Deo I

Algoritmi i strukture podataka

Kompjuterska memorija je linearna. Centralni procesor može da pristupi samo jednom podatku u jednom trenutku. Podaci se često unose uživo, u toku izvršavanja aplikacije. Redom uneti podaci ne moraju zauzimati susedna mesta u memoriji.

Način smeštanja podataka u kompjuterskoj memoriji je **struktura podataka**. Ona može biti vrlo složena. Jedni te isti podaci se u kompjuterskoj memoriji mogu zapisati na više načina. U zavisnosti od načina kako su podaci zapisani se **bira algoritam** koji će biti najefikasniji za rešavanje problema.

Kompjuteri obično imaju dve vrste memorijskih medija: brže i sporije. Od vrste medija na kome se nalaze podaci za obradu takođe zavisi izbor algoritma i strukture podataka.

Za neke probleme je algoritam koji ih rešava jednostavan i postoje procesori koji su u mogućnosti da ga brzo izvrše nad velikom grupom susednih polja u memoriji. U tom slučaju se **bira struktura podataka** koja će se smestiti u susedna polja memorije.

Zaključak je da se izučavanje algoritama ne može posmatrati odvojeno od strukture podataka nad kojom se algoritam primenjuje.

Glava 1

Algoritmi i kompjuterski programi

1.1 Kompjuterski programi

Ako su ulaz i izlaz algoritma podaci koji se mogu predstaviti na kompjuteru, onda se algoritam može pretvoriti u **kompjuterski program**.

Kompjuter izvršava program napisan na **mašinskom jeziku**. Program se sastoji iz niza instrukcija i podataka koji procesoru zadaju naredbe za izvršavanje operacije nad podacima. Instrukcije kao i podaci su kodirani brojevima zapisanim u binarnom sistemu, koriste zajedničko memorijsko polje i izvršavaju se u centralnom procesoru: CPU[1].

Ulazni i izlazni uređaji, memorija, CPU, periferije (tastatura, monitor,...) su svi spojeni na jednu **magistralu**. Tako da CPU može pristupiti i periferijama. U današnjim kompjuterima ta magistrala je matična ploča[2], a ideja ovakve kompjuterske arhitekture je potekla od Fon Nojmana[3].

1.1.1 Istorija

Potreba za mašinom koja će izvršavati složene zadatke i rešavati probleme je prvo realizovana kao imaginarni računarski uređaj. Tako je Alan Tjuring[4] 1936. godine formulisao **Tjuringovu mašinu**.

Tjuringova mašina je matematički model koji definiše apstraktnu mašinu za manipulisanje simbolima na beskonačnoj traci u skladu sa tabelom pravila. Iako je to veoma jednostavan model, za bilo koji kompjuterski algoritam može se konstruisati Tjuringova mašina sposobna da implementira njegovu logiku.

Iz praktičnih potreba tehnološkog razvoja, radi sprovođenja složenih radnji nad podacima razvijane su mašine koje danas nazivamo **kompjuteri**.

[1] Central Processing Unit - Centralni procesor, EN
[2] Motherboard, EN
[3] Von Neumann, 1903–1957, matematičar, fizičar, informatičar, inženjer i polimata
[4] Alan Turing 1912 – 1954, engleski matematičar, informatičar, logičar, kriptoanalitičar, filozof, teorijski biolog

1.1.2 Dalja istorija

Još tridesetih godina dvadesetog veka kreirana je mehanička mašina za šifrovanje poruka **Enigma**, a početkom četrdesetih i **Bombe** za dešifrovanjene tako kriptovanih poruka. I Enigma i Bombe su igrale značajnu ulogu u Drugom svetskom ratu. Ipak, to su bile mašine kreirane da izvršavaju samo jedan algoritam - nisu bile programabilne.

Prvi kompjuteri su upotrebljavani za rešavanje složenih matematičkih proračuna.

Pre pojave kompjutera kakve sada znamo, složene proračune u prvoj polovini dvadesetog veka izvršavali su specijalno obučeni ljudi koje su zvali **kompjuteri** (EN, Computer).

Naziv "**kompjuter**" za programabilnu mašinu koja izvršava neki isprogramirani algoritam je u opštoj upotrebi tek od kraja pedesetih godina dvadesetog veka.

U Nemačkoj je 1941. godine Konrad Cuze[5] pustio u rad prvi programabilni kompjuter Z3, zasnovan na relejnim kolima.

Vremenom su elektromehanički releji prevaziđeni upotrebom električnih flip-flopova. Za potrebe američke mornarice Fon Nojman je dao nacrte kako da se naprave elektronski kompjuteri.

Prvi elektronski programabilni kompjuteri zasnovani na Fon Nojmanovskoj arhitekturi su se pojavili krajem četrdesetih godina dvadesetog veka. Njih su na mašinskom jeziku programirali obučeni tehničari i tako je nastala profesija **programera**.

