.*

$$\begin{array}{rcrcl|l} 0.75 & \cdot & 2 & = & \underline{1}.5 & 1 \\ 0.5 & \cdot & 2 & = & \underline{1}.0 & 1 \\ 0 & \cdot & 2 & = & 0 & \end{array}$$

Dakle, $3.75 = 11.11_2$.

b)

$$\begin{array}{rcrcl|l} 0.47 & \cdot & 2 & = & 0.94 & 0 \\ 0.94 & \cdot & 2 & = & 1.88 & 1 \\ 0.88 & \cdot & 2 & = & 1.76 & 1 \\ 0.76 & \cdot & 2 & = & 1.52 & 1 \\ 0.52 & \cdot & 2 & = & 1.04 & 1 \\ 0.04 & \cdot & 2 & = & 0.08 & 0 \\ & & & \vdots & & \vdots \end{array}$$

Dakle, $0.47 = 0.11110\ldots$

Pokušajmo opravdati valjanost gornjeg postupka pretvaranja. U tu svrhu raspišimo neki binarni broj s točkom $R_{(2)} = 0.b_{-1}b_{-2}\ldots b_{-q\,(2)}$ u raspisu po dekadskoj bazi:

$$R_{(10)} = b_{-1}2^{-1} + b_{-2}2^{-2} + \ldots + b_{-q}2^{-q}.$$

Ako gornji raspis pomnožimo s dva dobit ćemo $2 \cdot R_{(10)} = b_{-1}.b_{-2}\ldots b_{-q\,(2)}$. Binarna znamenka b_{-1} će se pojaviti lijevo od točke, a postupak iterativno nastavimo na broju $0.b_{-2}\ldots b_{-q}$ i tako ukupno q puta dok ne dobijemo svih q binarnih znamenaka.

Primijetite da će se broj 0.47 iz primjera 3.18 pretvoriti u beskonačan niz znamenki nula i jedinica u binarnom zapisu. Ono što možda na prvi pogled zbunjuje je mogućnost da jednostavan broj poput broja 0.47 postane beskonačan broj nula i jedinica. Odgovor leži u binarnom zapisu racionalnog broja (3.1).

Ako pomnožimo i podijelimo zapis racionalnog broja (3.1) s 2^q dobijemo racionalni broj R oblika:

$$R = \frac{b_{p-1}2^{p-1+q} + b_{p-2}2^{p-2+q} + \ldots + b_1 2^{1+q} + b_0 2^q + b_{-1}2^{q-1} + \ldots + b_{-q}2^0}{2^q}$$

Dakle, svaki binarni broj s konačno mnogo znamenaka, tj. $p+q$ znamenaka, oblika

$$b_{p-1}b_{p-2}\ldots b_1 b_0 . b_{-1}b_{-2}\ldots b_{-q}$$

egzaktno reprezentira neki racionalni dekadski broj čiji se nazivnik može napisati **kao potencija broja 2**.

Vratimo se na primjer 3.18. Primijetite, nazivnik broja 47/100 ne može se napisati kao potencija broja dva, za razliku od nazivnika broja 75/100.

3.3.1 Prikaz racionalnih brojeva u računalu

Osnovna ideja prikazivanja racionalnih brojeva u računalu je prebaciti racionalni broj u binarni zapis s točkom, na gore opisani način, i broj *normirati*. Za neki racionalni broj z zapisan u bazi b kažemo da je normiran ako je zapisan u obliku

$$z_{(b)} = z_0 . z_1 z_2 \ldots z_{t-1} \cdot b^E$$

Drugim riječima, broj normiramo tako da decimalnu točku postavimo iza prve znamenke različite od nula, a sam broj pomnožimo s odgovarajućom bazom na eksponent E koji odgovara upravo broju pomaka decimalne točke.

Primjer 3.19 *Prebacite sljedeće brojeve najprije u binarni zapis, a zatim ih normirajte:*

a) $3.75 = 11.11_{(2)} = 1.111 \cdot 2^1$

b) $11.3125 = 1011.0101_{(2)} = 1.0110101 \cdot 2^3$

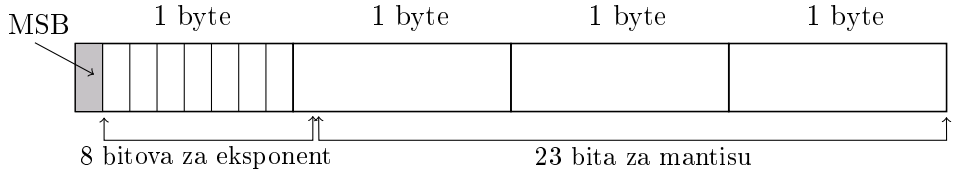
c) $0.125 = 0.001_{(2)} = 1.0 \cdot 2^{-3}$

Općenito, svaki će racionalni broj u računalu biti oblika

$$\pm z_0 . \underbrace{z_1 z_2 \ldots z_{t-1}}_{\text{mantisa}} \cdot 2^E,$$

gdje je eksponent E cijeli broj dobiven nakon postupka normiranja. Primijetite da znamenku z_0 nismo naveli pod mantisu budući je ona poznata i načelno će uvijek biti 1[2]. Takav se prikaz broja zove **prikaz broja s pomičnim zarezom** (eng. *floating point*).

Racionalne brojeve u računalu uglavnom spremamo u 4 bytea (tip *float*).



Slika 3.2: Racionalni broj spremljen u 4 bytea. Prvi bit (MSB bit ili 'most significant bit') rezerviran je za predznak broja, sljedećih osam bitova za eksponent, a preostalih 23 za mantisu.

Primijetite kako je samo jedan bit rezerviran za predznak, i to predznak broja. Međutim, što ako je sam eksponent E negativan broj? Problem je riješen tako da se u 8 bita sprema vrijednost od koje se oduzima -127 te se dobije prava vrijednost eksponenta E, s tim da se posebno razmatra slučaj kada su svi bitovi eksponenta E jednaki 1, i slučaj kada su svi bitovi jednaki 0. Tako je

[2] Znamenka z_0 u računalu neće biti 1 samo u nekim specijalnim slučajevima o kojima ovdje nećemo govoriti. Za razumijevanje prikaza racionalnih brojeva u računalu dovoljno je uvijek razmišljati o znamenci z_0 kao o jedinici.

izbjegnuto rezerviranje još jednog bita za predznak samog eksponenta. Budući je 8 bitova rezervirano za eksponent imamo $2^8 = 256$ različitih eksponenata u rasponu od -126 do 127, plus dvije rezervirane vrijednosti:

a) vrijednosti svih bitova eksponenta su 0: koristi se za reprezentaciju broja 0, ali i tzv. subnormalnih brojeva (o kojima ovdje nećemo detaljnije govoriti);

b) vrijednosti svih bitova eksponenta su 1: koristi se za reprezentaciju nekih specijalnih vrijednosti poput beskonačnosti i NaN ("not a number").

23 bita rezervirana za spremanje mantise zapravo dopušta 2^{24} različitih brojeva u rasponu od 0 do $2^{24} - 1$ budući da je broj normiran, a znamenka z_0 je uvijek jedan.

Stoga je najveći racionalni broj koji se na ovakav način može spremiti u računalo reda veličine

$$(1 + 2^{-1} + 2^{-2} + \ldots + 2^{-23}) \cdot 2^{127} \approx 3.4028 \cdot 10^{38}.$$

Analogno tome, najmanji pozitivni racionalni broj možemo izraziti kao $1 \cdot 2^{-126} \approx 1.1755 \cdot 10^{-38}$.

Iracionalni brojevi. Naučili smo da se racionalni brojevi čija binarna reprezentacija nije konačna (nazivnik se ne može prikazati kao potencija broja 2) ne mogu egzaktno spremiti u binarnom zapisu tipa *float*. Međutim, važno je primjetiti kako to još uvijek ne znači da takve brojeve ne možemo spremiti egzaktno i bez aproksimacije u računalo. Na primjer, takve brojeve uvijek egzaktno možemo spremiti kao dva cijela broja, nazivnik i brojnik.

Na žalost, ipak postoje i brojevi koji se ne mogu egzaktno spremiti u računalu pa ih se mora aproksimirati. Takve brojeve nazivamo iracionalnim brojevima, onima koji se ne mogu napisati u obliku razlomka. Primjeri iracionalnih brojeva su:

$$\sqrt{2}, \sqrt{3}, \pi, e, \ldots$$

Zadaci za provjeru znanja

Zadatak 3.33 Pretvorite sljedeće binarne brojeve u dekadske:

a) $1100.1101_{(2)}$

b) $-10.001_{(2)}$

c) $1011.1111_{(2)}$

d) $10011.010101_{(2)}$

e) $111.111_{(2)}$

f) $-100011.000001_{(2)}$

Zadatak 3.34 Pretvorite sljedeće dekadske brojeve u binarne. Ako postupak nije moguć, objasnite zašto.

a) $13.625_{(10)}$

b) $0.515625_{(10)}$

c) -20.171875

d) $65.225_{(10)}$

e) $-100.85_{(10)}$

Zadatak 3.35 Normirajte sljedeće brojeve u bazi u kojoj su zapisani:

a) $813_{(10)}$

b) $0.00027135_{(10)}$

c) $1001.01_{(2)}$

d) $0.0100101_{(2)}$

e) $1AB45.F_{(16)}$

Zadatak 3.36 U računalnoj memoriji upisan je broj u obliku *float*. U tom obliku računalo pohranjuje racionalne brojeve koristeći četiri byte-a (32 bita) tako da prvi bit označava predznak, idućih osam označava eksponent s pomakom -127, a preostalih 23 bita predstavlja mantisu. Napišite taj broj u dekadskom obliku.

a) $C2\ B0\ 00\ 00_{(16)}$

b) $43\ 00\ 20\ 00_{(16)}$

c) $C1\ 4A\ 00\ 00_{(16)}$

d) $82\ 0C\ 00\ 01_{(16)}$

Zadatak 3.37 Kako se u računalu koristeći tip *float* pohranjuju sljedeći racionalni brojevi:

a) $\frac{1}{2}_{(10)}$

b) $-\frac{3}{8}_{(10)}$

c) $\frac{50000}{16}_{(10)}$

d) $-\frac{3}{1024}_{(10)}$

Zadatak 3.38 Pretpostavite da računalo koristi tip *mojfloat* koji se sastoji od:

- 1 bita za predznak,
- 7 bitova za eksponent i
- 8 bitova za mantisu.

Ako mantisu nije moguće prikazati sa zadanim brojem binarnih znamenaka, zadnja znamenka se zaokružuje.

Promotrite rješavanje kvadratne jednadžbe $ax^2 + bx + c = 0$ koristeći varijable tipa *mojfloat*, ako su zadani koeficijenti $a = 1$, $b = 1000$, $c = 0.001$.

a) Zašto je došlo do nepreciznosti kod jednog od rješenja?

b) Kolike su relativna i apsolutna pogreška u rezultatu?

c) Može li se problem drugačije postaviti kako bi se smanjila pogreška?

3.3.2 Pogreške u računanju u aritmetici s pomičnim zarezom

Strojni brojevi najčešće predstavljaju aproksimaciju nekog realnog broja. Sljedeći rezultat govori o relativnoj pogrešci aproksimacije realnog broja strojnim brojem (eng. *floating point arithmetic*).

Neka je $\widetilde{x}$ strojna reprezentacija broja x u bazi b. Tada vrijedi

$$\frac{|\widetilde{x} - x|}{x} \leq \mu \leq b^{1-t},$$

gdje je t broj znamenaka u strojnoj reprezentaciji.

Broj μ zovemo strojna preciznost i obično iznosi $\sim 10^{-16}$ u dvostrukoj, odnosno $\sim 10^{-8}$ u jednostrukoj točnosti.

Na primjer, ako strojne brojeve prikazujemo u 32-bitnom prikazu (kao u prošlom poglavlju), tada je strojna točnost

$$\mu = 2^{-23} \sim 1.2 \cdot 10^{-7},$$

a ako koristimo 64-bitni zapis, tada je strojna točnost

$$\mu = 2^{-52} \sim 2.2 \cdot 10^{-16}.$$

3.3.2.1 Apsolutna i relativna pogreška aproksimacije

Ako je neki realni broj a aproksimiran brojem $\widetilde{a}$, onda se apsolutna vrijednost razlike $\triangle a = \widetilde{a} - a$ ili $|\widetilde{a} - a|$ naziva apsolutna pogreška aproksimacije. Kada je $a \neq 0$, definirana je i tzv. relativna pogreška aproksimacije s:

$$\delta a = \frac{|\triangle a|}{a} = \frac{|\widetilde{a} - a|}{a}.$$

Pišemo

$$\tilde{a} = a + \triangle a = a\left(1 + \frac{|\triangle a|}{a}\right).$$

Iz gornje jednakosti vidimo da će nas zanimati koliko najviše mogu biti veličine

$$\triangle a \quad \text{i} \quad \frac{|\triangle a|}{a}.$$

3.3.2.2 Zakoni strojne aritmetike

Treba imati na umu kako su u računalu pohranjene približne vrijednosti realnih brojeva, tako se i računske operacije izvode s određenom točnošću, odnosno pogreškom.

Oznake $\oplus, \odot, \ominus, \oslash$ označavat će strojne operacije zbrajanja, množenja, oduzimanja i dijeljenja.

Pitamo se, kolika je relativna razlika između brojeva $x+y$ i $x \oplus y$?

Označimo li ζ najbliži manji i ξ najbliži veći strojni broj broju $x+y$, tada vrijedi

$$\frac{|(x+y)-(x\oplus y)|}{|x+y|} \leq \frac{1/2|\xi-\zeta|}{\zeta} \leq \frac{1}{2}\, r$$

gdje je

$$r = \frac{|\xi-\zeta|}{\zeta}.$$

Dakle $e = \frac{1}{2}\, r$ najveća je relativna pogreška strojnog zbrajanja $\oplus$.

Možemo, dakle, pisati da vrijedi

$$x \oplus y = (x+y)\cdot(1+\varepsilon_{x+y})\,, \qquad |\varepsilon_{x+y}| \leq e\,.$$

Nas u praksi ne zanima stvarna vrijednost broja ε_{x+y}. Ono što želimo jest osigurati da uvijek vrijedi

$$|\varepsilon_{x+y}| \leq e = 2^{-24} \sim 10^{-8} \quad \text{(u jednostrukoj točnosti)}.$$

Napomena:

1. Za relativnu grešku ε_{x+y} zbrajanja brojeva $x, y \in \mathcal{F}$ vrijedi

$$\frac{|(x+y)-(x\oplus y)|}{|x+y|} \leq \varepsilon_{x+y}.$$

2. Neka δ_i označava relativnu grešku aproksimacije računalnim brojem nekog broja. Tada vrijedi $x = x(1+\delta_1)$ i $y = y(1+\delta_2)$, a zbrajanje brojeva x, y možemo pisati

$$(x \oplus y) = x(1+\delta_1) + y(1+\delta_2) = (x+y)(1+(\delta_1+\delta_2))\,,$$

odnosno, za relativnu grešku ε_{x+y} zbrajanja brojeva $x, y \in \mathcal{F}$ vrijedi

$$\frac{|(x+y)-(x\oplus y)|}{|x+y|} \leq \delta_1 + \delta_2 \leq \varepsilon_{x+y}.$$

To znači da je relativna greška ε_{x+y} zbrajanja brojeva $x, y \in \mathcal{F}$ uvijek veća od zbroja pogrešaka aproksimacija strojnih brojeva.

Isti zaključci vrijede i za ostale strojne aritmetičke operacije:

$$\begin{aligned}
x \ominus y &= (x-y)\cdot(1-\varepsilon_{x-y}), & |\varepsilon_{x-y}| \leq e \\
x \odot y &= (x\cdot y)\cdot(1-\varepsilon_{x.y}), & |\varepsilon_{x\cdot y}| \leq e \\
x \oslash y &= \frac{x}{y}\cdot(1-\varepsilon_{x/y}), & |\varepsilon_{x/y}| \leq e \quad y \neq 0 \\
\mathrm{SQRT}(x) &= \sqrt{x}\cdot(1-\varepsilon_{\sqrt{x}}), & |\varepsilon_{\sqrt{x}}| \leq e \quad y \geq 0
\end{aligned}$$

(za $\sqrt{x}$ se često koriste posebni algoritmi)

Napomena: za svaki $x \in \mathcal{F}$ funkcija $\mathrm{ABS}(x) = x$ se izračunava **egzaktno** jer je to jednostavna operacija promjene bita predznaka.

Važno je biti svjestan da za strojne operacije općenito ne vrijede neki od standardnih računskih zakona poput distributivnosti ili asocijativnosti. Točnije:

$$\begin{aligned}
(x\oplus y)\oplus z &\neq x\oplus(y\oplus z) \\
(x\odot y)\odot z &\neq x\odot(y\odot z) \\
x\odot(y\oplus z) &\neq x\odot y\oplus x\odot z \\
x\odot(1\ominus x) &\neq 1
\end{aligned}$$

Primjer 3.20 *Promotrimo računanje funkcije $f(x) = e^x - x - 1$ u točki $x = 0.01$, na stroju koji radi s pet značajnih znamenaka.*

Izračunajmo najprije

$$a = e^x - 1 = 1.0101 - 1 = 0.0101,$$

sada vidimo da je vrijednost funkcije $f(0.01) =$ **0.0001**

Međutim, točan rezultat u prikazu s pet značajnih znamenaka glasi $f(0.01) =$ **0.000050167**.

Što se dogodilo? Otkud tolika razlika?

Možemo li izbjeći oduzimanje? (Pretpostavljamo kako smo tu pogriješili ako su ostali računi provedeni točno.)

Oduzimanja se često, na sreću, mogu izbjeći. Primjerice, prikažimo e^x kao red potencija:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

sada je očito

$$f(x) = \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots$$

ili u prikazu s 5 značajnih znamenaka

$$f(0.01) = \frac{0.0001}{2!} + \frac{0.000001}{3!} + \frac{10^{-8}}{4!} = 0.000050167\,.$$

Primijetite kako pribrojnici poslije $x^4/(4!)$ ne doprinose točnosti rješenja.

Pojava opisana u prethodnom primjeru spada u tzv. ***katastrofalno kraćenje*** (**Catastrophic Cancelletion**).

Do ove pojave dolazi kada oduzimamo bliske brojeve. Nažalost, kod takvog oduzimanja često može doći do velikih grešaka i gubljenja točnosti, tj. možemo dobiti rezultat koji nema niti jednu točnu značajnu znamenku.

Primjer 3.21 *Promotrimo jednostavan primjer rješavanja kvadratne jednadžbe*

$$ax^2 + bx + c = 0\,.$$

U srednjoj smo školi naučili da su rješenja dana sa

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 &= \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \end{aligned}$$

*Neka je $b^2 \gg 4ac$. Tada je $\sqrt{b^2 - 4ac} \approx |b|$ pa pri računanju $-b \pm |b|$ u jednom slučaju nastupa **katastrofalno kraćenje**.*

Što uraditi?

*Treba **izbjeći** katastrofalno kraćenje po svaku cijenu.*

Zapišimo x_1 i x_2 malo drugačije:

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b - \sqrt{b^2 - 4ac})} = \frac{2c}{-b - \sqrt{b^2 - 4ac}}. \end{aligned}$$

Analogno, za x_2 dobivamo:

$$\begin{aligned} x_2 &= \frac{-b-\sqrt{b^2-4ac}}{2a} \cdot \frac{-b+\sqrt{b^2-4ac}}{-b+\sqrt{b^2-4ac}} \\ &= \frac{b^2-(b^2-4ac)}{2a(-b+\sqrt{b^2-4ac})} = \frac{2c}{-b+\sqrt{b^2-4ac}} \end{aligned}$$

Neka je sada $b^2 \gg 4ac$ (bez smanjenja općenitosti pretpostavimo da je $b > 0$). Iz gornjih jednakosti slijedi:

$$\begin{aligned} x_1 &= \frac{-2c}{b+\sqrt{b^2-4ac}} \approx \frac{-2c}{2b} \\ x_2 &= \frac{-b-\sqrt{b^2-4ac}}{2a} \approx \frac{-b}{a}. \end{aligned}$$

Kako bi formule za x_1 i x_2 trebale izgledati ako je $b < 0$?