Arhitektura kompjutera je od elektronskih mašina sa kablovima i lampama vremenom prešla na tranzistorsku tehnologiju. Vremenom su tranzistorska kola integrisana u integralne čipove koji su vremenom dobijali sve veći stepen integracije i nove poluprovodničke tehnologije.

Uporedo kako je kompjuterska arhitektura postajala moćnija, programi su postajali duži i složeniji. Mali broj programera je mogao da piše, prepravlja i kontroliše programe pisane na mašinskom jeziku.

Prvi korak ka olakšanju programiranja je bio da se brojevima, koji predstavljaju instrukcije, dodele logički nazivi (LOAD, ADD, ...) i tako je nastao zapis kompjuterskog programa koji se zove **asemblerski kod**. Za zapis brojeva se počeo koristiti **heksadekadni sistem** a za veličinu reči je odabran jedan **bajt** = 8 bitova[6]. Instrukcije pisane u asembleru su se "jedan na jedan" pretvarale u instrukcije mašinskog jezika odgovarajućeg procesora. Retko je bilo da program pisan u asembleru za jedan procesor radi na drugom procesoru. Za prevođenje ispravno napisanog asemblerskog koda na mašinski jezik su korišteni kompjuterski programi - asemblerski prevodioci, kraće **asembleri**.

Godine 1954. Džon Bejkus[7] je sa timom u IBMu ustanovio novi **programski jezik**, alternativu asembleru, za prevođenje na mašinski jezik algoritama za izračunavanje formula. Bejkus je rekao da je izumeo programski jezik i napisao program za prevođenje na mašinski jezik zato što je bio lenj da piše petlje za računanje formula u asembleru.

Tako je nastao prvi programski jezik široke primene, **FORTRAN**[8]. U Fortranu, formula ili programska petlja se zapisuje na poseban način, koji programerima omogućava lakše programiranje

[5] Konrad Zuse, 1910 — 1995, inženjer, kompjuterski naučnik, pronalazač i biznismen

[6] 1 byte = 8 bits; 1 bit = 0/1 informacija = netačno/tačno

[7] John Backus, 1924 - 2007, matematičar, informatičar

[8] FORTRAN - Formula translating system - Sistem za prevođenje formula

i kontrolu programa, a koji ćemo zvati **programski kod** ili samo **kod**.

Bilo je puno verzija FORTRANA, a 1961. godine je ustanovljena verzija FORTRAN IV. Njen značaj je bio u nezavisnosti od arhitekture kompjutera. To je omogućilo da se programiranje uči i bez fizičkog pristupa kompjuteru, a da se stečeno znanje može primeniti na svakom kompjuteru. Verzije Fortrana su se nizale godinama, a i dan danas se koriste programi pisani u verziji FORTRAN 77. Instrukcije Fortrana su slične govornom jeziku, a prevođenje programa napisanog u Fortranu na asembler ili mašinski jezik rade programi koji se zovu **kompajleri** (compilers) ili prevodioci.

1.1.3 Bliža istorija

Vremenom je razvijeno puno programskih jezika od kojih mnogi imaju namenu u posebnoj oblasti. **Programski jezik C** (čita se "si") je proceduralni programski jezik opšte namene. C je stvoren sa ciljem da se omogući efikasno prevođenje na mašinski jezik. C je najviše upotrebljavan u aplikacijama prethodno pisanim u asembleru. To su prvenstveno operativni sistemi ali i razni aplikativni softveri. Danas se C upotrebljava svuda - od ugrađenih(embedded) komponenti do superkompjutera.

Godine 1990. ISO[9] i IEC[10] su zajedno usvojili standard **ANSI C** kao međunarodni standard pod imenom C90. Taj standard je dugo zaživeo uz male izmene i omogućio kompatibilnost C programa na raznim platformama.

Izmene su bile u verzijama C95 (kada je prihvaćen standard za floating point[11] predstavljanje brojeva IEEE[12] 754 iz 1985. godine), C99, C11 i sada važeći C17. Cifre u nazivu standarda (otprilike) kazuju koje godine je usvojen.

Standard IEEE 754 je ugrađen u procesore svih kompjutera u zadnje dve decenije. Doživeo je male izmene, ali je omogućio razmenu podataka između programa pisanih u raznim programskim jezicima i razmenu između različitih kompjutera i operativnih sistema.

Standardizacija je doprinela velikom razvoju kompjutera i napretku softverske industrije, razmeni podataka, razvoju Open source[13] pokreta.

Preokret u kompjuterskoj industriji je napravljen početkom osamdesetih godina dvadesetog veka. Tada je kompanija IBM počela proizvoditi **IMB PC**[14] kompjutere otvorene arhitekture. Iako je arhitektura bila otvorena - mogli su svi koji se drže standarda kompatibilnosti proizvoditi IBM PC kompatibilne računare i komponente - najviše je zaradila kompanija IBM.

Glavni preokret je u tome što se sa višekorisničkih kompjutera sa multitasking[15] operativnim sistemom prešlo na personalne kompjutere sa jednokorisničkim operativnim sistemom koji nije bio multitasking. U početku se nije verovalo da će industrija PC kompjutera toliko porasti, odnosno, da će velik broj ljudi moći sebi da priušti lični kompjuter. Godine 1984. kompanija Apple

[9] International Standards Organization - Međunarodna organizacije za standarde
[10] International Electrotechnical Commission - Međunarodna elektrotehnička komisija
[11] pokretan zarez, EN - predstavljanje brojeva u verziji mantisa + eksponent
[12] IEEE - Institute of Electrical and Electronics Engineers, čita se "aj tripl i"
[13] Otvoreni kod, EN - besplatna dostupnost koda programa
[14] PC - čita se "pi si" - Personal Computer, EN - personalni (lični) kompjuter
[15] multitasking - više procesa radi istovremeno, EN

je lansirala **Macintosh** i započela borbu za tržište personalnih računara. Apple je skoro bankrotirao 1986. da bi se krajem devedesetih godina vratio. Danas kompanije IBM i Apple sarađuju na raznim projektima.

Pisanje operativnog sistema za IBM PC: PC DOS (kasnije **MS DOS**) je prepušteno kompaniji **Microsoft**. Operativni sistem je bio jednostavan, jednokorisnički, a doveo je do revolucije proizvodnje aplikativnih programa. PC kompatibilci su vrlo brzo po ceni postali dostupni velikom broju ljudi i zavladali tržištem kompjutera, a MS DOS je postao najpopularniji operativni sistem.

1.1.4 Sadašnjost

Novi preokret je bio u softveru, počeo je 1990. godine, kada je Linus Torvalds[16] podelio sa svetom besplatni operativni sistem zasnovan na Unixu za PC kompatibilne računare. Ubrzo je taj sistem dobio ime **Linux**, stavljen u open source i dobio licencu **GPL**[17] projekta GNU.

Tako je započela utrka open source i komercijalnih programa. Mnogi tvorci besplatnog sofvera su počeli da zarađuju na uslugama, a neke velike korporacije (na pr. Microsoft) su počele da puštaju svoje programe u open source. Danas je najveći broj produktivnih servera na Linux operativnom sistemu, koji je višekorisnički i multitasking.

U dvadesetom veku povezivanje dva kompjutera, a zatim umrežavanje više kompjutera je počelo već sedamdesetih godina. Osamdesetih godina su funkcionisale velike komercijalne kompjuterske mreže u Americi i Evropi. **TCP/IP** protokol za razmenu podataka u mreži je nastao sredinom osamdesetih godina a 1989. godine je pušten u Open source. ISP[18] su postojali već 1990. godine.

Postepeno su gašene druge mreže, a sredinom devedesetih godina **Internet**, zasnovan na TCP/IP protokolu, je zavladao svetom, promenio kulturu i doveo do velikog porasta broja kompjutera i programa koji su radili kroz Internet mrežu. Sa CERNa[19] je potekao informacioni sistem WWW[20] dostupan sa bilo kog čvora na mreži.

Vremenom su programiranje velikih programa, timski rad, modularnost, dokumentovanje, web programiranje, klijent-server arhitektura, veliki broj biblioteka, . . . , postavili nove zadatke za programere i programske jezike. Rešenje se pojavilo u **objektno orijentisanom programiranju**.

Internet je omogućio ljudima širom sveta da se povezuju i rade zajedno na internacionalnim projektima. Kompjuteri su ušli u telefone, televizore, automobile, . . . Softverska industrija se razvila, pojavila su se nova zanimanja programera koji su specijalizovani samo za određene delove programa. Pojavom Pandemije COVID-19, koristeći Internet, mnogi poslovi su prešli na rad od kuće, neki bez izgleda da se vrate u kancelarije.

1.1.5 Iz programskog jezika u mašinski jezik

Ono što se danas dešava u sofverskoj industriji je imalo začetke još pre nekoliko decenija. Tako je osamdesetih godina dvadesetog veka nastao **C++** kao objektno orijentisano proširenje program-

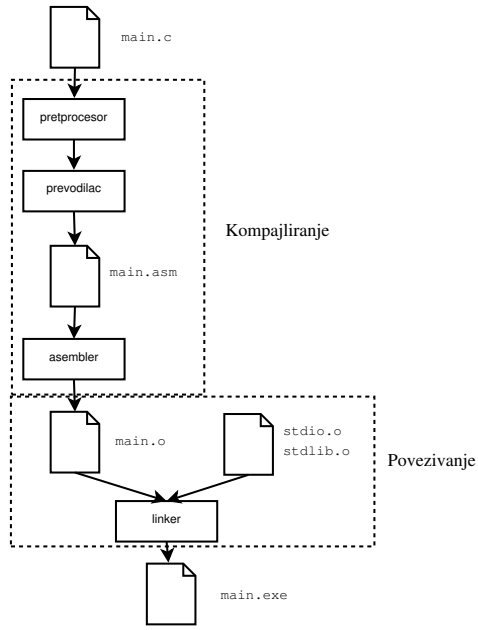
[16] 1969 - , Finsko-američki softverski inženjer, tvorac Linux kernela
[17] General Public License
[18] Internet Service Provider - Provajderi uslužnog povezivanja kompjutera u mrežu TCP/IP protokolom
[19] CERN - Conseil européen pour la recherche nucléaire, FR - evropska istraživačka laboratora za fiziku čestica, Ženeva
[20] World Wide Web - svetska informaciona mreža za povezivanje hipertekst dokumenata