Primjer 3.22 *Neka je zadana funkcija $f : \mathbb{Q}\backslash\{0\} \to \mathbb{Q}$ definirana kao*

$$f(x) = \frac{1-\cos(x)}{x^2}.$$

Računat ćemo vrijednosti funkcije f za neke malene vrijednosti x (npr. $x < 10^{-7}$) iz domene funkcije u aritmetici s pomičnim zarezom.

Veoma se jednostavno može pokazati (npr. koristeći L'hospitalovo pravilo) da vrijedi

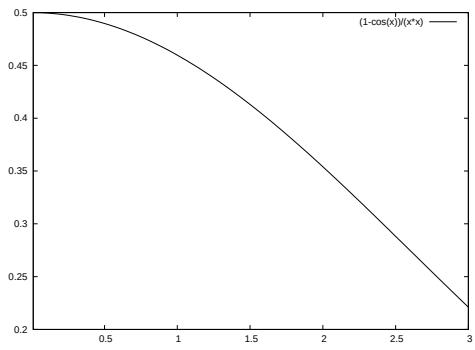
$$\lim_{x\to 0} \frac{1-\cos(x)}{x^2} = \frac{1}{2}.$$

Drugim riječima, za malene vrijednosti x očekivali bismo da kao vrijednosti funkcije $f(x)$ izračunavamo brojeve približno jednake 1/2 (slika 3.3, lijevo).

Međutim, u praksi će se pojavljivati problem. Promotrite sljedeći programski kôd napisan u `C++` programskoj jeziku (slika 3.3, desno).

Označimo sa y pravu vrijednost funkcije $f(x)$ u nekoj danoj točki x, te neka y_0 označava vrijednost koju računalo vrati nakon izračunavanja funkcije $f(x)$ u aritmetici s pomičnim zarezom.
Označimo s

$$\triangle y_0 = |y - y_0|$$




```
#include<iostream>
#include<math.h>
using namespace std;

float f(float x)
{
 float rezultat = (1-cos(x))/(x*x);
 return rezultat;
}

int main(){
   float x = 0.00000001;
   float rez = f(x);
   cout << "f(x) = " << rez << endl;

   return 1;
}
```

Slika 3.3: Graf funkcije $f(x) = (1-\cos(x))/x^2$ na intervalu $[0.01, 3]$, na lijevoj strani. Jednostavan programski kôd za izračunavanje funkcije $f(x)$ napisan u programskom jeziku C++, na desnoj strani.

apsolutnu pogrešku aproksimacije, a s

$$\delta y_0 = \frac{|y - y_0|}{|y|}$$

relativnu pogrešku aproksimacije, te promotrimo vrijednosti u tablici 3.22 koje donji programski kôd izračuna za navedene vrijednosti x.

x	$f(x)$	$\triangle y_0$	δy_0
10^{-7}	0.50000149	$1.490116 \cdot 10^{-6}$	$2.980232 \cdot 10^{-6}$
10^{-8}	0.49981722	0.012108	0.000365
10^{-9}	0.48789101	0.000182	0.024217
10^{-10}	0.0	0.5	1.0

Tablica 3.4: Eksperiment je napravljen u linux distribuciji Debian 2.6.26 te koristeći GCC 4.1.3 prevoditelj (*compiler*).

3.4 Binarni i alfanumerički kodovi

Kako je u svakodnevnom životu čovjeku jednostavnije upotrebljavati brojeve iz dekadskog brojevnog sustava, nego nizove nula i jedinica (brojeve iz binarnog

sustava), tako je računalima upravo obratno, tj. njima je mnogo jednostavnije raditi s nizovima nula i jedinica.

Kao rješenja tog problema razvijeno je nekoliko vrsta *dogovora* koji će biti pretpostavka za uspješno komuniciranje između ljudi i računala. To su dogovori o skupu znakova koji će se koristiti u radu s računalom i o pripadajućim binarnim kombinacijama za svaki od znakova.

Skup svih znakova koji se tako koriste naziva se **apstraktna abeceda**. Ona, zajedno s pripadajućim binarnim (ili nekim drugim) kombinacijama, tvori **kôd**. Pojedini znakovi u kodu nazivaju se **kodni elementi**, a pripadajuća im se zamjena, binarna kombinacija, neki drugi znak ili nešto treće, naziva kodnom zamjenom.

Mi ćemo spomenuti neke **težinske kodove** (BCD, npr.) i jedan alfanumerički kôd (ASCII).

3.4.1 Težinski kodovi

Broj kodnih elemenata u apstraktnoj abecedi naziva se **obim kôda**. Među popularnije 4-bitne kodove spadaju tzv. **težinski kodovi**.

U takvoj reprezentaciji, svaki od 4 bita pomoću kojih zapisujemo dekadski broj N ima svoju težinu, tj.

$$N = w_1 b_1 + w_2 b_2 + w_3 b_3 + w_4 b_4$$

gdje su b_i, $i = 1, \ldots 4$ bitovi, a w_i, $i = 1, \ldots 4$ težine. Niz binarnih znamenki koje predstavljaju dekadski broj N zovemo **kodna riječ**.

Jedan od najpoznatijih 4-bitnih kodova je BCD (**B**inary-**C**oded-**D**ecimal). BCD je težinski kôd s težinama 8-4-2-1 što odgovara *reprezentaciji decimalnog broja u binarnom obliku.*

Tablica 3.5 sadrži prikaz dekadskih znamenki u različitim kodovima BCD (8-4-2-1) i 2-4-1-2 kodu.

Primjer 3.23 *Prikažite dekadski broj* 9 *u* $2-4-2-1$*, odnosno u* $8-4-2-1$ *težinskom kodu.*

$$1 \cdot \mathbf{2} + 1 \cdot \mathbf{4} + 1 \cdot \mathbf{2} + 1 \cdot \mathbf{1} = 9_{(10)}, \quad 1 \cdot \mathbf{8} + 0 \cdot \mathbf{4} + 0 \cdot \mathbf{2} + 1 \cdot \mathbf{1} = 9_{(10)}\,.$$

Tako je reprezentacija broja 9 *u* $2-4-2-1$ *dana s* 1111*, dok je u* $8-4-2-1$ *kodu dana s* 1001*.*

Napomena 3.3 *Zapis broja u danom kodu* ***ne mora biti jedinstven:***

$$1 \cdot \mathbf{2} + 1 \cdot \mathbf{4} + 0 \cdot \mathbf{2} + 0 \cdot \mathbf{1} = 6_{(10)}, \quad 0 \cdot \mathbf{2} + 1 \cdot \mathbf{4} + 1 \cdot \mathbf{2} + 0 \cdot \mathbf{1} = 6_{(10)}.$$

dekadske znamenke	kôd 2-4-1-2	BCD kôd 8-4-2-1
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	1011	0101
6	1100	0110
7	1101	0111
8	1110	1000
9	1111	1001

Tablica 3.5: Zapis dekadskih znamenki u težinskim kodovima

Prirodno se nameću sljedeća pitanja:

a) Kako računalo zna koji kôd koristimo?

b) Može li se istovremeno koristiti više različitih kodova?

Odgovor na drugo pitanje je **ne**. Može se koristiti samo jedna vrsta koda. Za odgovor na prvo pitanje potrebno je definirati novu vrstu koda. To je tzv. **samokomplementirajući kôd**, odnosno kôd za koji vrijedi da se zapis broja $9 - N$ dobije komplementiranjem (zamjenom 0 u 1 i obratno) zapisa broja N. Tablica 3.6 prikazuje kako izgleda zapis broja N i $9 - N$ u samokomplementirajućem kodu.

Primjer 3.24 *U tablici 3.6 prikazani su zapisi dekadskih znamenaka u kodu 2-4-2-1 koji je samokomplementirajuć. Primijetite da se komplement brojeva N $(9 - N)$ dobivaju jednostavnim komplementiranjem pripadajućeg niza binarnih znamenaka.*

3.4.2 Kodovi za zapisivanje znakova

Do sada smo naučili kako računalo reprezentira brojeve (prirodne i racionalne) u svome svijetu jedinica i nula. Međutim, u svakodnevnom su nam pisanju potrebni i znakovi abecede, znakovi interpunkcije te posebni znakovi (npr. razmak između dvije riječi).

N_{10}	$2-4-2-1$	$(9-N)_{10}$	reprezentacija $9-N$ u $2-4-2-1$ kodu
0	0000	9	1111
1	0001	8	1110
2	0010	7	1101
3	0011	6	1100
4	0100	5	1011
5	1011	4	0100
6	1100	3	0011
7	1101	2	0010
8	1110	1	0001
9	1111	0	0000

Tablica 3.6: Samokomplementirajući kôd 2-4-2-1

Standardna američka tipkovnica sadrži najmanje 128 različitih znakova (koje treba kodirati):

Brojevi :	10	
Slova (velika i mala)	52	
Posebni znakovi	33	(uključujući !, @, #, $, %, ...)
Kontrolni znakovi	33	(uključujući Enter, space, backspace, ...)
UKUPNO	128	

Najmanji potrebni broj bitova za prikazivanje 128 znakova iznosi 7 (jer je $2^7 = 128$).

Najčešće korišteni alfanumerički kôd je tzv. **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), što prevedeno znači *američki standardni kôd za razmjenu informacija*. Tijekom razvoja računalne tehnologije definirano je više varijanti toga koda. Najčešće je u uporabi 8 bitna varijanta u kojoj je prvih 128 kombinacija (0-127) standardizirano, a drugih 128 kombinacija (128-255) dano na volju korisniku kako bi sam kreirao kodne elemente. Prvih 128 elemenata koda prikazano je u tablici 3.7.

Primjer 3.25 *Kako glasi ASCII reprezentacija poruke „Pozdrav svima"*

	P	*o*	*z*	*d*	*r*	*a*	*v*		*s*	*v*	*i*	*m*	*a*
dec	80	111	122	100	114	97	118	32	115	118	105	109	97
hex	50	6*F*	7*A*	64	71	61	76	20	73	76	69	6*D*	61

dekadski	hex.	binarni	vrijednost
000	000	00000000	NUL
001	001	00000001	SOH
002	002	00000010	STX
003	003	00000011	ETX
004	004	00000100	EOT
005	005	00000101	ENQ
006	006	00000110	ACK
007	007	00000111	BEL
008	008	00001000	BS
009	009	00001001	HT
010	00A	00001010	LF
011	00B	00001011	VT
012	00C	00001100	FF
013	00D	00001101	CR
014	00E	00001110	SO
015	00F	00001111	SI
016	010	00010000	DLE
017	011	00010001	DC1
018	012	00010010	DC2
019	013	00010011	DC3
020	014	00010100	DC4
021	015	00010101	NAK
022	016	00010110	SYN
023	017	00010111	ETB
024	018	00011000	CAN
025	019	00011001	EM
026	01A	00011010	SUB
027	01B	00011011	ESC
028	01C	00011100	FS
029	01D	00011101	GS
030	01E	00011110	RS
031	01F	00011111	US
032	020	00100000	SP
033	021	00100001	!
034	022	00100010	"
035	023	00100011	#
036	024	00100100	$
037	025	00100101	%
038	026	00100110	&
039	027	00100111	’
040	028	00101000	(
041	029	00101001	)
042	02A	00101010	*
043	02B	00101011	+
044	02C	00101100	,
045	02D	00101101	-
046	02E	00101110	.
047	02F	00101111	/
048	030	00110000	0
049	031	00110001	1
050	032	00110010	2
051	033	00110011	3
052	034	00110100	4
053	035	00110101	5
054	036	00110110	6
055	037	00110111	7
056	038	00111000	8
057	039	00111001	9
058	03A	00111010	:
059	03B	00111011	;
060	03C	00111100	<
061	03D	00111101	=
062	03E	00111110	>
063	03F	00111111	?

dekadski	hex.	binarni	vrijednost
064	040	01000000	@
065	041	01000001	A
066	042	01000010	B
067	043	01000011	C
068	044	01000100	D
069	045	01000101	E
070	046	01000110	F
071	047	01000111	G
072	048	01001000	H
073	049	01001001	I
074	04A	01001010	J
075	04B	01001011	K
076	04C	01001100	L
077	04D	01001101	M
078	04E	01001110	N
079	04F	01001111	O
080	050	01010000	P
081	051	01010001	Q
082	052	01010010	R
083	053	01010011	S
084	054	01010100	T
085	055	01010101	U
086	056	01010110	V
087	057	01010111	W
088	058	01011000	X
089	059	01011001	Y
090	05A	01011010	Z
091	05B	01011011	[
092	05C	01011100	
093	05D	01011101	]
094	05E	01011110	^
095	05F	01011111	_
096	060	01100000	‘
097	061	01100001	a
098	062	01100010	b
099	063	01100011	c
100	064	01100100	d
101	065	01100101	e
102	066	01100110	f
103	067	01100111	g
104	068	01101000	h
105	069	01101001	i
106	06A	01101010	j
107	06B	01101011	k
108	06C	01101100	l
109	06D	01101101	m
110	06E	01101110	n
111	06F	01101111	o
112	070	01110000	p
113	071	01110001	q
114	072	01110010	r
115	073	01110011	s
116	074	01110100	t
117	075	01110101	u
118	076	01110110	v
119	077	01110111	w
120	078	01111000	x
121	079	01111001	y
122	07A	01111010	z
123	07B	01111011	{
124	07C	01111100	|
125	07D	01111101	}
126	07E	01111110	
127	07F	01111111	DEL

Tablica 3.7: Tablice ASCII znakova

Decimalne znamenke	težinski kôd $8-4-2-1$	Paritet (parni)	Paritet (neparni)
0	0000	**0**	1
1	0001	**1**	0
2	0010	**1**	0
3	0011	**0**	1
4	0100	**1**	0
5	0101	**0**	1
6	0110	**0**	1
7	0111	**1**	0
8	1000	**1**	0
9	1001	**0**	1

Tablica 3.8: Decimalne znamenke s pripadajućim paritetima.

ASCII za kodiranje znakova koristi samo 7 bita. Međutim, na njemu se temelji i većina modernih znakovnika koji imaju veći raspon znakova od engleske abecede kao što su 8-bitni CP437, CP852, Windows-1250 i Windows-1252, te 16-bitni i 32-bitni Unicode.

3.4.3 Kodovi za otkrivanje pogrešaka

U prijenosu (transmisiji) binarnih podataka postoji mogućnost pogreške, tj. pojave *šuma* ili *buke*, a što je posljedica nesavršenosti uređaja. Ako do takve pogreške u komunikaciji dođe, to znači da je u nekim bitovima došlo do promjene nule u jedinicu ili obratno. Kôd i dalje može biti pročitan, ali sadržaj mu je netočan.

U ovom ćemo poglavlju prikazati jedan (najjednostavniji) način na koji možemo otkriti je li pročitani kôd ispravan i, ako nije, kako pronaći pogrešku te kako ju ispraviti.

Pretpostavimo da se može mijenjati samo jedan bit. Tada pogreške možemo otkriti i ispraviti pomoću **kôda za otkrivanje jedne pogreške**.

Kako bismo otkrili pogrešan bit, koristimo **paritet**. To je dodatni bit koji, u slučaju parnog pariteta, treba postaviti tako da cijeli kôd ima paran broj jedinica, ili za neparni paritet taj se bit definira tako da cijeli kôd ima neparan broj jedinica (tablica 3.8).

Komunikacija između dvaju računala. Računalo koje šalje informacije nazivamo **izvor**, a računalo koje prima informacije nazivamo **odredište** (cilj).

Postupak slanja i primanja poruka prikazat ćemo u 5 koraka:

(i) Ako **izvor** ne šalje podatke, onda se stalno prenosi niz jedinica.

(ii) Bit 0 određuje početak poruke.

(iii) Paritetni bit se postavlja po paritetnoj konvenciji koju koristimo (parni ili neparni paritet).

(iv) Nakon paritetnog bita stavljamo dodatnu jedinicu koja ukazuje na kraj poruke.

(v) Nakon toga **izvor** može nastaviti slati sljedeći byte ili nastaviti slati niz jedinica što odgovara prestanku poruke.

Primjer 3.26 *Pretpostavimo kako dva računala međusobno komuniciraju po konvenciji parnog pariteta u smjeru strelice*

$\longrightarrow$

0101011001	*0110100101*	*0110111101*	*0110110001*	*0110010101*
0110010001	*0111001111*	*0010000011*	*0110000111*	*0111001001*
0110010101	*0010000011*	*0110001011*	*0110110001*	*0111010111*
0110010101	*0010111001*			

Koristeći prethodnu tablicu provjeravamo svaki niz od 10 *bitova (8 bita je 1 byte i dodatno imamo 1 bit za prekid poruke i jedan paritetni bit):*

$\longrightarrow$

1.	*010101100*✓	*2.*	*011010010*✓	*3.*	*011011110*✓
4.	*011011000*✓	*5.*	*011001010*✓	*6.*	*011001000*⋆
7.	*011100111*✓	*8.*	*001000001*✓	*9.*	*011000011*✓
10.	*011100100*✓	*11.*	*011001010*✓	*12.*	*001000001*✓
13.	*011000101*✓	*14.*	*011011000*✓	*15.*	*011101011*✓
16.	*011001010*✓	*17.*	*001011100*✓		

Zadaci za provjeru znanja

Zadatak 3.39 Nađite grešku koja nastaje kao posljedica šuma u komunikacijskom kanalu između dvaju računala, ako znamo da se za otkrivanje greške koristi parni paritet i da poruka glasi ovako:

0110010001 0010101100 0111100000 0111101010 0100110000
0101010001

Zadatak 3.40 Nađite grešku koja nastaje kao posljedica šuma u komunikacijskom kanalu između dvaju računala, ako znamo da se za otkrivanje greške koristi neparni paritet i da poruka glasi ovako:

0010111000 0111010101 0001110001 0111001001 0101101000
0000101101

POGLAVLJE 4

Planiranje i razvoj programa

4.1 Osnove pisanja programa

Stvaranje računalnih programa proces je stvaranja niza uputa potrebnih računalu za obradu pojedinih informacija (*programski kod* ili *kod*). Takav niz računalu govori kako treba obraditi pojedine informacije.

Osnovna struktura računalne obrade podataka prikazana je sljedećom shemom:

Ulazni podaci $\Rightarrow$	Obrada podataka $\Rightarrow$	Izlazni podaci
(**Input**)	(**Process**)	(**Output**)

Tablica 4.1: Shematski prikaz rada računala

Dakle, u postupku obrade podataka računalo koristi niz uputa dobivenih od programera. U sljedećem ćemo primjeru na jednostavnom problemu prikazati *niz uputa* koje ilustriraju svaki od tri osnovna dijela rada računala.

Primjer 4.1 *Što bi bili ulazni, a što izlazni podaci i kako bi izgledala njihova obrada, ako želimo odrediti prosječnu ocjenu studenata 3. godine iz predmeta Uvod u numeričku matematiku za prošli semestar?*

ULAZ: *Popis svih pozitivnih ocjena iz Uvoda u numeričku matematiku za prošli semestar.*

OBRADA: *Zbrojimo sve ocjene i podijelimo ih s ukupnim brojem ocjena.*

IZLAZ: *Dobiveni je broj prosječna ocjena studenata 3. godine iz Uvoda u numeričku matematiku za prošli semestar.*

4.2 Programski jezici

U ovom ćemo poglavlju proučavati osnovna pravila i načine pisanja programskih jezika (*programiranja* ili *kodiranja*). Nakon što riješimo problem (konstruiramo rješenje) i načinimo dijagram toka (*pseudokod*), prelazimo na pisanje programa. Programe ćemo pisati u programskim jezicima C i C++, pri čemu ćemo uvijek istaknuti razliku u obliku programa, ako ona postoji.