skog jezika C. Standard C99 je pokušao zadržati da veliki broj programa napisanih u C-u može da se kompajlira kao C++ i obrnuto.

Ipak, danas nije C podskup C++ ni obrnuto. Osim C++, pojavili su se mnogi objektno orijentisani programski jezici. Objektno orijentisano programiranje je postalo standard u softverskoj industriji.

Uobičajeni tok pisanja programa u C-u je da se napiše glavni program (sa kojim će početi izvršavanje), recimo `main.c`. Potom se **kompajlira** u dva prolaza (preprocesiranje i prevođenje) u asemblerski kod. Kad se kompajlira na arhitekturi na kojoj će se izvršavati, obično se automatski asemblerski kod pretvara u objektni kod za odgovarajuću procesorsku arhitekturu.

Kreiranje izvršnog koda se završava *povezivanjem*[a]. Taj posao obavlja **linker** spajajući objektni kod sa sistemskim bibliotekama i, eventualno, drugim bibliotekama koje smo kompajlirali ili nabavili u objektnom kodu. Često se te bibilioteke ne ubacuju direktno u izvršni mašinski kod, već se učitavaju po potrebi pri izvšavanju programa - dinamički.



Slika 1.1: Prevođenje C u mašinski program

[a] povezivati - to link, EN

Na sličan način se postupa sa programima u drugim programskim jezicima. Vlasnik izvornog koda[21] može da proda izvršnu verziju programa, odnosno aplikacije, u mašinskom jeziku za odgovarajuću kompjutersku arhitekturu i operativni sistem. Izvršnu verziju je skoro nemoguće inverznim inženjeringom vratiti u izvorni oblik. Na taj način tehnologija u obliku izvornog koda ostaje u vlasništvu u obliku softvera izvornog koda.

Često se vlasnici izvornog koda potrude da ga prilagode za više platformi koje zavise od mašine - hardvera (hardware) i operativnog sistema. Tu se nailazi na **probleme kompatibilnosti sotvera** sa hardverskim platformama i operativnim sistemima.

Za programski jezik C postoje standardne biblioteke koje omogućavaju pristup funkcijama i procedurama operativnog sistema za vezu sa hardverom. Linker povezuje kompajliran program sa standardnim bibliotekama (dinamički) koje često pripadaju operativnom sistemu. Zajedno sa unapređivanjem i ažuriranjem operativnog sistema, ažuriraju se i standardne biblioteke. Stari programi nekad zahtevaju da se naslede biblioteke iz starije verzije operativnog sistema (legacy, EN).

Precizno uputstvo o upotrebi standardnih C biblioteka je definisano u API[22], a prototipovi C funkcija su u header fajlovima koji se uključe na početku programa pretprocesorskom direktivom #include.

Za nove verzije Windows operativnog sistema postoji Visual C++ kompajler kojeg besplatno uz

[21] source code - izvorni kod, čita se sors kod, EN

[22] Application programming interface - Interferjs za pristup procedurama iz aplikacije (EN)

operativni sistem održava Microsoft. Njime se kompajliraju i povezuju sa Microsoftovom standardnom C bibliotekom programi u C i C++ programskom jeziku.

Gnu C kompajler (GCC) takođe kompajlira i povezuje C i C++ programe. On je stadndardni kompajler koji se koristi na Linux operativnim sistemima. Godine 2005. do 2010. od GCC se odvojila minimalistička Gnu verzija za Windowse (Mingw-w64).

Mingw-w64 uključuje GNU Compiler Collection (GCC), GNU Binutils za Windovs (assembler, linker, arhivski menadžer), skup header datoteka koje se slobodno distribuiraju za Windovs i statičke biblioteke za linkovanja, koje omogućavaju korišćenje Windows API, GNU Debugger i razne uslužne programe.

1.1.6 Iz programskog jezika na izvršavanje programa

Kompjuteri su vremenom postali moćni, sa velikim količinama memorije i velikim brzinama procesora. Pojavili su se programski jezici koji se ne prevode na mašinski jezik, već se interpretiraju. To znači da se učitava linija po linija koda u program **interpreter** i tim redom izvršava. Linije programskog koda se mogu pisati ručno, a mogu se snimiti u tekstualnu datoteku (fajl) i pustiti da se izvršavaju sve redom.