Osvrnimo se najprije na povijesni razvoj programskih jezika. Najstarija računala zahtijevala su od programera prijevod svih algoritama na strojni jezik. Takav je pristup dao još veću važnost izučavanju algoritama jer je učinkovitiji algoritam smanjivao broj redova programskoga kôda i broj prepisivanja heksadecimalnoga kôda s kojima se opisivala naredba, a time i vjerojatnost pogreške.

Tako su se razvili programi *asembleri* koji su program napisan mnemoničkim kodom prevodili u strojni jezik. S vremenom, strojni se jezik smatra prvom generacijom programskih jezika, a *asembler* drugom generacijom. Iako je *asembler* imao niz prednosti u odnosu na strojni jezik, ipak mu je nedostajalo cjelovitije okruženje.

Prvi nedostatak programa napisanog u *asembleru* jest ovisnost o procesoru kojemu je namijenjen i nije ga jednostavno prenositi na računalo s drugim tipom procesora (treba voditi računa o novom obliku instrukcijskog skupa i drugačijim registrima). Drugi je nedostatak način koji je programer prisiljen usvojiti ako želi nešto isprogramirati. Programer je prisiljen razmišljati o malim, inkrementalnim koracima strojnoga jezika. Situaciju možemo usporediti s problemom opisa gradnje kuće ako bismo morali razmišljati o položaju svake cigle, daske i slično. U nekom je trenutku dijelove kuće potrebno predstaviti i na takav način, ali je o funkcionalnosti projekta lakše i razumljivije govoriti koristimo li veće jedinice poput sobe, kuhinje, vrata, prozora i sl.

Ukratko, elementi od kojih je proizvod sastavljen ne moraju biti oni koje koristimo kada ga projektiramo. Projektirati je lakše na višoj razini na kojoj svaki dio označava koncept. Slijedom ove filozofije počeli su se razvijati novi programski jezici prilagođeniji razvoju *softwarea* od asemblerskih jezika. Tako

su nastali jezici treće generacije koji su se od prethodne razlikovali po tome što su bili na višoj razini i neovisni o procesoru. Tipični primjeri su FORTRAN (FORmula TRANslator) za matematičke i znanstvene proračune i COBOL (Common Business Oriented Language) kojega je za poslovne aplikacije razvila mornarica Sjedinjenih Američkih Država.

Bez obzira na naprednu tehnologiju računalo i dalje "razumije" samo nizove nula i jedinica, stoga se program koji programiramo mora "prevesti" na jezik računala. Postoje dvije osnovne vrste prevoditelja programa (*transletera*) koji prevode nizove naredbi u strojevni 0, 1 jezik:

Compileri i **Interpreteri**

- **Compiler** je prevoditelj nekog od viših programskih jezika (**samo jedanput**). Prevodi cijeli program nakon čega se program izvodi koristeći tzv. *izvedbeni oblik* (najčešće s ekstenzijom .exe za Windows operativne sustave).

  Primjeri: **C, C++, Turbo Pascal, Fortran 99,** ...

- **Interpreter** je prevoditelj koji prevodi i odmah izvodi redak po redak programa. Ako naiđe na pogrešku u određenom retku, prevoditelj prekida daljnje prevođenje na tom mjestu.

  Primjeri: **GVBasic, MatLab, Mathematica, Python, Perl,** ...

Posebno mjesto zauzimaju jezici koji su i **compileri** i **interpreteri**, npr. Java. Izvorni se Java kôd prevodi („kompajlira") u posebni *bytecode* koji se može izvoditi na različitim operacijskim platformama. Postoje i Java programi koji se zovu **apleti** (*applets*), a mogu se prenositi kroz *World Wide Web* i mogu se izvršavati na bilo kojem *Java-aware browseru*.

4.3 Kontrolne strukture i pisanje programa

U sljedećih ćemo nekoliko poglavlja pokušati shvatiti logiku programiranja u programskim jezicima proceduralnoga tipa. Osim proceduralnog načina programiranja (programski kôd se strukturira u procedure, tj. funkcije i podprograme), postoje i neki drugi načini slaganja programskog kôda, poput objektno orijentiranoga programiranja (OOP) ili funkcionalnog programiranja (FP).

U kontekstu proceduralnog slaganja programskoga kôda, pokušat ćemo najprije apstraktno shvatiti pojmove i logiku u pozadini, a na kraju sve implementirati u stvarnom programskom jeziku `C/C++`. Načelno ćemo više koristiti `C++`

budući da će nam tako biti pojednostavljeno ispisivanje varijabli korištenjem objekta `cout`, a i imat ćemo nešto više slobode pri pisanju samih programa. Svi bi se problemi na vrlo sličan način pisali i u samom `C` jeziku.

Zašto `C++`, a ne neki drugi jezik? Odgovor leži u činjenici da je danas `C++` najpopularniji programski jezik koji omogućava programeru visok stupanj kontrole nad samom programskom implementacijom. Iako ga se ne smatra najprikladnijim za sve vrste programiranja (npr. u mrežnim klijent-server implementacijama vjerojatno ćete češće koristiti Javu ili neki skriptni jezik), česta je uzrečica među programerima: „Ako znate programirati u `C++`-u, onda znate programirati u svakom programskom jeziku".

Međutim, idućih nekoliko poglavlja ipak neće biti usmjereno na objašnjavanje `C++` jezika koliko na samu logiku programiranja. Ako želite detaljnije učiti programski jezik `C++`, vjerojatno ćete radije pogledati neke od široke lepeze knjiga koje se njime bave. Važno je imati na umu da ćete svaki `C++` program započinjati kao što je prikazano u slici 4.1.

```
#include<iostream>

// izvan glavne funkcije main() pišemo funkcije i definiramo globalne varijable
using namespace std;
int main(){

 // unutar funkcije main() pišemo programske retke.

 return 1;
}
```

Slika 4.1: Unutar funkcije main() za sada pišemo glavninu kôda. Funkcije i globalne varijable (a o njima ćemo više učiti kasnije) ćemo pisati izvan funkcije main(). Primijetite da retke u kojima zapisujemo komentare započinjemo s //.

4.3.1 Komentiranje u programu

Svaki programski jezik, pa tako i `C/C++`, omogućuje programeru u samom programu zapisati objašnjenja pojedinih dijelova programa tako da prevodilac (*compiler*) tijekom izvršenja programa taj dio jednostavno zanemari (preskoči).

Iako komentiranje programa iziskuje dodatno vrijeme i napor, kod pisanja složenijih programa ono se redovito isplati. Dogodi li se da netko drugi mora ispravljati naš kôd ili da nakon duljeg vremena sami moramo ispravljati vlastiti

kôd (što je česta pojava), komentari olakšavaju razumijevanje originalnih autorovih namjera. Takvi dijelovi programa zovu se **komentari**. U jezicima `C/C++` oni se definiraju upisivanjem dviju kosih crta `//`, a sav tekst u retku iza njih prevoditelj zanemaruje, odnosno, tekst iza kosih crta je komentar.

Svatko tko želi ozbiljno pristupiti pisanju programa, mora biti svjestan činjenice da će, nakon što napiše stotinjak programa, početi zaboravljati čemu koji program služi pa je veoma važno da se na početku datoteke u kojoj se nalazi izvorni kôd programa stavi komentar koji sadrži osnovne podatke o programu: ime programera, ime datoteke u kojoj se nalazi osnovni kôd i način (sintaksa) na koji se program izvršava, kratki opis što program radi, popis varijabli, konstanti i funkcija koje koristi i slično.

4.3.2 Varijable i konstante

Varijabla je memorijska lokacija čiji se sadržaj može mijenjati tijekom izvođenja programa. **Konstanta** je memorijska lokacija čiji je sadržaj stalan tijekom izvođenja programa.

Računalo treba unaprijed znati koju memorijsku lokaciju će povezati s kojom varijablom, odnosno konstantom, pa se i varijable i konstante moraju deklarirati unaprijed.

4.3.3 Vrste jednostavnih podataka

Prisjetimo se, memorijska lokacija sastoji se od niza nula i jedinica. Takav niz može predstavljati cijeli broj, realni broj (*floating-point* broj), ASCII kôd nekog znaka (karaktera) ili jednostavno 1-bitnu DA/NE vrijednost, itd. Na primjer, riječ od 2 bytea

$$[00000000 \quad 01000001]$$

može predstavljati broj 65, ako ju čitamo kao cijeli broj ili, može predstavljati slovo "A", ako ju čitamo kao ASCII kôd.

U tablici 4.2 prikazane su osnovne vrste podataka (tzv. primitivni tipovi podataka) koje koriste jezici C i C++.

Prije nego upotrijebimo pojedinu memorijsku lokaciju, moramo računalu unaprijed reći čemu je ona namijenjena, varijabli ili konstanti. Tada računalo *rezervira prostor u memoriji* za danu varijablu tijekom vremena u kojem se program izvršava.

Taj postupak zovemo **deklariranje** varijable (konstante).

vrsta podatka	veličina i ime
boolean	dostupno samo kod nekih *compilera*
karakter	1 byte - char
„mali" cijeli broj	2 bytea - short
cijeli broj	4 bytea - int
„dugi" cijeli broj	4 ili 8 bytea - long
realni broj (*floating point*)	4 bytea - float
realni broj dvostruke preciznosti	8 bytea - double

Tablica 4.2: Jednostavne vrste podataka u C/C++.

Programer tijekom izrade programa koristi ime varijable, ali ne i njezinu memorijsku lokaciju. Međutim, programski jezici C/C++ dopuštaju programeru korištenje određene memorijske lokacije, no takav način programiranja izlazi ih predviđenih okvira ove knjige.

naredba (C/C++)	objašnjenje
const int NUM = 2;	deklaracija cjelobrojne konstante vrijednosti 2 imena **NUM**
const float PI = 3.14;	deklaracija realne konstante vrijednosti 3.14 imena **PI**
int sum;	deklaracija cjelobrojne varijable imena **sum**
char myCh;	deklaracija varijable koja će sadržavati karaktere imena **myCh**
float interestRate;	deklaracija realne varijable imena **interestRate**

Tablica 4.3: Deklariranje varijabli i konstanti u C/C++.

Izbor imena varijable ovisi o programeru. Često su prema "nepisanom pravilu" konstante pisane velikim slovima PI, E, RATE, ..., dok se za varijable koriste mala početna slova: sum, myCh, ...

Napomena 4.1 C/C++ *razlikuje velika i mala slova (Windows ne razlikuje, ali Unix/Linux operativni sustavi da), tako da su* `myCh` $\neq$ `mych`.

4.3.4 Pridruživanje vrijednosti

Nakon što smo varijabli izabrali ime i rezervirali memorijsku lokaciju, potrebno je varijablama dodijeliti vrijednosti.

Za takvo pridruživanje postoje tri mogućnosti:

- učitavanje vrijednosti iz datoteke
- unošenje podataka preko tipkovnice
- pridruživanje vrijednosti unutar programa

Sintaksa pridruživanja određene vrijednosti određenoj memorijskoj lokaciji ima oblik:

ime varijable = vrijednost

Pravila pridruživanja (*assignment rules*) su jednoznačna za bilo koji programski jezik:

i) Samo jedno ime varijable može biti s lijeve strane znaka jednakosti jer to predstavlja memorijsku lokaciju koja mora biti jednoznačno određena;

ii) Konstante ne mogu stajati s lijeve strane znaka jednakosti (jer se one ne mogu mijenjati);

iii) Veličina s desne strane jednakosti može biti konstanta, druga varijabla ili neki aritmetički izraz koji se sastoji od varijabli i konstanti;

iv) Sve što se nalazi s desne strane jednakosti ostaje nepromijenjeno;

v) Varijable i pridružene vrijednosti moraju biti istoga tipa.

Primjer 4.2 *Odredite sadržaj svake od varijabli:* initial *(character),* lenght *(integer, cjelobrojna varijabla),* areaOfSquare *(floating point, realna varijabla), nakon svakog od sljedećeg niza pridruživanja:*

a) prije nego se bilo što pridruži;

b) lenght = 5;

c) initial = 'P';

d) areaOfSquare = lenght;

e) areaOfSquare = initial;

f) areaOfSquare = 37.5;

g) lenght = areaOfSquare;

h) initial = areaOfSquare;

Rješenje: Rezultate sadržaja varijabli nakon izvedenih koraka od a) do h) prikazat ćemo u tablici 4.4.

Pridruživanje	Sadržaj varijabli nakon pridruživanja		
	initial	lenght	areaOfSquare
a) prije pridruživanja	nedefinirana	nedefinirana	nedefinirana
b) lenght=5	nedefinirana	5	nedefinirana
c) initial='P'	'P'	5	nedefinirana
d) areaOfSquare = lenght	'P'	5	5.0
e) areaOfSquare = initial	'P'	5	80.0
f) areaOfSquare=37.5	'P'	5	37.5
g) lenght=areaOfSquare	'P'	37	37.5
h) initial=areaOfSquare	'%'	37	37.5

Tablica 4.4: Prikazi različitih pridruživanja.

Pridruživanja pod b), c) i f) su jednostavna pa su pripadajuće vrijednosti varijabli upravo one učitane u tim recima. Primijetite da u retku d) cjelobrojna vrijednost 5 može biti spremljena u realnu (*floating point*) varijablu, pri čemu je vrijednost varijable areaOfSquare realni broj 5.0. Nadalje, u retku e) umjesto slova 'P' realnoj varijabli areaOfSquare pridružuje se ASCII vrijednost slova 'P'. U retku g) realni je broj pridružen cjelobrojnoj varijabli pa je rezultat toga zaokruživanje na 37. U posljednjem retku h) rezultat spremanja cijeloga broja u slovnu varijablu (*character*) dao je za rezultat odgovarajući ASCII znak (*character*) koji pripada broju (kodu) 37.

naredba	vrsta	objašnjenje
cin	objekt	kontrolira „vađenje” byteova iz niza koji ulaze s tipkovnice
>>	operator „ispisa”	uzima byteove iz ulaznog niza
cout	objekt	kontrolira „ubacivanje” byteova iz niza koji odlazi na zaslon
<<	operator „upisa”	stavlja byteove u izlazni niz
cerr	objekt	
$'\backslash n'$ ili endl	„konstantni manipulator”	ubacuje karakter za novi red tjera kursor u novi red

Tablica 4.5: Popis naredbi za unos i izlaz u C++.

4.3.5 Jednostavni unos i izlaz (Input/Output; I/O)

Prisjetimo se osnovnog oblika tijeka podataka u računalima: *unos, obrada* i *izlaz* (*input, process, output*).

Unos, odnosno izlaz (ispis ili spremanje) podataka općenito ovisi o tome otkuda čitamo podatke i kamo ih spremamo (upis s tipkovnice ili unos iz datoteke, ...).

Jednostavni unos i izlaz odnose se na upis podataka s tipkovnice i ispis rezultata na zaslonu.

4.3.5.1 Unos i izlaz u C++ jeziku

Općenito, za upisivanje i ispisivanje podataka u programskom jeziku C++ potrebno je učitati odgovarajuću biblioteku:

```
#include<iostream.h>
```

U tablici 4.5 navedene su neke od naredbi za upis i ispis iz `iostream.h` biblioteke.

4.3.5.2 Unos i izlaz u C jeziku

U programskom jeziku C najčešće korištene funkcije za ispis i upis podataka (byteova) su `printf` i `scanf`.

Za korištenje navedenih naredbi potrebno je učitati standardnu I/O biblioteku

kontrolni znak za oblikovanje izlaza za printf() („karakter” konverzije)	pridruženi argument koji će biti ispisan
`%c`	„karakter”
`%d`	cijeli broj
`%f`	*floating-point* broj
`%s`	*string*

kontrolni znak za oblikovanje izlaza za printf() (znakovi „prekida”)	učinak
`\a`	*beep* zvuk
`\b`	*backspace*
`\f`	nova stranica
`\n`	novi red
`\t`	horizontalni tabulator

Tablica 4.6: Tablice kontrolnih znakova.

```
#include<stdio.h>
```

Naredbu **printf** koristimo za standardne uređaje za izlaz, kao što su zaslon, neki formatirani zapis (u datoteku) i slično, dok **scanf** koristimo za učitavanje podataka sa standardnih uređaja (npr. tipkovnice).

Oblik naredbe:

printf(kontrolni znak za oblikovanje izlaza, lista varijabli)

scanf(kontrolni znak za oblikovanje izlaza, lista varijabli)

Za detaljan popis kontrolnih znakova vidi tablicu 4.6.

Primjer 4.3 *Ispišite na zaslonu poruku: „Pozdrav Svijetu” u C i C++.*

Program 1:

```
#include<stdio.h>
void main()
{
 printf("Pozdrav Svijetu");
}
```

Program 2:

```
#include <iostream.h>
int main (void)
{
  cout << "Pozdrav Svijetu\n";
}
```

4.3.6 Aritmetički izrazi

U dosadašnjim smo razmatranjima proučavali načine na koje se pojedinačne vrijednosti (bilo da se radi o konstantnim ili promjenjivim veličinama) pridružuju nekoj varijabli. Dakako, pridruživanje može biti mnogo složenije. Primjerice, varijablama možemo dodjeljivati rezultate „različitih funkcija" koje u sebi sadrže aritmetičke izraze.

Treba istaknuti da redoslijed računskih operacija odgovara standardnom aritmetičkom redoslijedu:

1.) izvode se operacije u zagradama,

2.) množenje, dijeljenje te zaokruživanje, i

3.) zbrajanje i oduzimanje.

Popis računskih operacija koje možemo izvoditi dan je u sljedećoj tablici:

operator (*oznaka*)	računska operacija	"smjer" izvođenja
()	zagrada	iznutra prema van
* /	množenje i dijeljenje	s lijeva na desno
%	modulo (zaokruživanje)	s lijeva na desno
+−	zbrajanje i oduzimanje	s lijeva na desno

Tablica 4.7: Računske operacije u `C/C++`

4.3.7 Primjer cjelovitog programa

Pokušajmo sada napisati prvi cjeloviti `C++` program koji će rješavati dani problem.

Primjer 4.4 *Za neki zadani racionalni broj r (radijus) izračunajte i ispišite površinu kruga* $P = r^2\pi$.

Rješenje ovog problema dano je sljedećim kodom:

```
#include<iostream>

using namespace std;
int main(){

 // zadamo radijus r kao neki racionalni broj
 float r = 12.4;

 // izračunamo vrijednost površine
 float P = (r*r)*3.14;

 // te ispišemo izračunatu vrijednost
 cout << "Površina kruga je: " << P << endl;

 return 1;
}
```

Prije nego što damo složenije primjere problema i njihovih rješenja, objasnit ćemo značenje operatora i njihovu sintaksu u C++ jeziku.

4.3.8 Booleova algebra

Računalni programi izvode točno ono što im je rečeno. Oni dobivene upute izvode u nizu, izborom i ponavljanjem. Te se upute većinom izvode „jedna za drugom" (u nizu), kako su i napisane.

Ponekad programer ima dva ili više mogućih izbora koji se mogu izvesti u ovisnosti o određenoj varijabli, odnosno Booleovoj varijabli koja može sadržavati jednu od dvije vrijednosti: *istina* ili *laž*.

Booleova algebra je, kao dio matematičke logike, algebarska struktura koja sažima osnovu operacija **I** (AND), **ILI** (OR) i **NE** (NOT) kao i skup teorijskih operacija kao što su unija, presjek i komplement. Booleova algebra je dobila naziv po svome tvorcu, Georgeu Booleu, britanskom matematičaru iz 19. stoljeća. Booleova algebra je, osim kao dio apstraktne algebre, izuzetno utjecajna kao matematički temelj računarskih znanosti.

4.3.8.1 Booleove varijable i konstante

Booleove varijable mogu primati dvije vrijednosti, *istinu (true)* ili *laž (false)*. Preporuča se takvim varijablama davati opisna imena kako bi smo se prilikom čitanja programskoga kôda lakše snalazili.

Sljedeći jednostavni primjer u C++ prikazuje kako možemo koristiti Booleove varijable:

```
const bool ISTINA = true ; // deklaracija konstante istine
bool LAZ = false;   // deklaracija laži
```

Booleov izraz se sastoji od konstanti i varijabli povezanih relacijskim operatorima.

4.3.8.1.1 Operatori nejednakosti

- operator „manje", $<$,
- operator „manje ili jednako", $<=$,
- operator „veće", $>$,
- operator „veće ili jednako", $>=$,

4.3.8.1.2 Operatori jednakosti

- operator „jednako", $==$,
- operator „različito", $!=$.

Primjer 4.5 *Neka su zadane vrijednosti* $x = 3, y = 4$. *Tada vrijede sljedeće tvrdnje:*

a) $x - 4 < y$. *Tvrdnja je točna, tj. istinita (eng. True).*

b) $x != y - 1$. *Tvrdnja je netočna tj. neistinita (eng. False).*

c) *"Domagoj"<"Dominik". Tvrdnja je točna (leksikografski uređaj).*

Pojedinačni se **Booleovi izrazi** mogu kombinirati u složenije izraze spajanjem pomoću logičkih operatora.

4.3.8.1.3 Logički operatori

- operator „negacija", ! ,
- operator logički „i", && ,
- operator logički „ili", ||.

Ako je logička tvrdnja istinita pretpostavit ćemo da je njezina vrijednost 1 ili T, a ukoliko je neistinita 0 ili F. Takav logički tip podatka koji može poprimiti dvije vrijednosti (istina ili laž) nazivamo Booleov tip (eng. *Boolean or logical data type*). Ilustrirajmo sljedećim tablicama implementacije logičkih operatora:

Operator negacije:

	0	1
!	1	0

Operator logički „i”:

$$0 \,\&\&\, 0 = 0, \quad 0 \,\&\&\, 1 = 0$$
$$1 \,\&\&\, 0 = 0, \quad 1 \,\&\&\, 1 = 1$$

Operator logički „ili”:

$$0 \,||\, 0 = 0, \quad 0 \,||\, 1 = 1$$
$$1 \,||\, 0 = 1, \quad 1 \,||\, 1 = 1$$

Primjer 4.6 *Neka su zadane vrijednosti* $x = -3, y = 4$. *Pokažimo jesu li sljedeće tvrdnje istinite ili lažne:*

$$a)\ \underbrace{\overbrace{(x == 3)}^{0} \,\&\&\, \overbrace{(y\,! = 3)}^{1}}_{0} \qquad b)\ \underbrace{\overbrace{(x > 0)}^{0} \,\&\&\, \overbrace{(y == 4)}^{1}}_{0}$$

$$c)\ \underbrace{\overbrace{(x > 0)}^{0} \,||\, \overbrace{(y == 4)}^{1}}_{1} \qquad d)\ \underbrace{\overbrace{(x < 0)}^{1} \,||\, \underbrace{\overbrace{(x == 3)}^{0} \,\&\&\, \overbrace{(y > 0)}^{1}}_{0}}_{1}$$

$$e)\ \underbrace{\overbrace{(y > 0)}^{1} \,\&\&\, \overbrace{(y < 100)}^{1}}_{1}$$

U većini programskih jezika, uključujući i `C/C++`, nije implementiran logički operator XOR, ili isključivi „ili” operator. Djelovanje XOR operatora prikazano je sljedećom tablicom:

$$0 \text{ XOR } 0 = 0, \quad 0 \text{ XOR } 1 = 1$$
$$1 \text{ XOR } 0 = 1, \quad 1 \text{ XOR } 1 = 0$$

U programskom jeziku `C/C++` XOR operator možemo jednostavno implementirati pomoću logičkih operatora „i”, „ili” i „negacija”, npr. $\text{XOR} = (x||y)\,\&\&\,!(x\&\&y)$.

Zadaci za provjeru znanja

Zadatak 4.1 Koja je vrijednost sljedećih izraza za zadane vrijednosti x i y?

Izraz	$x = 0, y = 4$	$x = 3, y = -2$	$x = -2, y = -1$
$x - 4 > y$			
$x + 1 == y$			
$x * y <= 0$			
$x! = 1 - y$			

Zadatak 4.2 Koja je vrijednost sljedećih izraza za zadane vrijednosti x, y i z?

Izraz	$x = -3, y = 4, z = 2$	$x = 3, y = -2, z = 0$	$x = y = z = 0$
$(x < z)\&\&(y > z)$			
$(y < z)\|\|(x < z)$			
$(x == -3)\|\|(y! = 3)\&\&(z < x)$			
$!((x < y)\&\&(z < y))\|\|(x + y + z < 1)$			

Zadatak 4.3 Napišite Booleove izraze kojima se provjerava istinitost sljedećih izjava:

a) *brzina* je točno 90km/h

b) x je iz intervala $[0, 100]$

c) x nije iz intervala $[0, 100]$

d) x je iz skupa $[0, 100] \setminus [10, 20]$

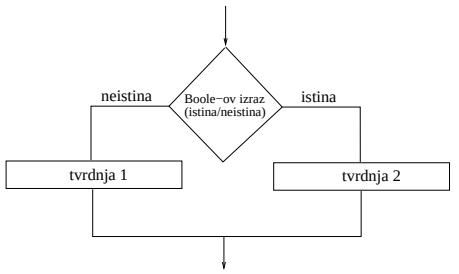
Zadatak 4.4 Na nekom se fakultetu iz nekog kolegija piše ispit koji se sastoji od pet zadataka od kojih svaki nosi najviše 20 bodova. Neki je student iz tih zadataka dobio redom `b1`,`b2`,`b3`,`b4` i `b5` bodova. Za prolaz na ispitu potrebno je skupiti barem 50 bodova i potpuno točno riješiti barem jedan zadatak. Napišite izraz koji provjerava istinitost sljedeće tvrdnje: „Student je položio ispit."

Zadatak 4.5 Provjerite svoja rješenja zadataka 4.1 i 4.2 u `C++`-u.

4.4 Kontrolne strukture

4.4.1 IF-ELSE kontrolna struktura

Praktična upotreba logičkih tipova podataka u programskim jezicima nalazi se u logičkim strukturama grananja. Promotrite ilustraciju `if-else` uvjetnog grananja prikazanog grafički u slici 4.2. Promotrimo sljedeći primjer i upotrebu `if-else` grananja u konkretnom problemu.




```
if (boole-ov izraz) {
   niz programskih redaka;
}

ili

if (boole-ov izraz) {
   niz programskih redaka;
}else {
   niz programskih redaka;
}
```

Slika 4.2: Ako je logički izraz neistinit, izvršava se *tvrdnja 2.* U suprotnom, izvršit će se *tvrdnja 1* (lijevo). Primjer implementacije `if-else` uvjetnog grananja u `C++` jeziku (desno).

Primjer 4.7 *Za neki zadani racionalni broj r (radijus) i cjelobrojnu vrijednost K izračunajte površinu kruga $P = r^2\pi$. Ispišite:*

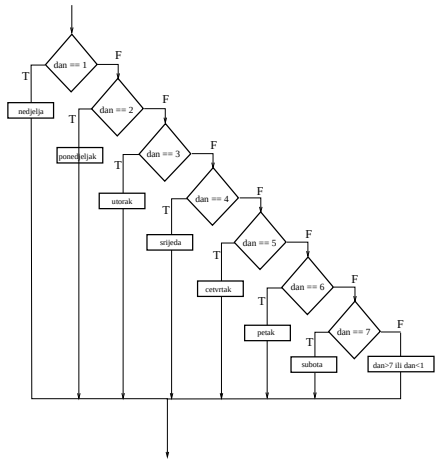
- *'Površina X je manja od K', ukoliko $P < K$;*
- *'Površina X je veća od K', ukoliko $P > K$;*
- *'Površina X je jednaka K', ukoliko $P == K$,*

gdje na mjestu X treba pisati odgovarajuća vrijednost izračunate površine P.

Rješenje ovog problema dano je sljedećim kodom:

```
#include<iostream>

using namespace std;
int main(){

 // zadamo radijus r i broj K
 float r = 12.4;
 int K = 20;

 // izračunamo vrijednost površine
 float P = (r*r)*3.14;

 // te ispišemo izračunatu vrijednost uz zadane uvjete
 if (P < K){
    cout << "Površina " << P << " je manja od K. " << endl;
 }else{
    if (P == K){
        cout << "Površina " << P << " je točno jednaka K. " << endl;
    }else{
        cout << "Površina " << P << " je veća od K. " << endl;
    }
 }
 return 1;
}
```

Kontrolna struktura `if-else` čini se veoma praktičnom kada treba izabrati između dva moguća izbora, istine ili laži. Međutim, čini se manje praktičnom ako treba izabrati između više mogućnosti. Promotrite sljedeći primjer.



Slika 4.3: Logički prikaz ispisivanja dana u tjednu.

Primjer 4.8 *Napišite programski kôd koji će za danu vrijednost od 1 do 7 ispisati odgovarajući dan u tjednu.*

Na slici 4.3 prikazan je logički slijed rješavanja problema, dok je na slici 4.4 prikazan programski kôd rješenja problema.

Napomena 4.2 *Ako se blok naredbi u jeziku* `C++` *koji se nalazi unutar dviju vitičastih zagrada sastoji od samo jedne naredbe, tada se vitičaste zagrade mogu ispustiti.*

I dalje valja voditi brigu o tome da programski kôd bude pravilno uvučen.

```
int dan = x; // gdje x zamijenite brojem od 1 do 7.
if (dan == 1){
    cout << "Nedjelja" << endl;
}else{
    if (dan == 2){
        cout << "Ponedjeljak" << endl;
    }else{
        if (dan == 3){
            cout << "Utorak" << endl;
        }else{
            if (dan == 3){
                cout << "Utorak" << endl;
            }else{
                if (dan == 4){
                    cout << "Srijeda" << endl;
                }else{
                    if(dan == 5){
                        cout << "Cetvrtak" << endl;
                    }else{
                        if(dan == 6){
                            cout << "Petak" << endl;
                        }else{
                            if(dan == 7){
                                cout << "Subota" << endl;
                            }else{
                                cout << "Dani mogu biti samo od 1 do 7" << endl;
                            }
                         }
                     }
                 }
             }
         }
     }
 }
```

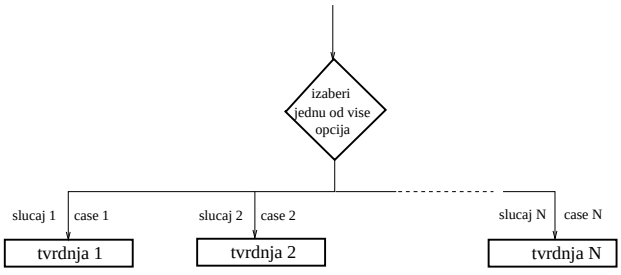
Slika 4.4: Programski kôd koji se nalazi unutar funkcije `main()` ispisivanja dana u tjednu.

Primijetite da se u slici 4.4 svaki blok naredbi sastoji zapravo od samo jednog `cout` retka. Upravo se zbog toga mogu ispustiti zagrade te napisati

programski kôd istoga značenja, ali mnogo čitljiviji.

4.4.2 SWITCH-CASE kontrolna struktura

U slučaju višestrukog izbora praktičnije je koristiti `switch-case` strukturu, čiji logički slijed vidimo u slici 4.5




```
switch(varijabla ili izraz)
{
   case (vrijednost1):
        programski kod;
        break;
   case (vrijednost2):
        programski kod;
        break;
   .
   .
   .
   case (vrijednostN):
        programski kod;
        break;
   default:
        programski kod;
}
```

Slika 4.5: Logički prikaz `switch-case` kontrolne strukture (lijevo). Sintaksa u `C++` jeziku (desno).

Zadaci za provjeru znanja

Zadatak 4.6 Napišite problem **ispisivanje dana u tjednu** iz primjera 4.8 pomoću `switch-case` kontrolne strukture.

Zadatak 4.7 Napišite `if-else` strukturu koja će ispisati je li broj smješten u cjelobrojnoj varijabli `mojBroj` paran ili neparan.

Zadatak 4.8 Napišite `C++` kôd koji će za zadanu varijablu `temp` ispisati „`toplo`" ako je temperatura veća od 20°, a inače „`hladno`".

Zadatak 4.9 Napišite `C++` kôd koji će za zadanu varijablu `god` ispisati je li to prijestupna godina ili nije. Godina je prijestupna ako je djeljiva s 4 i nije djeljiva sa 100 ili ako je djeljiva s 400 (znači, 2000. je bila prijestupna, a 1900. nije).

Zadatak 4.10 Napišite `C++` kôd koji će za zadanu cjelobrojnu varijablu `ocjena` ispisati „`nedovoljan`", „`dovoljan`", „`dobar`", „`vrlo dobar`" ili „`odličan`", ovisno o vrijednosti. Probajte napisati kôd koristeći prvo `if-else`, a zatim služeći se `switch-case` naredbom.

Zadatak 4.11 Koristeći `switch-case` naredbu napišite `C++` kôd koji će za zadani cijeli broj `br` ispisati je li:

- neparan
- djeljiv sa 2, a nije sa 4
- djeljiv sa 4, a nije sa 8
- djeljiv sa 8.

Zadatak 4.12 Na nekom se fakultetu iz nekog kolegija piše ispit koji se sastoji od pet zadataka od kojih svaki nosi najviše 20 bodova. Za prolaz na ispitu potrebno je skupiti barem 50 bodova i potpuno točno riješiti barem jedan zadatak. Za ocjenu dobar treba skupiti bar 70 bodova i u potpunosti riješiti bar dva zadatka. Za četvorku treba imati bar 80 bodova i tri potpuno riješena zadatka, a za odličan treba najmanje 90 bodova i četiri potpuno riješena zadatka. Napišite `C++` program koji će za zadane bodove `b1`,`b2`,`b3`,`b4` i `b5` ispisati pripadajuću ocjenu.
Možete li riješiti problem koristeći `switch-case` naredbu?

4.4.3 Pravilno uvlačenje programskog kôda

Pravilno uvlačenje redaka kôda ili indentacija kôda od iznimne je važnosti pri programiranju. Promotrite sljedeći primjer i greške koje mogu nastati pri zaključivanju zbog toga što je kôd neispravno indentiran.

Primjer 4.9 *Promotrite sljedeći kôd i pokušajte odgovoriti koji će redak biti ispisan.*

```
temp = 30; // temperatura je 30 stupnjeva
if (temp > 30)
    if (temp > 40)
        cout << "Dan je vruc" ;
else
    cout << "Dan je lijep" ;
cout << "Dan je suncan" << endl;
```

Točan odgovor je, naravno, „Dan je suncan". Međutim, zbog toga što je `else` *u petom retku krivo indentiran, vrlo lako bi svatko od nas mogao pogriješiti te zaključiti kako će uz poruku „Dan je suncan" biti ispisana i poruka „Dan je lijep".*

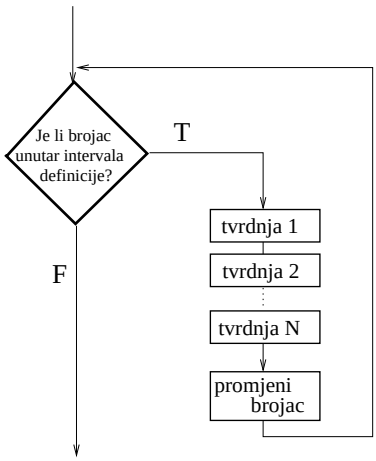
4.5 Ponavljanja ili petlje

Ponavljanja ili petlje u programskim jezicima koristimo kada želimo ponavljati iste retke programskoga kôda više puta. Naprimjer, pretpostavite da vas netko zatraži da napišete programski kôd koji ispisuje vaše ime i prezime točno n puta. Jedno od rješenja problema, iako veoma nepraktično, bilo bi da n puta napišete identičnu naredbu za ispis u programskom jeziku kojega koristite. Elegantno bi rješenje bilo korištenje mehanizma ponavljanja ili petlji. Razlikujemo tri različite vrste petlji u programskom jeziku `C++`, a to su: `for` petlja, `while` petlja te `do-while` petlja.

4.5.1 `for` petlja

Ta se petlja koristi za ponavljanje nekog programskog kôda točno određeni broj puta. Broj ponavljanja je određen specijalnom cjelobrojnom varijablom koju nazivamo brojač te intervalom u kojemu je brojač definiran (vidite sliku 4.6). Promotrite primjer `for` petlje

```
for(int i=1; i<=10; i++)
{
   cout << "i = " << i << endl;
}
```




```
for(int br = pocetak; br # kraj; promjeni br)
{

   neki programski kod;

}
```

Slika 4.6: Logički slijed izvođenja `for` petlje (lijevo). Sintaksa `for` petlje u `C++`-u (desno), gdje oznaku # valja zamijeniti s jednim od operatora nejednakosti, $\leq$, $<$, $\geq$ ili $>$.. Varijabla `br` u `for` petlji označava brojač.

te pokušajte odgovoriti na sljedeća pitanja:

- Kako se zove brojač u gornjem primjeru?
- U kojem intervalu je brojač definiran?
- Na koji način je definirana promjena brojača nakon svake iteracije?
- Koliko puta će se ponoviti ispisivanje vrijednosti varijable `i` te kako će ispis izgledati?

Ako znate odgovoriti na sva gore postavljena pitanja, tada vrlo vjerojatno u potpunosti razumijete na koji način funkcionira ova petlja. Primijetite da je brojač cjelobrojna varijabla `i` koja poprima vrijednosti iz skupa $\{1, 2, \ldots, 10\}$. Sama promjena brojača definirana je s `i++` što znači da se brojač `i` inkrementira za 1 nakon svake iteracije. Dakle, brojač `i` će poprimiti sve vrijednosti iz skupa $\{1, \ldots, 10\}$, dok će naredba `cout` i ispisati te vrijednosti na standardni izlaz.

Primjer 4.10 *Pokušajte razumjeti sljedeći primjer te kako će izgledati ispis kôda.*

```
for(int x=20; x>9; x--)
{
   cout << "vrijednost brojaca  x = " << x << endl;
}
```

Za sam kraj, promotrite sljedeća dva primjera koja ispisuju sve parne brojeve od 10 do 99.

Primjer 4.11

```
for(int i=10; i<100; i=i+2)
{
   cout << i << endl;
}
```

```
for(int i=10; i<100; i++)
{
   if (i%2 == 0)
      cout << i << endl;
}
```

U lijevom primjeru brojač se inkrementira za 2 nakon svake iteracije. Budući je početna vrijednost brojača paran broj 10, `cout` *naredba će uistinu ispisati sve parne brojeve. U desnom se primjeru koristi binarni operator % koji vraća ostatak pri dijeljenju. Bez obzira što brojač tako poprima sve vrijednosti od 1 do 99, ispisuju se samo one koje su djeljive sa 2.*

4.5.2 `while` petlja

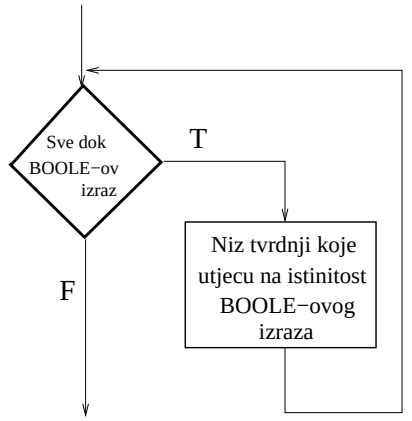
Ova se petlja koristi za ponavljanje nekog programskog kôda, a da unaprijed eksplicitno nije zadan broj ponavljanja. Sama petlja će se izvršavati sve dok je neki uvjet logičkog tipa ili tzv. Booleov izraz istinit (vidi sliku 4.7).