Ovako pisani i izvršavani programi ne postižu brzinu koju postižu oni koji su prevedeni na mašinski jezik. Ali, imaju niz prednosti u razvoju, otkrivanju grešaka, pristup trenutno raspoloživim resursima kompjutera i bolju interakciju sa korisnikom.

Postoje programski jezici koji se kompajliraju na **jezik virtuelne mašine**. Na primer, JVM[23] je virtuelna mašina na kojoj se izvršavaju programi kompajlirani u Java "mašinski" kod. Osim programskog jezika **Java**, postoji niz programskih jezika koji se prevode u kod izvršiv na JVM.

Prednost JVM je što postoji za sve popularne kompjutere i operativne sisteme, i što programi pri izvršavanju na JVM imaju isti izgled i rezultat na svim kompjuterima. JVM omogućava prevazilaženje zavisnosti softvera od hardverske platforme i operativnog sistema.

Postoji i **JavaScript**, programski jezik velike moći koji se interpretira u internet pretraživaču (web browseru). Podržavaju ga svi novi pretraživači. Sa razvojem novih JavaScript verzija, pojavljuju se i nove verzije internet pretraživača. Zajedno sa CSS[24] i HTML[25] JavaScript danas čini platformu na kojoj je postavljena većina sadržaja WWW mreže.

Programski jezici Java i JavaScript su preuzeli veliki deo sintakse iz C-a. Njihove sintakse su slične, ali su to, ipak, dva različita jezika.

Obično se JavaScript, CSS i HTML koriste za programiranje **frontenda** (ono što korisnici vide), Java za **backend**, pozadina, koja daje logiku povezivanja sa SQL[26] **bazom podataka** (osnova).

Postoji SQL programski jezik koji je u velikoj meri kompatibilan sa najpoznatijim relacionim bazama podataka. Njegove komande se interpretiraju pomoću klijentskih programa. Uobičajena je **klijent - server** komunikacija koja omogućava povezivanje baze sa drugim komponentama softvera.

[23] Java Virtual Machine - Java virtuelna mašina, kompjuterski program koji izvršava Java mašinski kod
[24] Cascading Style Sheets - jezik stilova za opisivanje prezentacije
[25] HyperText Markup Language - standardni jezik za označavanje - koristi se za prikazivanje u web browseru
[26] Structured Query Language - strukturirani jezik upita za baze podataka

1.1.7 Izvorni kod

Programi, bilo za prevođenje na mašinski jezik, bilo za interpretiranje, se smeštaju u tekstualnu datoteku.

Sredinom šezdesetih godina dvadesetog veka je zaživeo 7-bitni **ASCII**[27] standard. On podržava engleski jezik. U to vreme je, takođe, zaživeo standard 8-bitnih procesorskih registara i memorijske jedinice **bajt** (byte, EN) od 8 bita. Osmi bit se neko vreme koristio za proveru parnosti zbog mogućih grešaka u komunikaciji. Tako je došlo do poistovećivanja jednog karaktera sa jednim bajtom.

Danas skoro svi programski jezici, bilo da se kompajliraju ili interpretiraju, imaju komande na engleskom jeziku koje mogu da se zapišu u ASCII standardu.

Većina svetskih jezika ima pisma čija slova pravazilaze ASCII standard. U početku je to rešavano uvođenjem kodnih strana i ubacivanjem potrebnih karaktere u drugu polovinu adresnog prostora jednog bajta: $2^8 = 256 = 2 \cdot 2^7$. Tu su se, između ostalih, našli ISO 8859 Latin 1 i Latin 2 standardi za pisma evropskih jezika, i Microsoftove kodne strane.

Naravno da 256 različitih simbola nije dovoljno za pisma svih jezika, a pristup kodnih strana nije omogućavao mešanje više pisama u istom fajlu.

U dvadesetom veku, krajem osamdesetih i početkom devedesetih, osmišljen je standard Unicode i osnovan konzorcijum za praćenje i dodavanje novih jezika u standard. Unicode konzorcijum se bavi formiranje pravila za kodiranje teksta bilo kog jezika: **UTF**[28]. Takođe se bavi pravilima za redosled simbola, održavanju sistema za transformisanje tekstualnog fajla u slova svakog pisma iz standarda.

Nastao je **UCS**[29] koji je usklađen sa ISO standardom, očuvao ASCII kompatibilnost i dobio mogućnost zapisivanja istog jezika u više pisama sa više redosleda slova.

Najviše su u upotrebi UTF-8 i UTF-16 standardi. Danas je preko 98% sadržaja svih web stranica kodirano **UTF-8** standardom. UTF-8 može da kodira 1.112.064 simbola u Unicode-u koristeći jedan do četiri bajta. Svi tekstovi pisani u ASCII standardu (jedan bajt) ostaju kompatibilni sa UTF-8.