Promotrite sljedeći primjer korištenja `while` petlje:

```
int x = 5;
while(x>0)
{
   cout << "x = " << x << endl;
   x = x-1;
}
```

te pokušajte odgovoriti na sljedeća pitanja.

- Koliko iteracija će petlja imati i zašto?
- Što bi se dogodilo da redak `x = x-1;` izbacite, ili zamijenite s retkom npr. `x = x+1;`?




```
while(BOOLE-ov izraz istinit)
{

   Programski kod koji nužno mora utjecati na to hoće li
   BOOLEov izraz ostati istinit, ili će postati laž;

}
```

Slika 4.7: Logički slijed izvođenja `while` petlje (lijevo). Sintaksa `while` petlje u `C++`-u (desno).

- Što bi se dogodilo ako bismo zamijenili Booleov izraz `x>0` s `x<0`?

Primijetite da će se petlja ponoviti 5 puta te da će se u svakoj iteraciji petlje vrijednost `x` ispisati na standardni izlaz te dekrementirati za 1. Ukoliko retka `x=x-1;` ne bi bilo, sama petlja bi se ponavljala beskonačno dugo iz razloga što bi uvjet `x>0` uvijek bio istinit. I u slučaju da se `x=x-1;` zamijeni sa `x=x+1;` kao rezulat će se generirati beskonačna petlja. Ukoliko bi zamijenili Booleov izraz u `x<0`, kôd unutar same petlje nikad se ne bi izvršio.

Primjer 4.12 *Napišite program koji zbraja brojeve* $1+2+3+\ldots$ *sve dok suma ne postane veća ili jednaka 100 te ispiše dobivenu sumu.*

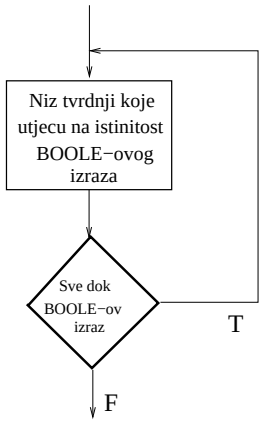
```
int suma = 0;
int broj = 0;
while(suma<=100)
{
   broj = broj + 1;
   suma = suma + broj;
```

```
}
cout << "suma je : "  << suma << endl;
```

Zadatak za vježbu 4.1 Primjetite da će u primjeru 4.12 varijabla `suma` poprimiti vrijednost veću ili jednaku broju 100. Modificirajte rješenje tako da dobivena suma bude strogo manja od 100.

4.5.3 `do-while` petlja

To je vjerojatno i najrjeđe korišteni oblik petlje ili ponavljanja. Vrlo je slična `while` petlji s tom razlikom da prvo izvrši kôd unutar petlje, a tek onda provjeri istinitost logičke tvrdnje (vidite sliku 4.8).




```
do {

   Programski kod koji nužno mora utjecati na to hoće li
   BOOLEov izraz ostati istinit, ili će postati laž;

}while(BOOLEov izraz istinit);
```

Slika 4.8: Logički slijed izvođenja `do-while` petlje (lijevo). Sintaksa `do-while` petlje u `C++`-u (desno).

Promotrite sljedeći primjer korištenja `do-while` petlje.

```
int x = 5;
do {
   cout << "x = " << x << endl;
   x = x-1;
} while(x>0)
```

Primjetite kako ne postoje razlike između ovoga kôda i identičnog primjera kojega smo koristili kao ilustraciju `while` petlje. Međutim, ako bi smo Booleov

izraz `x>0` zamijenili s `x<0`, tada bi ipak došlo do razlike. U kontekstu `do-while` petlje kôd unutar petlje bi se izvršio barem jednom, neovisno o tome je li Booleov uvjet zadovoljen ili nije.

4.6 Biblioteka matematičkih funkcija

Primijetite kako u `C/C++` nema operatora potenciranja, ali i mnogih drugih matematičkih elementarnih funkcija na koje smo naučili. Velika većina gotovih matematičkih funkcija može se koristiti samo uz dodatnu biblioteku matematičkih funkcija: `#include<cmath>` u slučaju Microsoft Visual `C++` prevoditelja, ili `#include<math.h>` u slučaju standardnog Linux (GCC) `C++` prevoditelja.

Neke od funkcija koje ta biblioteka sadrži su:

- apsolutna vrijednost broja a: `abs(a)`
- potencija (a na b): `pow(a,b)`
- drugi korijen od a: `sqrt(a)`
- sinus (odnosno kosinus ili tangens): `sin(a)`
- logaritam (prirodni logaritam) broja a: `log(a)`
- dekadski logaritam broja a: `log10(a)`

Zadaci za provjeru znanja

Zadatak 4.13 (Jednostavni kalkulator) Napišite program pomoću `do-while` petlje koji će prikazati sljedeći izbornik:

a) Zbrajanje

b) Oduzimanje

c) Množenje

d) Dijeljenje

e) Izađi van

Zadatak 4.14 Koji su rezultati sljedećih Booleovih izraza:

- `(5>4)||(3==2)`
- `(3>=4)&&(5<8)`
- `!(10<0)`
- `(5/2)==2.5`

Zadatak 4.15 Pomoću `if-else` napišite kôd koji će ispisati je li broj smješten u cjelobrojnoj varijabli mojBroj paran ili neparan.

Zadatak 4.16 Napišite program koji ispisuje znamenke dekadskog brojevnog sustava.

Zadatak 4.17 Napišite program koji ispisuje sumu svih cijelih brojeva između 100 i 200.

Zadatak 4.18 Napišite program koji ispisuje sve znamenke iz intervala $[0, 30]$ koje zadovoljavaju uvjet $a^2 + b^2 = c^2$ (Pitagorine trojke).

Zadatak 4.19 Napišite `C++` kôd koji će odbrojavati od zadane varijable `vrijeme` do 0.

Zadatak 4.20 Napišite C++ kôd koji će ispisati sve brojeve iz intervala $[1, 1000]$ koji su djeljivi sa 7 i 13. Brojeve ispisivati od najvećeg prema najmanjem.

Zadatak 4.21 Pretpostavite da je u banci oročeno 1000 kuna na 15 godina s kamatnom stopom 4% godišnje. Na početku svake godine kamate se pridodaju glavnici. Napišite C++ kôd koji će ispisivati iznos u banci za svaku godinu.

Zadatak 4.22 Napišite C++ kôd koji će ispisati tablicu množenja do 10.

Zadatak 4.23 Napišite C++ kôd koji će ispisati vrijednost a^b za prethodno definirane cjelobrojne varijable `a` i `b`.

Zadatak 4.24 Napišite C++ kôd koji će zbrajati cijele neparne brojeve počevši od 5 dok suma ne bude veća ili jednaka 100. Na kraju treba ispisati sumu i zadnji pridodani broj.

Zadatak 4.25 Napišite C++ kôd koji će zbrajati sve prirodne brojeve dok suma ne bude djeljiva sa 100. Na kraju treba ispisati sumu i zadnji pridodani broj.

Zadatak 4.26 Napišite C++ kôd koji će množiti prirodne brojeve djeljive sa 6 dok ukupni produkt ne bude veći od $4 \cdot 10^7$, ili broj iteracija ne bude veći od 5. Na kraju treba ispisati produkt, zadnjeg množitelja i broj iteracija.

Zadatak 4.27 Rješite zadatak 7 koristeći *do-while* petlju.

Zadatak 4.28 Napišite program koji ispisuje zbroj neparnih brojeva iz intervala $[0, 100]$.

Zadatak 4.29 Napišite program koji izračunava faktorijel unesenog cijelog broja.

Zadatak 4.30 Napišite program koji izračunava aritmetičku sredinu n učitanih brojeva (broj n treba prethodno učitati) i ispisuje najvećeg od njih.

Zadatak 4.31 Napišite program koji ispisuje brojeve veće od 500 i manje od 700 koji su djeljivi sa 17. Na kraju treba ispisati koliko ima takvih brojeva.

Zadatak 4.32 Napišite program koji ispisuje prvu potenciju broja 5 veću od 125, te ispisuje koja je to potencija.

POGLAVLJE 5

Funkcije i potprogrami

Pri pisanju programa od izuzetne je važnosti korištenje funkcija i potprograma koji programeru omogućuju podjelu programa na manje logičke jedinice i module.

Funkcije i potprogrami samostalne su programske strukture koje rješavaju neki problem. Za razliku od potprograma, funkcija uvijek vraća točno jednu vrijednost određenog tipa.

5.1 Pisanje funkcija i potprograma

Funkcija je programska struktura koja vraća neku vrijednost te može prihvaćati neke ulazne vrijednosti. Ulazne vrijednosti nazivamo *argumenti* ili *parametri* funkcije. Funkcija neku vrijednost vraća pomoću ključne riječi `return`, a sama se funkcija može pozivati unutar glavne funkcije `main` ili bilo koje druge funkcije ili potprograma.

O svakoj funkciji napisanoj u nekom programskom jeziku možemo razmišljati matematički kao o preslikavanju $f : X \to Y$, gdje je X skup svih ulaznih vrijednosti, a Y skup iz kojega dolazi vraćena vrijednost. Na primjer, promotrite funkciju $\max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ koja izračunava veći od dva cijela broja, tj. $\max(x, y) = \max\{x, y\}$. Iako je nama odmah jasno koji je od dva cijela broja veći, računalu ipak moramo eksplicitno objasniti kako izračunati veći od dva

cijela broja. Sljedeći kôd je upravo jedna takva implementacija funkcije max() u programskom jeziku C++.

```
int max(int x, int y)
{
 if (x>y)
    return x;
 else
    return y;
}
```

U programskim se jezicima svaka funkcija se sastoji od *zaglavlja* i *tijela*. U zaglavlju se funkcije naznačuje:

- ime funkcije (npr. `max`),
- tip vrijednosti koje funkcija vraća (npr. `int`),
- argumenti ili parametri funkcije (npr. `int x, int y`).

U tijelu se funkcije definira sâma funkcija. Tijelo funkcije određeno je dvjema zagradama `{` te `}`.

Zadatak za vježbu 5.1 Pokušajte sami napisati funkciju `int max(int a, int b, int c)` koja vraća najveći od tri cijela broja.

Promotrite sljedeći, nešto složeniji, primjer korištenja funkcija.

Primjer 5.1 *Napišite program koji će za neke racionalne vrijednosti a, b, c i d te prirodni broj n izračunati $a^n + b^n + c^n + d^n$ te ispisati dobiveni rezultat.*

Budući se računanje n-te potencije nekog broja ponavlja čak četiri puta u problemu, definirat ćemo funkciju `pot` *čiji će argumenti biti uređeni par nekog racionalnog i prirodnog broja, na primjer* `(float x, int e)`*, a vraćena vrijednost upravo prosljeđeni racionalni broj* `x` *na* `e`*-tu potenciju.*

```
float pot(float x, int e)
{
 float rez = 1;
 for(int i=1; i<=e; i++)
    rez = rez*x;
 return rez;
}
```

Unutar glavne funkcije `main` *sada jednostavno možemo izračunati traženu vrijednost uzastopno pozivajući funkciju* `pot` *s odgovarajućim argumentima.*

```
int main()
{
 // ... unos brojeva a, b, c, d, n

 float rezultat = pot(a,n) + pot(b,n) + pot(c,n) + pot(d,n);
 cout << "Trazeni zbroj potencija iznosi: " << rezultat << endl;

 return 1;
}
```

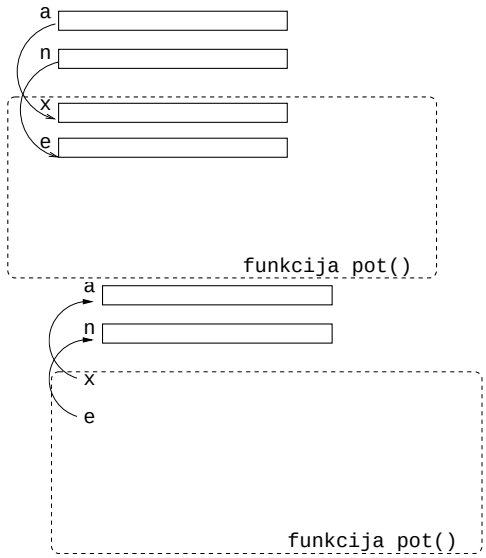
U gornjem primjeru zapravo možete uočiti svrhu pisanja funkcija u programskim jezicima. Umjesto ponavljanja jedne te iste `for` petlje četiri puta i gomilanja nepotrebnog koda, problem je elegantno riješen korištenjem funkcije `pot`.

Nadalje, primijetite kako parametri s kojima se funkcija poziva bivaju preimenovani imenima određenim u potpisu (zaglavlju) funkcije, pri čemu je od presudne važnosti pozicija parametra u samom funkcijskom pozivu. Naime,

```
        pot(      a,       n     );  |        pot(      b,       n     );
                  ↓        ↓       |                  ↓        ↓
  float pot(  float x,  int e   );  |  float pot(  float x,  int e   );
-------------------------------------------------------------------------
        pot(      c,       n     );  |        pot(      d,       n     );
                  ↓        ↓       |                  ↓        ↓
  float pot(  float x,  int e   );  |  float pot(  float x,  int e   );
```

Programsku strukturu koja može prihvaćati ulazne vrijednosti, ali ne vraća nikakvu vrijednost (tipa `void`) nazivamo **potprogram**. Sljedeća struktura je primjer jednostavnog potprograma.

```
void ispisi(int x)
{
 cout << "Vrijednost cjelobrojne varijable je: " << x << endl;
 return;
}
```



Slika 5.1: Prikaz prosljeđivanja po vrijednosti (lijevo). Varijable `x` i `e` postaju identične kopije prosljeđenih varijabli `a` i `n`. U slučaju prosljeđivanja parametara po memorijskoj lokaciji (desno), `x` i `e` pokazuju na iste memorijske adrese kao `a` i `n`.

5.1.1 Prosljeđivanje vrijednosti funkcijama i potprogramima

Postoje dva različita načina prosljeđivanja neke varijable kao argumenta funkcije (ili potprograma), a to je prosljeđivanje po vrijednosti i po memorijskoj lokaciji.

5.1.1.1 Prosljeđivanje po vrijednosti (*Pass-by-value*)

U uvom se slučaju vrijednosti varijabla koje se prenose funkciji kopiraju na novu memorijsku lokaciju. Ovakav način prosljeđivanja parametara funkciji do sada ste vidjeli i kod funkcije `max` i kod funkcije `pot`.

Pokušajmo shvatiti, na primjer, što se točno dogodilo pri pozivima funkcije

```
float pot(float x, int e).
```

Pri pozivu `pot(a,n)` došlo je do kreiranja racionalnog broja `x` te cijelog broja `e`, a zatim do kopiranja vrijednosti `a` u `x`, te `n` u `e` (vidi sliku 5.1, lijevo). Sama funkcija `max` ne vidi niti može pristupiti memorijskim lokacijama varijabli `a` i `n`,

međutim može raditi s njihovim kopijama. Isto se dogodilo i pri svim ostalim pozivima funkcije `pot`.

5.1.1.2 Prosljeđivanje po memorijskoj lokaciji (*Pass-by-reference*)

Pri prosljeđivanju parametara po memorijskoj lokaciji funkciji se prenosi memorijska lokacija ili adresa vrijednosti koje prosljeđujemo. Tako funkciji eksplicitno pokazujete gdje se u memoriji nalazi vrijednost koju prosljeđujete, a time funkciji dopuštate izravan pristup vrijednosti.

U `C++` jeziku memorijsku adresu neke vrijednosti možete dobiti korištenjem unarnog operatora `&`.

```
int x;
cout << "Cijeli broj x se nalazi na " << &x << " adresi." << endl;
```

Ilustracije radi, promotrimo na primjeru `pot` funkcije kako proslijediti parametre po memorijskoj lokaciji. Jedina promjena u sintaksi je u zaglavlju funkcije `pot`

```
float pot(float &x, int &e).
```

Primijetite kako će u tom slučaju `x` pokazivati na istu memorijsku lokaciju kao i `a`, a `e` na ista četiri bytea rezervirana za cijeli broj kao i `n` (vidi sliku 5.1, desno). Primjetite i kako svaka promjena unutar funkcije `pot` varijabli `x` i `e` uzrokuje izravnu promjenu na `a` i `n`.

Na sljedećem primjeru promotrimo korisnu primjenu prosljeđivanja parametara funkciji po memorijskoj lokaciji.

Primjer 5.2 *Napišite funkciju koja će zamijeniti vrijednosti varijablama* `a` *i* `b`.

```
void zamjeni(float &x, float &y)
{
 float z;
 z=x;
 x=y;
 y=z;
 return;
}
```

Unutar glavnog programa možemo pozvati funkciju `zamjeni()` *na sljedeći način:*

```
int main()
{
 float a=1.0, b=2.0;
 zamjeni(a,b);
 cout << "novi a = " << a << " i novi b = " << b << endl;
 return 1;
}
```

Pravu svrhu prenošenja vrijednosti po memorijskoj lokaciji shvatit ćete nakon poglavlja *Polja u računalima*.

5.2 Rekurzivne funkcije

Rekurzivna funkcija je svaka funkcija koja poziva samu sebe.

Primjer 5.3 *Promotrimo sljedeću definiciju funkcije* $f(n) = n!$ *napisane rekurzivno.*

$$f(n) = \begin{cases} 0, & za\ n < 0 \\ 1, & za\ n = 0 \\ n \cdot f(n-1), & za\ n > 0 \end{cases}$$

Prva dva retka u gornjoj definiciji su tzv. ***rubni*** *uvjeti, dok je u trećem retku dana rekurzivna definicija funkcije faktorijel. Rubni uvjeti su nužni kako bi rekurzija znala kada treba stati.*

Promotrimo sada jednostavnu C++ *implementaciju gornje rekurzivne definicije faktorijela:*

```
int fact(int n)
{
  if (n<=0) return 0;
  return n*fact(n-1);
}
```

Zadatak za vježbu 5.2 Rekurzivno zbrojite prvih n prirodnih brojeva.

Imajte na umu da računala „vole" rekurzije i rekurzivni način rješavanja problema. Međutim, svaki rekurzivni poziv troši dio memorije računala. Stoga se u stvarnosti mogu dogoditi dvije stvari. Ili će funkcija na vrijeme prestati pozivati samu sebe i vratiti izračunato, ili će računalo ostati bez memorije, ako će rekurzivnih poziva biti mnogo. Sljedećim primjerom ćemo pokušati „natjerati" računalo kako bi izbacilo grešku zbog prevelikog broja rekurzivnih poziva.

Primjer 5.4 *Promotrite sljedeću rekurzivnu funkciju koja ne radi ništa osim pozivanja same sebe i ispisivanja dubine rekurzije.*

```
#include<iostream>
using namespace std;
int i = 0; // globalna varijabla

void recursion()
{
    cout << i++ << endl;
    recursion();
}

int main()
{
  recursion();
  return 1;
}
```

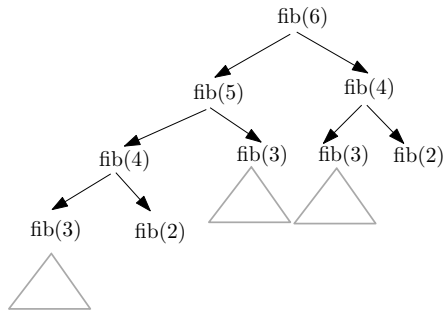
Kada će točno gornja rekurzija „puknuti" ovisi o mnogo parametara. Na Linux (Debian) operacijskom sustavu koristeći `gcc C++` *prevoditelj (verzija 4.3.2), gornji primjer je izbacio "Segmentation Fault" nakon točno 523791 funkcijskih poziva.*

Fibonaccijevi brojevi. Niz brojeva $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ naziva se Fibonaccijev niz. Fibonaccijev se n-ti broj rekurzivno definira na sljedeći način:

$$\text{fib}(n) = \begin{cases} 0, & \text{za } n = 0 \\ 1, & \text{za } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{za } n \geq 2 \end{cases}$$

Načiniti programsku implementaciju koja slijedi gornju rekurzivnu definiciju je sada uistinu lak posao.

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



Slika 5.2: Prikaz stabla rekurzije Fibonaccijevih brojeva s prvim pozivom fib(6). Primijetite kako već na trećoj razini rekurzije postoje dva istovjetna rekurzivna poziva fib(3), koja se računaju neovisno jedan od drugoga. Nadalje, za neki će se veći Fibonaccijev broj broj istovjetnih rekurzivnih poziva povećavati. Može se pokazati kako broj istovjetnih rekurzivnih poziva ima eksponencijalnu ovisnost o n-tom Fibonaccijevu broju.

Međutim, nije svako rekurzivno rješenje problema i dovoljno dobro rješenje. Vidite li zašto? Promotrimo što se događa kada računamo n-ti Fibonaccijev broj za $n = 6$ (vidi sliku 5.2).

5.3 Lokalne i globalne varijable

Lokalne varijable su varijable definirane unutar neke funkcije i njihov je „život" ograničen na tu funkciju. U svim slučajevima koje ste do sada vidjeli varijable su bile lokalnoga tipa. Postojanje svake lokalne varijable određeno je odgovarajućim blokom zagrada `{` i `}`. Točnije, svaka lokalna varijabla postoji od trenutka svoga nastanka pa sve do kraja odgovarajućeg bloka naredbi. Promotrite sljedeći primjer.

```
int suma_prvih_n_kvadrata(int n)
{
 int suma = 0;
 for (int i=1;i<n;i++)
 {
    int kv = i*i;
    suma = suma + kv;
```

```
 }
 cout << " i= " << i << " kv = " << endl; // pogreska
 return suma;
}
```

U gornjem smo primjeru stvorili tri lokalne varijable cjelobrojnog tipa, a to su `suma`, `i` te `kv`. Ako pokušate dohvatiti varijablu `i` ili `kv` izvan bloka naredbi `for` petlje, prevoditelj će prijaviti pogrešku.

Globalne se *varijable* definiraju izvan funkcija i žive od trenutka svoga postanka pa sve do kraja programskoga koda. Promotrite sljedeći primjer korištenja globalne varijable.

```
int x = 6; // globalna varijabla

void add_one()
{
 x = x+1;
 return;
}

void square()
{
 x = x*x;
 return;
}

int main()
{
 cout << " x = " << x << endl;
 square();
 add_one();
 square();
 cout << " x = " << x << endl;
 return 1;
}
```

Možete li pretpostaviti kolika će biti vrijednost varijable `x` pri prvom ispisu, a kolika pri drugom? Primijetite da je, od trenutka svog nastajanja, `x` dostupan svugdje, a njegovo djelovanje prestaje sa završetkom samog programa.

Nakon pokretanja programa, sve njegove globalne varijable, kao i one lokalne, zauzimaju određene pozicije u glavnoj memoriji. Memorijske pozicije globalnih varijabli ostaju nepromijenjene tijekom izvođenja cijelog programa, dok se pozicije lokalnih varijabli čuvaju samo tijekom izvođenja dijela potprograma, funkcije ili potprograma u kojem koristimo određenu lokalnu varijablu, nakon čega se te pozicije oslobađaju. Vidimo da, tijekom izvođenja glavnog programa, **globalne** i **lokalne** varijable imaju različito **vrijeme trajanja** (*lifetime*) u memoriji. Drugim riječima, vrijeme trajanja pojedine varijable ili konstante određuje koliko će dugo pridružena vrijednost određenoj varijabli ostati spremljena.

Dakle, **vrijeme trajanja globalne varijable** određeno je ukupnim vremenom izvođenja programa, dok je **vrijeme trajanja lokalne varijable** ograničeno na vrijeme izvođenja dijela potprograma, funkcije ili čitavoga potprograma u kojem koristimo lokalnu varijablu.

Prije nego objasnimo kako ćemo i kada koristiti lokalne, odnosno globalne, varijable istaknimo tri osnovna pravila koja treba zapamtiti pri izboru vrste (globalne ili lokalne) varijable.

(1) Osnovno je pravilo da **djelokrug djelovanja** ili *vidljivost* varijabli treba držati što je moguće uže. Nekontrolirano korištenje globalnih varijabli može prouzrokovati razne probleme, od nepotrebnog opterećenja memorije do velike mogućnosti pogreške zbog nepravilne promjene sadržaja globalne varijable. Kako bi se izbjegli takvi problemi, treba smanjiti upotrebu globalnih varijabli. Idealno bi bilo kad bi se globalne varijable tijekom izvođenja programa mijenjale samo jedanput .

(2) Treba optimizirati broj globalnih varijabli jer one ostaju u memoriji tijekom cijelog vremena u kojem se program izvodi, što opterećuje RAM.

(3) Treba izbjegavati davanje jednakih imena lokalnim i globalnim varijablama, jer ako lokalna varijabla ima isto ime kao i neka globalna, onda se toj globalnoj varijabli više neće moći pristupiti unutar tog dijela programa.

Zadaci za provjeru znanja

Zadatak 5.1 Napišite funkciju `poz()` koja ispisuje „Poziv!" pri svakom pozivu.

Zadatak 5.2 Napišite funkciju `zbroj()` koja vraća zbroj dvaju cijelih brojeva.

Zadatak 5.3 Napišite funkciju `uvecaj()` koja vrijednost proslijeđene varijable uvećava za 1. Izmjenite tu funkciju tako da joj se prosljeđivanjem dva cijela broja, prvi uveća za vrijednost drugoga.

Zadatak 5.4 Napišite funkciju `zamijeni()` koja zamjenjuje vrijednosti dviju cjelobrojnih varijabli. Koristeći tu funkciju, zamijenite vrijednosti četiriju varijabli na sljedeći način: $(a, b, c, d) \to (b, c, d, a)$.

Zadatak 5.5 Napišite funkciju `nx2()` koja će, za zadani racionalni broj `a` i cijeli broj `n`, `n` puta udvostručiti vrijednost od `a` i ispisati svaku od međuvrijednosti.

Zadatak 5.6 Napišite funkciju `f()` s dva cjelobrojna argumenta koja će mijenjati vrijednosti tih argumenata na sljedeći način:
ako je druga varijabla veća od nule, prvu će uvećati za vrijednost druge i drugu će umanjiti za vrijednost prve (istovremeno), inače će se drugoj promjeniti predznak;
ako je nakon toga prva varijabla veća od nule, promijenit će joj se predznak i dodat će se drugoj varijabli.

Zadatak 5.7 Rekurzivno izračunajte `n`-tu potenciju broja, pri čemu je `n` cijeli broj.

Zadatak 5.8 Napišite rekurzivnu funkciju `niz()` koja za zadani broj `n` vraća `n`-ti član niza zadanog s $a_0 = 1$, $a_1 = 3$, $a_n = a_{n-2} - a_{n-1} + 1$.

5.4 Strukturiranje podataka u tablice

U problemima su podaci često strukturirani na način prirodan upravo tom problemu. Promotrite sljedeći primjer.

Primjer 5.5 *Neka je zadan sustav linearnih algebarskih jednadžbi od tri jednadžbe s tri nepoznanice.*

$$\begin{aligned} x + 2y - 3z &= 0 \\ -x + y + z &= 7 \\ 2x + 2y + z &= 3 \end{aligned}$$

Iako je gornji sustav zadan s tri nepoznanice, primijetite kako smo mogli općenito zadati i sustav sa 100 ili 100000 nepoznanica.

Jednostavnosti radi, označimo nepoznanice s $x_1, x_2, \ldots x_n$, za neki prirodni broj n. Dakle, u gornjem primjeru $n = 3$.

Označimo sa $\mathbf{x}$ niz rješenja, pri čemu su komponente rješenja x_i, tj.

$$\begin{aligned} x_1 &\quad \text{– prva komponenta} \\ x_2 &\quad \text{– druga komponenta} \\ &\vdots \quad \vdots \\ x_n &\quad \text{– } n\text{-ta komponenta} \end{aligned}$$

Pogledamo li desnu stranu sustava, vidimo niz brojeva: $0, 7, 3$. Očito se i desna strana može složiti u cjelinu, koju ćemo nazvati $\mathbf{b}$, pri čemu su komponente od $\mathbf{b}$

$$b_1 = 0,\ b_2 = 7,\ b_3 = 3.$$

Pogledajmo sada koeficijente sustava. Oni su oblika:

$$\begin{matrix} 1 & 2 & -3 \\ -1 & 1 & 1 \\ 2 & 2 & 1 \end{matrix}$$

Vidimo da se gornji brojevi prirodno mogu organizirati u tablicu brojeva (odnosno matricu):

1	2	−3
−1	1	1
2	2	1

$$\equiv \begin{bmatrix} 1 & 2 & -3 \\ -1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix}$$

Ako cijelu matricu označimo npr. oznakom $\mathbf{A}$, onda pojedine koeficijente (brojeve iz tablice) „lociramo" (preciziramo) navodeći indeks retka i stupca u kojemu se nalaze:

$$\begin{aligned}
\mathbf{A}_{11} &= 1 \quad \text{element u 1. retku i 1. stupcu} \\
\mathbf{A}_{12} &= 2 \quad \text{element u 1. retku i 2. stupcu} \\
\mathbf{A}_{13} &= -3 \\
\mathbf{A}_{21} &= -1 \quad \text{element u 2. retku i 1. stupcu} \\
\mathbf{A}_{22} &= 1\,, \quad \mathbf{A}_{23} = 1 \\
\mathbf{A}_{31} &= 2\,, \quad \mathbf{A}_{32} = 2, \quad \mathbf{A}_{33} = 1\,.
\end{aligned}$$

Dakle $\mathbf{A}_{ij}$ je element koji se nalazi na presjeku i-tog retka i j-tog stupca.

Vidimo da je matrica

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -3 \\ -1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix}$$

dvodimenzionalno polje, tj. dvodimenzionalni niz, za razliku od vektora

$$\mathbf{b} = \begin{bmatrix} 0 \\ 7 \\ 3 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

koji su jednodimenzionalni nizovi, tj. **jednodimenzionalna polja**.

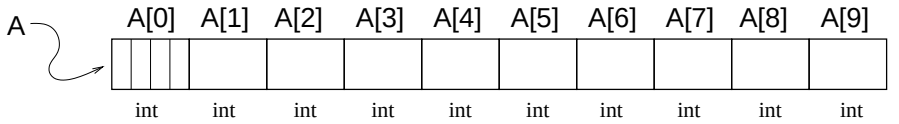
Dakle, neki su podaci organizirani *jednodimenzionalno*, a neki *dvodimenzionalno*.

U idućem ćemo poglavlju naučiti kako podatke organizirane u jednodimenzionalna i dvodimenzionalna polja reprezentirati u memoriji računala.

5.4.1 Polja u računalima

Polje se sastoji od niza uzastopnih memorijskih lokacija određenog tipa, a samo ime polja pokazuje na memorijsku lokaciju prvog bytea polja. Na primjer, u `C++` programskom jeziku, nakon naredbe `int A[10]` kreira se polje od 10 uzastopnih memorijskih blokova od 4 bytea (uz pretpostavku da računalo za reprezentaciju cijelog broja koristi 4 bytea) (vidi sliku 5.3).

Napomena 5.1 *Imajte na umu, indeksi polja u programskom jeziku* `C++` *kreću se od 0 do* $n-1$*, gdje* n *označava duljinu polja, tj. broj elemenata polja.*



Slika 5.3: Primjer polja od 10 cijelih brojeva. Deset memorijskih blokova od po 4 bytea nalaze se točno jedan iza drugoga u memoriji računala. Ime polja `A` pokazuje na adresu prvog bytea.

Osnovna prednost spremanja na uzastopne memorijske lokacije je u dohvaćanju elemenata polja. U gornjem primjeru polja `A` od 10 cijelih brojeva promotrimo što se točno događa pri dohvaćanju, na primjer, šestog elementa polja, tj. `A[5]`.

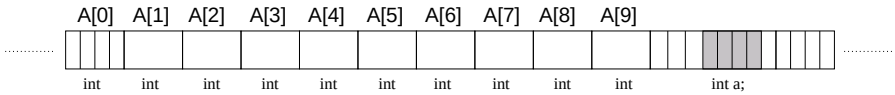
Ime polja `A` zna gdje se nalazi početak, tj. prvi byte polja. Koristeći operator `&`, kojeg smo već upoznali, lako možete i sami saznati adresu prvog bytea nekog polja `A` pomoću `&A[0]`[1]. Pomoću indeksa 5 jednostavno se možemo pomaknuti za potreban broj mjesta u memoriji, budući su brojevi spremljeni na uzastopnim lokacijama. Primijetite kako se na takav način svaki element polja dohvaća veoma brzo i gotovo neovisno o činjenici dohvaćamo li neki element polja od 10 elemenata ili od 10 000 elemenata.

Primjer 5.6 *Definirajte polje od 10 proizvoljnih racionalnih brojeva i ispišite ga.*

```
float A[10]={1.1,3.2,45.4,0.6,-122.4,23.23,1.0,-3.4,2.1,0.6};
for(int i=0;i<10;i++)
   cout << A[i] << " ";
```

Dakle, elemente polja možete strelovito brzo dohvaćati. Međutim, korištenje polja ima i svoje nedostatke poput naknadnog dodavanja ili brisanja elemenata polja. Pretpostavite, na primjer, da ne znate koliko veliko polje zapravo trebate u svom programskom kodu. Jednom kada izvršite naredbu poput `int A[10]`, nije moguće naknadno proširivati polje jer računalo možda upravo koristi byteove potrebne za proširenje postojećeg polja (vidi sliku 5.4). U tom bi slučaju jedno od rješenja bilo stvaranje novog polja s brojem elemenata većim od 10, a svih 10 elemenata polja `A` kopirati u novo polje te oslobodite memoriju računala od starog polja `A`.

[1] Ako ispišete `cout « &A[0];` kao rezultat ćete dobiti broj (najčešće u heksadecimalnom obliku, npr. 0xbfecb02cd, gdje 0x s početka broja dolazi iz svijeta UNIX računala i upozorava korisnika da se radi o broju zapisanom u bazi 16) koji označava fizičku adresu prvog bytea u radnoj memoriji računala.



Slika 5.4: Primjer polja od 10 cijelih brojeva i neke cjelobrojne varijable `a` u memoriji računala. Primijetite da polje ne možete proširiti budući da postoje samo tri slobodna bytea, a nakon toga memoriju zauzima varijabla `a`.

5.4.2 Prosljeđivanje polja kao parametra funkcije ili potprograma

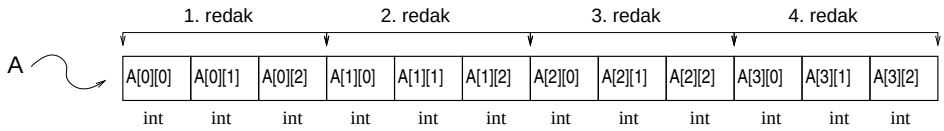
Polje možemo proslijediti nekoj funkciji kao parametar. Polja se prosljeđuju po *memorijskoj lokaciji*. U `C++` jeziku funkciji kao parametre prenosimo ime polja koje pokazuje gdje se polje nalazi u memoriji, te duljinu polja.

Primjer 5.7 *Sljedeć kôd primjer je potprograma koji ispisuje polje racionalnih brojeva. Obratite pozornost na zagrade* `[]` *koje zahtijeva sintaksa* `C++` *jezika pri prosljeđivanju imena polja.*

```
void ispisi(float A[], int n)
{
 for (int i=0; i<n; i++)
    cout << A[i] << " ";
 cout << endl;
 return;
}
```

5.4.3 Dvodimenzionalna polja brojeva – matrice

Dvodimenzionalno polje brojeva je jednostavno poopćenje polja brojeva. Takav ćete zapis koristiti kada želite spremiti neku tablicu (matricu) brojeva u računalo. U `C++` jeziku se nakon naredbe `int A[4][3]` kreira polje od 4×3 uzastopnih memorijskih blokova od 4 bytea potrebnih za spremanje cijelih brojeva. Iako se radi o matrici brojeva, sami brojevi će opet biti spremljeni na uzastopan niz blokova u memoriji računala, baš kao da ste izvršili naredbu `int A[12]` (vidi sliku 5.5). Svaki broj spremljen u određeni redak i stupac u matrici pohranjen je na točno određenu poziciju u memoriji koju računalo može jednostavno dohvatiti pomoću imena dvodimenzionalnog polja `A` te indeksa retka i stupca u kojemu se broj nalazi. Primjerice, memorijska adresa prvog bytea broja $A[i][j]$



Slika 5.5: Primjer dvodimenzionalnog polja tipa 4×3.

matrice A tipa $m \times n$ nalazit će se na

$$\&A[0][0] + (i * n + j) * 4$$

gdje $\&A[0][0]$ označava prvi byte polja A, a broj 4 dolazi od pretpostavke da cijele brojeve spremamo u točno 4 bytea.

Kako bi računalo znalo pristupiti broju na nekoj poziciji, npr. `A[2][3]`, mora interno uvijek znati od koliko se stupaca sastoji matrica. Stoga valja biti posebno oprezan pri prosljeđivanju dvodimenzionalnog polja kao argumenta funkciji te uvijek eksplicitno navesti broj stupaca.

Primjer 5.8 *Sljedeći kôd primjer je potprograma koji ispisuje matricu* 3×4 *cijelih brojeva. Obratite pozornost na zagrade* `[][]` *koje zahtijeva sintaksa* `C++` *jezika pri prosljeđivanju imena matrice. U drugoj zagradi je eksplicitno naveden broj stupaca matrice.*

```
void ispisi(int A[][4], int m, int n)
{
 for (int i=0; i<m; i++) // petlja kroz retke
 {
    for(int j=0; j<n; j++) // petlja kroz stupce
       cout << A[i][j] << " ";
    cout << endl;
 }
 cout << endl;
 return;
}
```

Primijetite da i elemente dvodimenzionalnog polja možemo veoma brzo dohvaćati i neovisno o veličini samoga polja.

Primjer 5.9 *Napišite funkciju koja množi dvije kvadratne matrice tipa* 4×4.

```
void pomnozi(int A[][4], int B[][4], int C[][4])
{
 for (int i=0; i<4; i++)
   for(int j=0; j<4; j++)
   {
      C[i][j] =0;
      for(int k=0; k<4; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
   }
 return;
}
```

Zadaci za provjeru znanja

Zadatak 5.9 Napišite program koji će ispisivati zadano jednodimenzionalno polje `a` duljine `n` unatrag.

Zadatak 5.10 Napišite program koji će u polje `F` duljine `n` na i-to mjesto spremiti i-ti Fibonaccijev broj.

Zadatak 5.11 Napišite funkciju `min()` koja će za zadano cjelobrojno polje i njegovu duljinu vraćati najmanju vrijednost polja.

Zadatak 5.12 Napišite funkciju `pro()` koja će za zadano cjelobrojno polje i njegovu duljinu vraćati prosječnu vrijednost polja.

Zadatak 5.13 Napišite funkciju `dia()` koja će za zadano cjelobrojno polje i njegovu duljinu vraćati razliku najvećeg i najmanjeg elementa polja.

Zadatak 5.14 Napišite funkciju `ispis()` koja će za zadano polje znakova `slova` i znak `z` ispisati `slova` sa `z` nakon svakog znaka.