Jedna "mana" UTF-8 standarda je što se za zapisivanje jednog slova pisma koristi jedan do četiri bajta, pa se na osnovu dužine fajla ne može prebrojati broj slova u tekstu. Isto, leksikografsko sortiranje ne može da se vrši bajt na bajt (vidi poglavlje 3.6.2). Postoji mogućnost za predstavljanje vizuelno i sintaksno istih leksičkih simbola na više načina. Recimo, u našem jeziku, latinično i ćirilično a. Kombinacije dva ili tri karaktera koje čine jedan glas (digrafi i trigrafi) u mnogim jezicima menjaju ASCII leksikografski redosled reči predstavljenih stringovima. Unicode konzorcijum vodi računa o pravilima za poređenje leksičkog prethođenja, o pravilnom zapisivanju brojeva, cena i sličnom, za registrovane jezike.

Svi moderni i modernizovani programski jezici (C, C++, ...) podržavaju UTF-8 kao standard za pisanje izvornog koda programa. Operativni sistemi prihvataju UTF-8 standard za imena fajlova, imena korisnika, ...

[27] American Standard Code For Information Interchange, standard za kodiranje karaktera, 1963.
[28] Unicode Transformation Format - Pravila za prevođenje karaktera u UCS
[29] Universal Coded Character Set - Univerzalni kodni standard

GCC kompajler koristi UTF-8 kao interni kod. To znači da stringovi, koji mogu predstavljati i imena fajlova, i komentari mogu da se pišu u UTF-8. Standardna C biblioteka GCC omogućava lokalizaciju izlaza u skladu sa Unicode standardom.

Standardna C biblioteka Microsoft Windows omogućava posebnu lokalizaciju za svaki thread[30].

U računarskoj nauci thread je najmanja sekvenca programiranih instrukcija kojom može nezavisno upravljati operativni sistem. U većini slučajeva thread je komponenta procesa.

Ovakvo ponašanje standardne C biblioteke ne proizvodi grešku pri kompajliranju, ali daje drugačije ponašanje od Mingw-64 kompajlirinih programa na Windowsima. Poglavlje 3.6.2.

[30] thread = konac, nit, EN

Glava 2

Programiranje

Naslov ove glave zvuči pretenciozno. Ne može se čitajući ova glava naučiti programiranje. Niti se izlaganje može pratiti ako nemate barem malo iskustva sa programiranjem kompjutera.

Iz prethodne glave smo videli kako **algoritmi postaju programi** u kompjuteru.

U ovoj glavi ćemo učiti kako da predstavljamo i analiziramo algoritme.

2.1 Pseudokod i programski jezik C

Ova knjiga nema dovoljno materijala za učenje programiranja i pisanja C programa. Za proširivanje ovog znanja preporučujemo [1, 2, 6].

Pseudokod je način predstavljanja algoritama. Po sintaksi liči na **program** pisan u programskom jeziku **Pascal**[1]. Pascal je korišćen kao programski jezik za podučavanje programiranja. Korišćen je i za pisanje komercijalnih programa na personalnim računarima krajem sedamdesetih i početkom osamdesetih godina dvadesetog veka. Pascal kompajlere i IDE za IBM PC kompatibilce je prodavala kompanija Borland, koja se krajem osamdesetih preorijentisala na prodaju IDE za programski jezik C.

Sintaksa pseudokoda nije stroga, pseudokod se ne prevodi na mašinski jezik. Služi za opisivanje ideje i semantike algoritma onima koji znaju osnove programiranja.

Može se reći da je osmišljavanje algoritma mnogo teže od same implementacije - zapisivanja postojećeg algoritma u nekom programskom jeziku. Ili, može se reći da je pisanje pseudokoda **programiranje**, a zapisivanje pseudokoda u programskom jeziku **kodiranje**.

2.1.1 Sintaksa i semantika pseudo koda

Programi u pseudo kodu mogu da budu funkcije (**function**) koje direktivom **return** prekidaju izvršavanje i vraćaju argument **return** direktive. Mogu biti i procedure (**procedure**), koje dolaskom

[1] Niklaus Wirth, 1934 - , godine 1970. kreirao programski jezik Pascal, autor [6]

do **end procedure** ili direktive **return** završavaju program ne vraćajući izlaz.

Najčešće korištene direktive za regulisanje toka programa su

for ⋯ **to** ⋯ **do**	**while** ⋯ **do**	**repeat**	**if** ⋯ **then**
⋯	⋯	⋯	⋯
⋯	⋯	⋯	**else**
⋯	⋯	⋯	⋯
end for	**end while**	**until** ⋯	**end if**

Za izlaženje iz **for**, **while** i **repeat** petlje se koristi direktiva **break**.

Direktiva ← znači da se promenljivoj sa leva pridružuje vrednost izraza sa desna.

Tipovi promenljivih treba da su intuitivno jasni. Elementi niza se numerišu u uglastim zagradama od 1, 2, pa na dalje. Na primer: **A[3]** je treći član niza **A**.