Zadatak 5.15 Napišite funkciju `ispis2()` koja će za zadano polje znakova `slova` i znak `z` ispisati `slova` sa `z` između svakog znaka.

Zadatak 5.16 Napišite funkciju `aaa()` koja će za zadano polje znakova vratiti broj pojavljivanja slova `'a'`.

Zadatak 5.17 Napišite funkciju `izr()` koja će za zadano polje `y`, duljinu polja `n` i racionalne brojeve `a` i `b` na `x`-to mjesto polja spremiti vrijednost `ax+b`.

Zadatak 5.18 Napišite program koji će ispisivati zadano dvodimenzionalno polje `a` dimenzija `m`×`n`.

Zadatak 5.19 Napišite funkciju `ispis()` koja će ispisivati zadano dvodimenzionalno polje `a` dimenzija `m`×`n` (`m` i `n` su globalne konstante).

Zadatak 5.20 Usporedite dva niza (polja) cijelih brojeva (a i b) rekurzivno. Vratite 0, ukoliko $a == b$, vratite $+1$, ukoliko $a > b$, ili vratite -1 ukoliko $a < b$. Dva niza uspoređujte leksikografski.

Zadatak 5.21 Nađite rekurzivno najveći broj u polju od n elemenata.

POGLAVLJE 6

Uvod u algoritme

Kao što smo u uvodnom dijelu naveli, moderna su računala brza. Dobra računala danas mogu izvesti reda veličine 10^9 aritmetičkih operacija u sekundi (zbrajanja, oduzimanja, množenja i dijeljenja). Ilustrirajmo to na primjeru brzine svjetlosti.

Brzina svjetlosti iznosi približno $\sim 300000 \frac{km}{s}$ ili $3 \cdot 10^8 \frac{m}{s}$. To znači da dok svjetlost prijeđe nešto više od 3 metra, računalo zbroji dva broja.

U primjenama, za određene probleme razvijamo, razrađujemo i implementiramo odgovarajuće algoritme i očekujemo kako će za konkretne podatke naš algoritam dati **brzo** i **točno** rješenje. Algoritam možemo neformalno definirati kao konačan slijed dobro definiranih naredbi za rješavanje nekog problema.

U ovom ćete poglavlju naučiti nekoliko novih problema te algoritama za njihovo rješavanje. Sami će algoritmi biti napisani u pseudokodu, tj. bit će zapisani riječima. Pseudokod koji ćemo koristiti neće biti izravno prevodiv u neki programski jezik, međutim, sama struktura pseudokoda omogućavat će jednostavnu implementaciju u nekom od programskih jezika poput `C/C++`-a[1].

6.1 Složenost algoritama

Pretpostavimo na trenutak da su računala beskonačno brza te da je računalna memorija bezgranično velika. Međutim, i tada ima smisla proučavati algoritam

[1] Forma pseudokoda i sam predložak preuzeti su iz [2]

za neki problem i dokazati neka njegova svojstva, poput svojstva da algoritam završava svoje računanje u konačno mnogo koraka, pa i da ono što algoritam izračuna uistinu i jest točan rezultat.

Moderna računala jesu brza, ali svakako nisu beskonačno brza niti je računalna memorija bezgranično velika. Stoga kod pisanja algoritma za neki konkretan problem uvijek treba imati na umu da su resursi računala ograničeni i da ih treba pametno koristiti.

U praksi kada stvaramo neki algoritam za određeni problem, dobro je znati koliko je računskih (aritmetičkih) operacija[2] potrebno za izvođenje samog algoritma. To je prije svega praktičan problem jer u praksi želimo imati algoritam koji će nam u odgovarajućem vremenu dati rezultat. Na primjer, ako razvijamo algoritam koji daje preciznu vremensku prognozu za sljedeći dan, a njegovo izvršavanje traje 3 dana, onda očito takav algoritam nema smisla.

Kroz sljedeća dva primjera želimo ilustrirati kako se problemi mogu riješiti točno na više različitih načina. Međutim, neka su rješenja brža, a neka sporija, u smislu ukupnog broja računskih operacija koje koriste.

Primjer 6.1 *Rješenje kvadratne jednadžbe*

$$ax^2 + bx + c = 0\,, \qquad a \neq 0 \tag{6.1}$$

može se zapisati u obliku

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \end{aligned}$$

Primijetite kako smo upravo odredili algoritam koji za bilo koje podatke a, b, c ($a \neq 0$) daje dva rješenja x_1, x_2 kvadratne jednadžbe (6.1). Za $a = 0$ jednadžba (6.1) je linearna jednadžba $bx + c = 0$, koja u slučaju $b \neq 0$ ima rješenje $x = -c/b$. Konačno, u slučaju $a = b = 0$ i $c \neq 0$ nemamo rješenje, a za $a = b = c = 0$ rješenja su svi realni brojevi.

Pseudokod samog algoritma rješavanja kvadratne jednadžbe čije ćemo računanje prikazati kao funkciju SolveQuadraticEq() dan je u sljedećem obliku:

[2] Računske operacije su: apsolutna vrijednost, zbrajanje, oduzimanje, dijeljenje, ostatak pri dijeljenju, množenje te korjenovanje.

SolveQuadraticEq(a, b, c)

```
1  if a ≠ 0
2     then x1 = (−b+√(b²−4ac))/2a
3          x2 = (−b+√(b²−4ac))/2a
4          return x1, x2 ▷ Vrati x1 i x2 i izađi van iz funkcije
5  elseif b ≠ 0
6     then x1 = −c/b
7          return x1 ▷ Vrati x1 i izađi van iz funkcije
8  elseif c ≠ 0
9     then ▷ Prijavi da jednadžba nema rješenja
10    else ▷ Prijavi da su rješenja svi realni brojevi
```

Prebrojimo koliko je računskih operacija potrebno za rješavanje općenite kvadratne jednadžbe, tj. za računanje x_1 i x_2.

Na primjer: za x_1 imamo 4 množenja ($4 \cdot a \cdot c$, $2 \cdot a$ i b^2) i jedno korjenovanje, te jedno zbrajanje i jedno dijeljenje. Ukupno 7 računskih operacija[3]. Dakle, ukupan broj operacija za oba rješenja iznosi **14**.

Međutim, postavlja se pitanje postoji li algoritam koji računa oba rješenja kvadratne jednadžbe koristeći manje od 14 aritmetičkih operacija.

Uočimo prvo da se rješenja x_1 i x_2 mogu zapisati u obliku:

$$t = -\frac{b}{2a}\,, \qquad \Delta = \sqrt{t^2 - \frac{c}{a}}\,,$$
$$x_1 = t + \Delta\,, \qquad x_2 = t - \Delta\,.$$

Ukupan broj operacija tada iznosi 2 (jedno množenje i jedno dijeljenje) za t, 4 za Δ (dijeljenje, kvadriranje, oduzimanje i korjenovanje) i još 2 za x_1 i x_2 ukupno **8**.

Stoga gore navedeni postupak treba izvoditi samo u slučaju $a \neq 0$, u suprotnom smo vidjeli da rješenje jednadžbe (6.1) dobivamo mnogo jednostavnije.

Zadatak za vježbu 6.1 Modificirajte pseudokod SolveQuadraticEq() tako da algoritam rješava kvadratnu jednadžbu u najviše 8 aritmetičkih operacija.

Promotrimo na sličan način problem rješavanja sustava od dvije linearne jednadžbe s dvije nepoznanice:

[3] Imajte na umu da je operacija korjenovanja u praksi znatno sporija od elementarnih računskih operacija poput + ili ∗.

Primjer 6.2

$$\begin{aligned} 3x + 2y &= 7 \\ 6x - y &= 4 \end{aligned}$$

Jedan od načina na koji možemo rješavati gornji sustav jest da izrazimo x pomoću y i onda taj izraz uvrstimo u drugu jednadžbu što će značiti da smo dobili jednadžbu po y.

$$\begin{aligned} 3x = 7 - 2y \quad &\Rightarrow \quad x = \frac{1}{3}(7 - 2y) \\ 6(\frac{1}{3}(7 - 2y)) - y = 4 \,, 14 - 4y - y = 4 \quad &\Rightarrow \quad 5y = 10 \,, y = \mathbf{2} \\ x = \frac{1}{3}(7 - 2 \cdot \mathbf{2}) \quad &\Rightarrow \quad x = \mathbf{1} \end{aligned}$$

Napišimo sada algoritam za rješavanje linearnog sustava od dvije jednadžbe s dvije nepoznanice u općem slučaju.

$$\begin{aligned} A_{11}x + A_{12}y &= b_1 \\ A_{21}x - A_{22} &= b_2 \\ x = \frac{1}{A_{11}}(b_1 - A_{12}y) \qquad & \frac{A_{21}}{A_{11}}(b_1 - A_{12}y) + A_{22}y = b_2 \\ (A_{22} - \frac{A_{12}A_{21}}{A_{11}})y = b_2 - \frac{A_{21}}{A_{11}}b_1 \,, \qquad & (A_{11}A_{22} - A_{12}A_{21})y = A_{11}b_2 - A_{21}b_1 \\ & (\text{što vrijedi za } A_{11} \neq 0) \\ y = \frac{A_{11}b_2 - A_{21}b_1}{A_{11}A_{22} - A_{12}A_{21}} \qquad & \text{uz } A_{11}A_{22} - A_{12}A_{21} \neq 0 \end{aligned}$$

Uvrštavanjem y u jednakost za x dobivamo:

$$x = \frac{A_{22}b_1 - A_{12}b_1}{A_{11}A_{22} - A_{12}A_{21}}$$

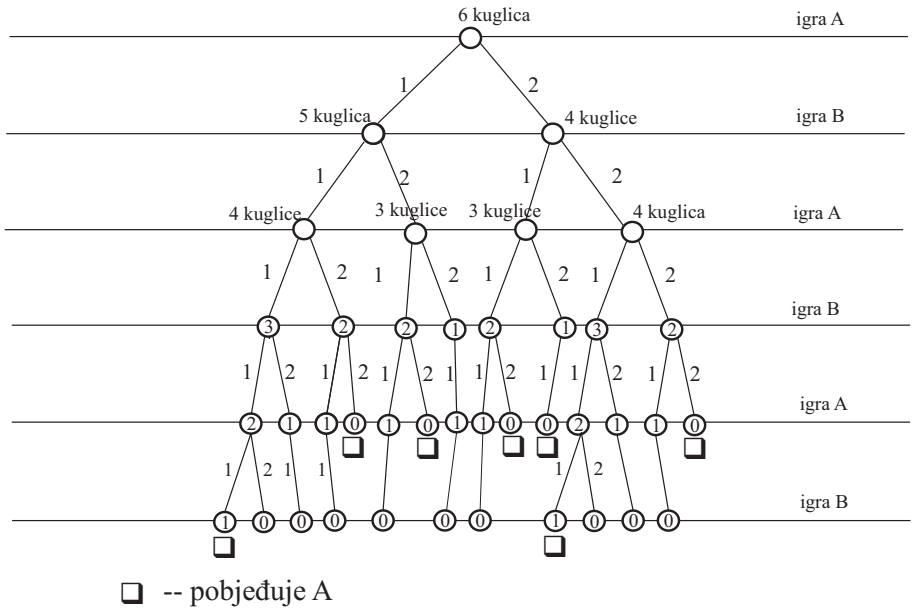
Dakle, rješenje linearnog sustava od dvije jednadžbe s dvije nepoznanice za koji vrijedi $A_{11}A_{22} - A_{12}A_{21} \neq 0$ jest:

$$\begin{aligned} \Delta_S &= A_{11} \cdot A_{22} - A_{12} \cdot A_{21} \,, \\ x &= \frac{b_1 \cdot A_{22} - b_2 \cdot A_{12}}{\Delta_S} \,, \\ y &= \frac{b_2 \cdot A_{11} - b_1 \cdot A_{21}}{\Delta_S} \,, \end{aligned}$$

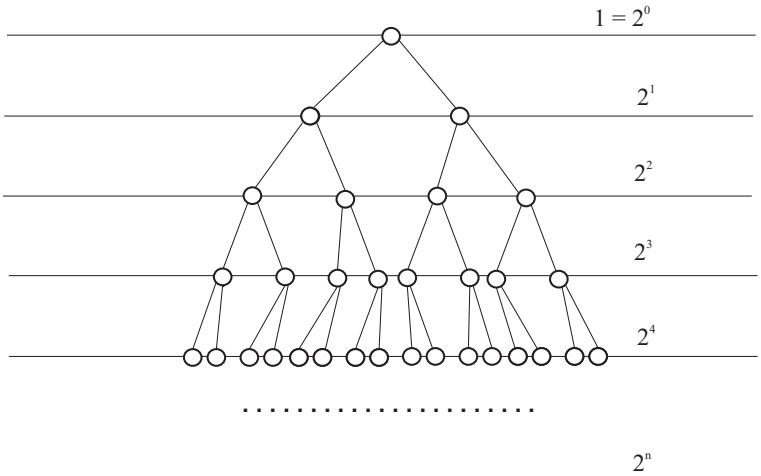
a za što je potrebno ukupno **11** operacija (6 množenja, 3 oduzimanja, dva dijeljenja).

Zadatak za vježbu 6.2 Napišite algoritam za rješavanje općenitog sustava dviju linearnih jednadžbi s dvije nepoznanice u pseudokodu.

Primjer 6.3 *Promotrimo sljedeću jednostavnu igru: Na stolu se nalazi* 6 *kuglica, igraju dva igrača koja naizmjence uzimaju jednu ili dvije kuglice. Gubi onaj igrač koji zadnji uzme kuglicu. Želimo li računalo naučiti tu igru, postavlja se pitanje kako, a pri čemu ga treba naučiti da pobjeđuje.*



Gornja igra odgovara sljedećem binarnom stablu



Ako želimo provjeriti sve mogućnosti u slučaju od n razina (što odgovara igri sa n kuglica pri čemu počinjemo od tzv. nulte razine), jasno je kako imamo 2^n mogućnosti.

Želimo li igrati s većim brojem kuglica, npr. $n = 100$, onda je za provjeru svih mogućnosti potrebno 2^{100} operacija (primijetimo da je uspoređivanje također jedna od operacija na računalu), odnosno

$$2^{100} = 2^{10^{10}} = (1024)^{10} > (1000)^{10} = 10^{30} \quad \text{operacija.}$$

Ako pretpostavimo da igrati učimo moćno računalo koje izvodi 10^{10} operacija u sekundi, treba nam više od

$$\frac{10^{30}}{10^{10}} = 10^{20} \text{ sekundi, odnosno preko } 10^{12}\text{godina.}$$

($godina = 31536000 sec.$ odnosno $godina < 10^8 sec.$)

Dakle, upravo smo pokazali kako se može dogoditi da je neki algoritam neizvediv u praksi, iako daje rješenje u konačno mnogo koraka.

6.1.1 Problem određivanja maksimalnog broja

Neka je $n \in \mathbb{N}$ i neka je zadan niz brojeva $X = x_1, x_2, \ldots, x_n$. Odredimo najveći broj zadanoga niza.

Dakle, naš je zadatak odrediti

$$x_m = \max_{1 \le i \le n} x_i = \max\{x_1, \ldots, x_n\}.$$

To znači kako treba pronaći indeks najvećeg broja iz danog niza.

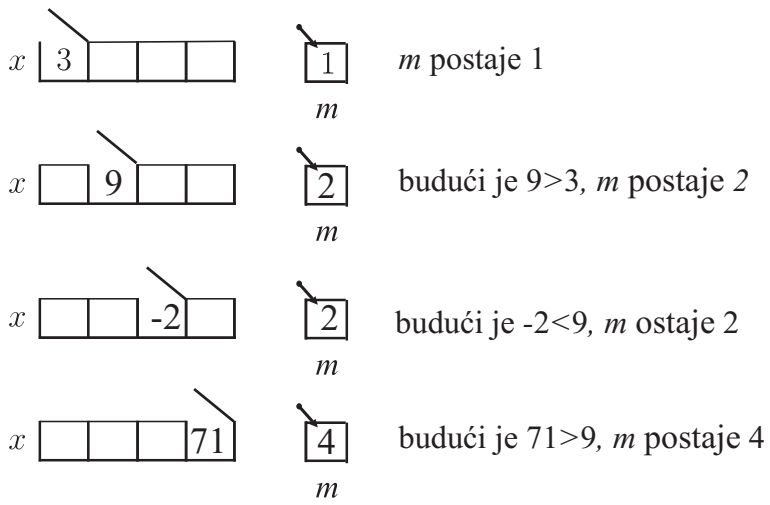
Na primjer, neka je dan niz od $n = 4$ broja, $x_1 = 3$, $x_2 = 9$, $x_3 = -2$ i $x_4 = -71$. Tada je rješenje problema $m = 2$, odnosno $x_m = 9$.

Prije nego počnemo *konstruirati* algoritam, moramo uzeti u obzir da pišemo algoritam koji će izvršiti računalo, a ono je jako ograničeno u svojim mogućnostima. Dakle, ulazni niz podataka u računalu niz je uzastopnih memorijskih lokacija u koje su upisane vrijednosti brojeva $x_1, \ldots x_n$. Treba naglasiti da se tom nizu može pristupiti samo element po element, pa naš problem zahtijeva novu, sličnu preformulaciju:

Zamislite da su naši brojevi $x_1, \ldots, x_n$ zapisani na kuglicama koje su stavljene u kutije sa n pretinaca od kojih svaki ima svoj poklopac. Kutija se zove x. Na početku su svi poklopci zatvoreni.

3	9	-2	71
x_1	x_2	x_3	x_4

U svakom trenutku možemo otvoriti samo jedan pretinac i pročitati broj na kuglici. Kako ćemo odrediti pretinac u kojemu je kuglica s najvećim brojem?



Pseudokod za ovaj program glasi:

$m \leftarrow 1$; MAX $= x_m$ (što znači da je MAX $= x_1$)

$$\begin{array}{c}
\text{ako je } x_2 > x_m, \text{ onda je } x_m = x_2 \text{ (odnosno } m \leftarrow 2\text{)} \\
\text{ako je } x_3 > x_m, \text{ onda je } x_m = x_3 \text{ (odnosno } m \leftarrow 3\text{)} \\
\text{ako je } x_4 > x_m, \text{ onda je } x_m = x_4 \text{ (odnosno } m \leftarrow 4\text{)} \\
\vdots \\
\text{ako je } x_n > x_m, \text{ onda je } x_m = x_n \text{ (odnosno } m \leftarrow n\text{)}
\end{array}$$

Cijeli postupak možemo kraće zapisati u obliku:

```
FindMax(X)
   ▷ vrati indeks max elementa iz polja X = ⟨x_1, x_2, ..., x_n⟩
1  max ← 1
2  for i ← 2 to n
3        do if X[i] > X[max]
4              then max = i
5  return MAX
```

6.1.2 Problem pretraživanja

U problemu pretraživanja dan je niz brojeva $A = a_1, \ldots, a_n$, $n \in \mathbb{N}$ i neki broj x. U problemu pretraživanja potrebno je utvrditi nalazi li se x u zadanom nizu brojeva te, ukoliko se nalazi, vratiti njegovu poziciju u nizu.

Na primjer, $x = 3$ nalazi se na petoj poziciji (ili četvrtoj, ako brojimo od nule) niza $8, 2, 4, 9, 3, 6$.

6.1.2.1 Linearno pretraživanje

Osnovna ideja kod linearnog pretraživanja je proći kroz niz brojeva A, s lijeva na desno, i za svaki element niza provjeriti je li jednak traženome x. Sljedeći pseudokod precizno opisuje svaki korak linearnog pretraživanja.

```
SequentialSearch(A, x)
   ▷ pronađi x u polju A = ⟨a_1, a_2, ..., a_n⟩
1  for i ← 1 to n
2        do if x = A[i]
3              then return i
4  return NIL
```

Složenost, tj. broj koraka, gornje procedure izravno ovisi o duljini niza kojeg pretražujemo. Na primjer, trebat će više vremena za pretraživanje niza

od 10000 elemenata, nego za pretraživanje niza od 10 elemenata. Primijetite da je broj operacija algoritma linearnog pretraživanja zapravo direktno proporcionalan broju elemenata u nizu. To se čini i prirodnim, samim time što pretražiti neki niz brojeva i znači pogledati svaki element niza i provjeriti je li jednak x ili nije.

Zadatak za vježbu 6.3 Implementirajte linearno pretraživanje u C++-u.

Međutim, pretraživanje niza se može drastično ubrzati ukoliko znamo nešto o samom nizu brojeva kojega pretražujemo, kao na primjer informacija da je niz brojeva kojega pretražujemo sortiran.

6.1.2.2 Binarno pretraživanje

Neka je zadan uzlazno sortirani niz brojeva $A = a_1 \leq a_2 \leq \ldots \leq a_n$, te broj x. Problem je utvrditi nalazi li se x u sortiranom polju A te, ukoliko se nalazi, vratiti njegovu poziciju.

Iskoristit ćemo informaciju da je niz brojeva sortiran i drastično ubrzati pretraživanje niza na sljedeći način:

K1. Uspoređujemo x sa srednjim elementom niza A.

K2. Ako je x manji od srednjeg elementa, nastavljamo tražiti x lijevo od njega.

K3. Ako je x veći od srednjeg elementa, nastavljamo tražiti x desno od njega.

K4. U suprotnom, vraćamo poziciju srednjeg elementa kao poziciju broja jednakog traženom x.

Primijetite kako upravo u koracima K2 i K3 koristimo informaciju da je niz brojeva uzlazno sortiran.

Još precizniji opis gore opisane procedure možete slijediti u sljedećem pseudokodu:

```
BinarySearch(A, x, p, r)
1  ▷ Algoritam u polju A traži element x od A[p] do A[r]
2  while p ≤ r
3        do m ← ⌊(p+r)/2⌋    ▷ nađi indeks sredine polja
4           if x = A[m]
5              then return m
6           if x > A[m]
7              then p ← m + 1
8              else  r ← m − 1
9  return NIL    ▷ vrati prazan indeks
```

Vidite li zašto je *binarno pretraživanje* drastično brža procedura od klasičnog linearnog pretraživanja? Odgovor na ovo pitanje se nalazi u činjenici da u svakoj iteraciji algoritma nastavljamo pretraživati na upola manjem nizu. Dakle, u svakoj se iteraciji početni niz zapravo prepolovi. Stoga je složenost binarnog pretraživanja proporcionalna broju raspolavljanja niza brojeva. Na primjer, ako je niz brojeva A duljine 8, broj raspolavljanja je 3. Drugim riječima, x u sortiranom nizu od 8 elemenata možete pronaći ili utvrditi da se ne nalazi u nizu u najviše 3 iteracije algoritma. Usporedbe radi, kod linearnog bi algoritma broj iteracija bio najviše 8. Općenito, x u nekom sortiranom nizu od n elemenata možete pronaći, ili utvrditi da se ne nalazi u nizu, u najviše $\lceil \log_2 n \rceil$ iteracija, dok će kod linearnog pretraživanja broj iteracija biti najviše n. Dakle, kako n raste, razlika broja koraka ovih dvaju algoritama postaje značajno izražena.

Zadatak za vježbu 6.4 Pravu moć binarnog pretraživanja možete i sami isprobati ako implementirate pseudokod u `C++`, te isprobate pretraživanje na nekom nizu duljine n.

Binarno pretraživanje na rekurzivan način. Pažljiviji je čitatelj možda već uočio kako se binarno pretraživanje u načelu može interpretirati i kao rekurzivni algoritam. U koraku kada element x uspoređujete sa srednjim elementom niza, ponavljate istu proceduru na lijevoj ili desnoj strani niza (vidi pseudokod), čime je ispunjen osnovni princip rekurzije.

```
RBinarySearch(A, x, p, r)
1  ▷ Algoritam u polju A rekurzivno traži element x od A[p] do A[r]
2  while p ≤ r
3        do m ← ⌊(p+r)/2⌋   ▷ nađi indeks sredine polja
4           if x = A[m]
5              then return m
6           if x < A[m]
7              then RBinarySearch(A, x, p, m − 1)
8              else RBinarySearch(A, x, m + 1, r)
9  return NIL   ▷ vrati prazan indeks
```

Rasprava oko složenosti *binarnog pretraživanja* u potpunosti ostaje ista i kada na rješenje gledate kao na rekurzivnu proceduru.

Zadatak za vježbu 6.5 Pokušajte objasniti zašto u slučaju binarnog pretraživanja problem možemo interpretirati kao iterativni algoritam, a opet i kao rekurzivni algoritam. Smatrate li da svaki iterativni algoritam možemo interpretirati rekurzivno? Vrijedi li možda i obrat?

Zadatak za vježbu 6.6 Implementirajte rekurzivno binarno pretraživanje u `C++`-u. Testirajte svoje implementacije za sekvencijalno pretraživanje, binarno pretraživanje i rekurzivno binarno pretraživanje na nizovima brojeva različite duljine n.

6.1.3 Problem sortiranja

Sljedeći primjer ilustrira važnost definiranja *uređenja* ulaznih podataka.

Primjer 6.4 *U rješavanju mnogih složenih problema ponekad je korisno ukoliko su podaci* dobro složeni*, uređeni, poredani po veličini, tj. sortirani.*

Uzmimo za primjer telefonski imenik. Budući da su imena složena abecednim redom, točno znamo kako naći nečiji telefonski broj. Zamislite kako bi smo to izveli da su podaci u imeniku "nabacani bez reda".

Kako bismo mogli govoriti o sortiranju, za svaki skup podataka moramo znati pripadajući uređaj, tj. za bilo koja dva podatka x i y znamo je li $x \leq y$ ili $y \leq x$.

Ovdje oznaka $\leq$ ima općenito značenje koje ovisi o vrsti (prirodi) podataka:

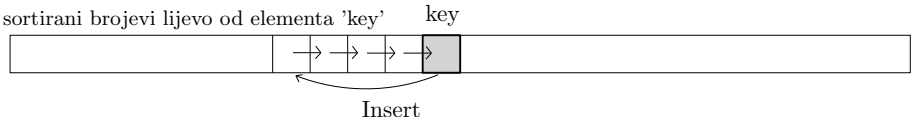
$$\begin{aligned} 3 &\leq 5 \\ \text{Babić} &\leq \text{Zrno} \quad \textit{(leksikografski uređaj)} \end{aligned}$$

Dakle, za dani niz elemenata $a_1, \ldots, a_n$, $n \in \mathbb{N}$ s definiranim uređajem '$\leq$' tražimo odgovarajuću permutaciju $a'_1, \ldots, a'_n$, takvu da vrijedi $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Sada je naš problem jasno definiran, pa ćemo pogledati dva različita algoritma za sortiranje, InsertionSort i MergeSort algoritam.

6.1.3.1 InsertionSort algoritam

Ovaj algoritam prolazi elementima niza s lijeva na desno i sortira na način da su svi elementi lijevo od elementa kojega se trenutno procesuira sortirani, a sam se element kojega se procesuira „umetne" (eng. Insert; otuda i ime algoritma "InsertionSort", tj. algoritam umetanjem) na pravu poziciju (vidi sliku 6.1).



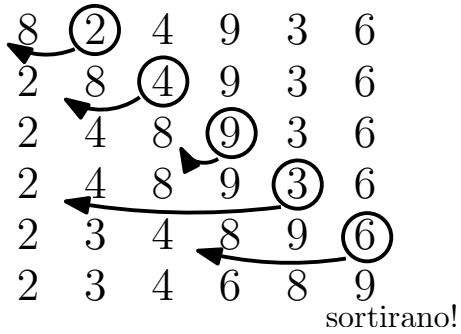
Slika 6.1: Element kojega procesuiramo nazvali smo KEY. Svi su elementi lijevo od njega već procesuirani i sortirani. U trenutku kad procesuiramo KEY element, potrebno ga je umetnuti (Insert) na odgovarajuće mjesto, te odgovarajuće elemente pomaknuti za jedno mjesto udesno. Na taj način završava jedna iteracija algoritma i procesuira se sljedeći element u nizu, kojega se sad proglašava KEY elementom.

Na primjeru prikazanom na slici 6.2 pokazani su koraci InsertionSort algoritama na konkretnom primjeru od šest brojeva.

Sami koraci algoritma su precizno definirani sljedećim pseudokodom:

InsertionSort(A)

```
1  ▷ sortiraj polje A.
2  for j ← 2 to length(A)
3       do key ← A[j]
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```



Slika 6.2: Element kojega procesuiramo, KEY element, označen je kružićem. Primijetite da su elementi desno od KEY elementa uvijek sortirani.

Složenost, tj. broj operacija INSERTIONSORT algoritma, ovisi o tome kakav niz elemenata sortiramo. Na primjer, ako sortiramo već sortirani niz elemenata, sama se procedura sortiranja svodi na jednostavan prolazak kroz polje s lijeva na desno. U tom je slučaju i broj operacija direktno proporcionalan samoj duljini polja, te je algoritam veoma brz. Međutim, ako pomoću INSERTIONSORT algoritma pokušate sortirati niz elemenata koji je dan kao obrnuto sortiran, tj. sortiran od najvećega prema najmanjem, tada će gornja procedura svaki INSERT „platiti" proporcionalno broju elemenata koji se nalaze s lijeve strane KEY elementa. To je zapravo vrlo loš scenarij za INSERTIONSORT algoritam. Stoga se ovakav algoritam najčešće u praksi ne primjenjuje, a posebice ne u slučajevima kada ne znate ništa o potencijalnom nizu elemenata kojega treba sortirati.

Zadatak za vježbu 6.7 Implementirajte u C++-u InsertionSort algoritam. Testirajte svoju implementaciju na različitim inputima različite veličine.

6.1.3.2 Rekurzivno sortiranje – MERGESORT algoritam

Pametniji način sortiranja na općenitim podacima (podacima o kojima nemamo nikakvo predznanje poput: „podaci su već gotovo sortirani") je rekurzivni algoritam MERGESORT. Osnovna tri koraka tog rekurzivnog algoritma mogu se opisati u sljedećem:

a) Podijelite niz od n podataka u 2 podniza veličine $n/2$.

b) *Rekurzivno* sortirajte oba podniza veličine $n/2$.

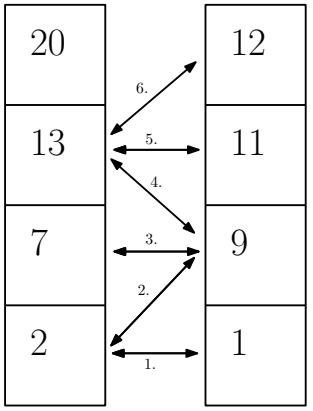
c) Dva rekurzivno sortirana podniza spojite (MERGE) u sortirani niz.

Pseudokod gore opisanih koraka je sljedeći:

```
MergeSort(A, p, r)
1  if p < r
2     then q ← ⌊(p+r)/2⌋
3          MergeSort(A, p, q)
4          MergeSort(A, q + 1, r)
5          Merge(A, p, q, r)
```

MERGESORT algoritam ime je dobio upravo zbog MERGE koraka, tj. koraka u kojemu se dva sortirana podniza spajaju u jedinstven sortiran niz. Primijetite da je taj korak najsloženiji korak koji je zapravo zaslužan za samo sortiranje elemenata niza. Drugim riječima, ako shvatimo kako efikasno spojiti dva sortirana podniza u jedan sortirani niz shvatili smo MERGESORT algoritam.

Promotrite na sljedećem primjeru dva sortirana podniza, oba duljine 4. Cilj je spojiti ih u jedan sortirani niz duljine 8.



Redosljed uspoređivanja dvaju elemenata u gornjem primjeru određen je rednim brojem iznad strelice. Drugim riječima, najprije uspoređujemo 1 i 2, te 1 zapisujemo u novi niz. Nakon toga uspoređujemo 2 i 9, te zapisujemo 2 u novi niz, itd. Nakon tog postupka stvorit ćemo novi sortirani niz duljine 8.

Ovim smo primjerom upravo objasnili na koji način MERGE procedura spaja dva sortirana podniza u jedinstveni sortirani niz. Što se tiče složenosti samoga

postupka, primijetite da broj operacija direktno ovisi o broju elemenata u oba podniza (budući ste u svakom koraku zapisali jedan element u novi niz, te ga nakon toga više nikada niste procesuirali). MERGE procedura je tako direktno proporcionalna ukupnom broju elemenata u oba podniza.

U sljedećem ćemo pseudokodu formalizirati korake MERGE procedure:

```
MERGE(A, p, q, r)
 1  n_1 ← q − p + 1
 2  n_2 ← r − q
 3  ▷ stvori pomoćna polja L[1..n_1 + 1] and L[1..n_2 + 1]
 4  for i ← 1 to n_1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7        do R[j] ← A[q + j]
 8  L[n_1 + 1] ← ∞
 9  L[n_2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else  A[k] ← R[j]
17                    j ← j + 1
```

Sada imate sve elemente potrebne za sljedeći zadatak.

Zadatak za vježbu 6.8 Napravite implementaciju MERGESORT algoritma u C++-u.

6.1.4 Množenje matrica

Neka su zadane kvadratne matrice $A = [a_{ij}]$ i $B = [b_{ij}]$, $i = j = 1, \ldots, n$. Želimo izračunati matricu $C = [c_{ij}] = A \cdot B$. Svaki element c_{ij} matrice C računamo iz izraza:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \text{ za svaki } i, j = 1, \ldots, n.$$

Dakle, složenost računanja jednog elementa matrice C direktno je proporcionalna broju n. Budući se matrica C sastoji od točno n^2 različitih elemenata, složenost računanja cijele matrice $C = A \cdot B$ bit će direktno proporcionalna broju n^3.

Precizan pseudokod množenja dviju kvadratnih matrica dan je u sljedećem:

```
MATRIX-MULTIPLY(A,B)
1  ▷ pretpostavka: A;B dimenzije n × n, alociraj matricu C
2  n ← rows[A]
3  for i ← 1 to n
4        do for j ← 1 to n
5              do for k ← 1 to n
6                    do C[i][j] = C[i][j] + A[i][k] · B[k][j]
7  return C ▷ korak koji se može implementirati primjenom pokazivača (C++)
```

Bitno je primijetiti da ukoliko je broj operacija nekog algoritma reda veličine n^3 tj. proporcionalan n^3, tada već govorimo o algoritmu koji zahtijeva mnogo operacija za računanje rješenja. Na primjer, ako želimo pomnožiti dvije matrice dimenzije $n = 1000$, broj operacija koje algoritam izvodi biti će proporcionalan broju $1000 \times 1000 \times 1000 = 10^9$.

Zadatak za vježbu 6.9 Implementirajte množenje matrica na gore opisani način u `C++`-u i provjerite koliko će vam vremena trebati za računanje matrice C za $n = 1000$.

6.1.4.1 Rekurzivno množenje matrica

Pokušat ćemo ubrzati množenje kvadratnih matrica tako što ćemo matrice pomnožiti rekurzivno. Prvi problem kod rekurzivnog razmišljanja je rastavljanje početnog problema, u našemu slučaju matrica, na manje podprobleme koje rekurzivno rješavamo. Postavlja se pitanje kako podijeliti matricu?

Osnovna je ideja matricu tipa $n \times n$ podijeliti u 2×2 blok-matrice tipa $n/2 \times n/2$. Bez smanjenja općenitosti pretpostavimo neka je n potencija broja 2, tako da stvaranje blok-matrica nije problem[4].

Dakle, možemo pisati:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},\ B = \begin{pmatrix} e & f \\ g & h \end{pmatrix},\ C = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

[4] Ako n nije potencija broja dva, matrice uvijek možete nadopuniti s nulama tako da nova, nadopunjena, matrica ima broj redaka i stupaca koji je potencija broja dva.

gdje su $a, b, c, d, e, f, g, h, r, s, t, u$ odgovarajuće $n/2 \times n/2$ blok-matrice, tako da je

$$C = \begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Lako je provjeriti da vrijedi:

$$\begin{aligned} r &= a \cdot e + b \cdot g \\ s &= a \cdot f + b \cdot h \\ t &= c \cdot e + d \cdot g \\ u &= c \cdot f + d \cdot h. \end{aligned}$$

Iz gornjih jednakosti slijedi da se umnožak dviju $n \times n$ matrica zapravo svodi na 8 **rekurzivnih** množenja te 4 zbrajanja $n/2 \times n/2$ matrica. Nažalost, može se pokazati kako rekurzivan postupak poput ovog gore i dalje u sebi sadrži broj operacija proporcionalan n^3. Dakle, u ovom se slučaju rekurzija nije isplatila. Dapače, ovakvo će rješenje u praksi raditi i sporije od klasičnoga množenja.

Zadatak za vježbu 6.10 Napišite pseudokod rekurzivnog množenja matrica.

Zadatak za vježbu 6.11 Implementirajte rekurzivno množenje kvadratnih matrica u `C++`-u i usporedite vrijeme potrebno za množenje matrica veličine $n = 100$ rekurzivno, nasuprot klasičnom množenju.

6.1.4.2 Strassenov algoritam

Njemački je matematičar Volker Strassen iskoristio činjenicu da je zbrajanje dviju matrica mnogo manje zahtijevan posao od množenja dviju matrica. Osnovna ideja Strassenova algoritma je smanjiti broj rekurzivnih množenja na 7, ali na uštrb zbrajanja.

U tu je svrhu definirao:

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

te primijetio nevjerojatnu poveznicu gornjih izraza s blok-matricama r, s, t i u:

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7.
\end{aligned}
$$

Uočite da blok-matrice r, s, t i u sada računamo pomoću samo 7 rekurzivnih množenja.

Zadatak za vježbu 6.12 Verificirajte istinitost gornjih izraza za r, s, t i u. Imajte na umu da množenje matrica nije komutativno.

Strassenov se algoritam može sažeti u sljedećim koracima:

a) Podijelite A i B u 2×2 blok-matrice tipa $n/2\times n/2$. Formirajte $P_1, \ldots, P_7$.

b) Rekurzivno izvrši 7 množenja $n/2 \times n/2$ matrica.

c) Formiraj C pomoću zbrajanja i oduzimanja $n/2 \times n/2$ matrica.

Zadatak za vježbu 6.13 Napišite pseudokod Strassenova algoritma.

Može se pokazati da je broj operacija Strassenova algoritma proporcionalan $n^{2.81}$. Načelno, ovdje se radi o najbržem algoritmu od svih koje smo do sada opisali. U praksi je Strassenov algoritam na današnjim strojevima uvijek brži od klasičnoga množenja matrice već za $n > 30$.

Zadatak za vježbu 6.14 Implementirajte Strassenov algoritam u `C++`-u. Usporedite svoju implementaciju i s klasičnim načinom množenja matrica. Uvjerite se i sami da za dovoljno velike n (npr. $n > 30$) Strassenov algoritam postaje brži od klasičnoga pristupa.

Literatura

[1] L. Budin, *Informatika za 1. razred gimnazije*, Element, 2000.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, MIT Press, 2009.

[3] J. Glenn Brookshear, *Computer Science: An Overview (8th Edition)*, Addison Wesley, 2004.

[4] John Kopplin, *An Illustrated History of Computers*, http://www.computersciencelab.com/ComputerHistory/History.htm

[5] R. Mata-Toledo, P. Cushman, *Schaum's outline of Introduction to Computer Science*, McGrow-Hill, New York, 2000.

[6] K. Melhorn, *Efficient data structures and algorithms, 3Ed* , Springer, 2003.

[7] R. Scitovski, *Numerička matematika*, Sveučilište u Osijeku, 2005.

[8] J. Šribar, B. Motik, *Demistificirani C++*, Element, Zagreb, 2001.