Od osnovnih tipova podataka ćemo koristiti numerički tip, karakter, logički (Bulovski), pokazivače, strukture i nizove prethodnih tipova. Po potrebi i stringove, koji su zapravo nizovi karaktera.

Izraze ćemo računati nad numeričkim i logičkim tipovima koristeći tipične matematičke i logičke operacije.

Izrazi mogu biti logički, odnosno bulovski. Netačno se obeležava nulom (0) (ili false) a tačno (true) je sve što je različito od nule, obično jedan (1). Logički izrazi se koriste kao uslovi za izvršavanje uslovnih direktiva **while**, **until**, **if**.

Procedure imaju ulazne argumente, a funkcije i ulazne i izlazni argument, koji se može koristiti u okviru izraza. U skladu sa preporukama proceduralnih programskih jezika sve promenljive uključujuću ulazne unutar procedure ili funkcije treba da su **lokalne**. Ipak, nekad upotreba **globalne** promenljive daje jednostavnost programu, videćemo na programu za prolazak kroz graf.

Ulazni parametri mogu biti prosleđenih **po vrednosti** ili **adresi**. Kada se prosledi adresa promenljive, promene na lokaciji te adrese će se videti van procedure. Kad se parametar prosleđuje po vrednosti, pravi se duplikat parametra. U tom slučaju njegova vrednost može da se menja unutar procedure, ali po izlasku iz procedure duplikat se gubi. Pridržavaćemo se pravila za prenošenje argumenata iz programskog jezika C.

2.1.2 Specifičnosti kodiranja u programskom jeziku C

U programskom jeziku C podrazumevano (default) prosleđivanje parametara je po vrednosti.

Treba primetiti da su nizovi adrese memorijskog polja gde se nalazi imenovani niz. Numeracija članova niza kreće od nule (0). Poslednji član niza sa n elemenata ima redni broj $n-1$.

Pri prosleđivanju niza u funkciju ili proceduru pravi se duplikat adrese niza pa se duplikat prosledi. Da bi procedura znala koliko ima elemenata u nizu, mora se proslediti i dužina niza.

C je proceduralni programski jezik, sličan Pascalu. C ima mali broj osnovnih tipova a postoji velik broj C biblioteka koje omogućavaju rad sa apstraktnim tipovima podataka - ADT[2].

Kad se u UTF-8 kodu napiše C program za prevođenje na mašinski jezik, prvo se vrši pretprocesiranje. Pretprocesor učitava program i razrešava `#include` i `#define` komande.

[2] ADT = Abstract Data Type

Potom C kompajler učitava program od početka prema kraju. U konzolnim aplikacijama, kakve ćemo radi vežbe praviti, mora postojati glavna procedura ili funkcija **`main`** od koje počinje program da se izvršava.

Funkcije ili procedure koje su potprogrami glavnog programa se mogu navesti ili referisati pre procedure **main**. Isto tako se navode promenljive koje su u tom slučaju globalno vidljive za sve procedure i funkcije u tom fajlu, osim ako se lokalno predefinišu.

Radi provere svog poznavanje programskog jezika C rešite sledeći zadatak.

1. Zadatak: Šta ispisuju ova tri C programa? (*swap* = zamena, EN)

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

void swap(int a, int b)
{
    int temp;
    temp = a; a = b; b = temp;
}

void main()
{
    int a = 2,b = 3;
    swap( a, b );
    printf("%3d, %3d\n",a,b);
}
```

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

void swap(int *x, int *y)
{
    int temp;
    temp=*x; *x=*y; *y=temp;
}

void main()
{
    int a = 2,b = 3;
    swap(&a, &b );
    printf("%3d, %3d\n",a,b);
}
```

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

int a = 2, b = 3;
void swap(int x, int y)
{
    int temp;
    temp = a; a = b; b = temp;
}

void main()
{
    swap(a, b );
    printf("%3d, %3d\n",a,b);
}
```

2. Zadatak: Napisati u jeziku C program koji ispisuje rešenja kvadratne jednačine $ax^2+bx+c=0$.

Rešenje:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define epsilon 1e-14

// gcc main.c -o kvadrjed -lm

int main()
{
    double a,b,c,d,x1,x2;

    a = 1.0;
    b = 2.0;
    c = 2.0;
    d = b*b-4*a*c;

    if(d>epsilon){
        x1 = -b/2/a - sqrt(d)/2/a;
        x2 = -b/2/a + sqrt(d)/2/a;
        printf("x1 = %5.2f, x2 = %5.2f\n",x1,x2);
    }
    else
        if(d < -epsilon){
            printf("x1 = %5.2f + i * %5.2f, x2 = %5.2f + i * %5.2f\n",-b/2/a,-sqrt(-d)/2/a,
                -b/2/a,sqrt(-d)/2/a);
        } else
            printf("x1 = %5.2f, x2 = %5.2f\n",-b/2/a,-b/2/a);

    return 0;
}
```

GNU C kompajler se poziva komandom gcc. Podrazumevano je da napravi izvršnu verziju programa, uključujući preprocesiranje, kompajliranje, asembliranje, brisanje asemblerskog koda i povezivanje sa sistemskim bibliotekama (slika 1.1). U izvornom kodu se komandom `#include <math.h>` učitava heder fajl matematičke biblioteke. Da bi se i sama biblioteka povezala (link) u mašinski kod, potrebno je proslediti opciju `-lm`. Opcija `-o ime` određuje da će izvršni kod biti snimljen u fajl `ime`.

Treba primetiti upotrebu male konstante epsilon koja sprečava da greška zaokruživanja promeni znak diskriminante d i ispiše konjugovano kompleksna rešenja kad su zapravo realna jednaka.

Program se kompajlira i izvršava na sledeći način:

```
$ gcc main.c -o kvadrjed -lm
$ ./kvadrjed
x1 = -1.00 + i * -1.00, x2 = -1.00 + i *  1.00
```

3. Zadatak: Napisati funkciju u programskom jeziku C za nalaženje rednog broja drugog po veličini elementa niza A dužine n.

2.1.3 Programiranje u pseudo kodu

U ovoj glavi ćemo na primerima videti kako se zapisuju i analiziraju programi u pseudo kodu.

Prvo ćemo definisati pojam koji će nam biti važan za sortiranje u glavi 3.

DEFINICIJA 1 *Neka je dat niz elemenata koji se mogu porediti po veličini. Kaže se za dva člana tog niza da su u **inverziji** ako je element sa manjim **indeksom** (rednim brojem) veći od elementa sa većim indeksom.*

*Niz je **sortiran** ako ne postoje dva elementa u inverziji.*

```
1:  function PROGRAM1(A)
2:      n ← length(A)
3:      flag ← true
4:      for j ← 1 to n − 1 do
5:          if A[j] > A[j + 1] then
6:              flag ← false
7:          end if
8:      end for
9:      return flag
10: end function
```

4. Zadatak: Šta sa ulaznim nizom A radi algoritam dat levo? Da li se može ubrzati, a da daje isti rezultat?

Rešenje:

Algoritam levo proverava da li je ulazni niz sortiran.

U liniji 2 se bulovska[a] promenljiva flag[b] postavlja na podrazumevanu vrednost true, koja pretpostavlja da je niz sortiran.

[a] Boolean, EN - u čast George Boole, 1815 - 1864

[b] *flag* = zastavica, EN; *true* = tačno, EN; *false* = netačno, EN; *length* = dužina, EN

U for petlji se za sve članove niza proverava da li je neki član niza u **inverziji** sa sledećim. Ako je to barem jednom ispunjeno, promenljiva flag će biti postavljena na false i kad j prođe kroz vrednosti do $n-1$, tako će ostati. Onda će **return** direktiva vratiti false, što znači da niz nije sortiran. Vidi sledeći zadatak.

Ubrzavamo PROGRAM1 i pravimo PROGRAM1A tako što iza linije 6 ubacujemo direktivu **break**, koja izlazi iz for petlje kada se prepozna inverzija susednih elemenata. Naime, dovoljno je samo jednom postaviti promenljivu flag na false.

Ukoliko u implementaciji (kodiranju na programski jezik) ne postoji direktiva **break**, umesto for petlje treba koristiti **while** petlju, PROGRAM 1B.

```
function PROGRAM1A(A)
    n ← length(A)
    flag ← true
    for j ← 1 to n − 1 do
        if A[j] > A[j+1] then
            flag ← false
            break
        end if
    end for
    return flag
end function
```

```
function PROGRAM1B(A)
    n ← length(A)
    flag ← true
    j ← 1
    while flag &(j ≤ n − 1) do
        if A[j] > A[j+1] then
            flag ← false
        end if
        j ← j + 1
    end while
    return flag
end function
```

```
function PROGRAM1C(A)
    n ← length(A)
    flag ← true
    j ← 1
    while flag &(j ≤ n − 1) do
        flag ← A[j] ≤ A[j+1]
        j ← j + 1
    end while
    return flag
end function
```

Takođe se u **while** petlji umesto direktive **if** $\cdots$ **then** može koristiti dodeljivanje vrednosti logičkog izraza $A[j] \leq A[j+1]$ promenljivoj flag, program PROGRAM 1C. U poslednja dva programa vrši se jedno uvećavanje promenljive j koje je nepotrebno.

Postoji još puno primera gde čitaoci mogu isprobati svoje veštine pisanja algoritama u pseudokodu. Mnogi programi u ovoj knjizi će biti dati u pseudo kodu.

Preporučujemo vežbanje i isprobavanje kodiranja algoritama zapisanih u pseudo kodu u programskom jeziku C, njihovo kompajliranje i testiranje.

5. Zadatak: Dokazati da ako se **return** direktivom flag vrati kao true, da je niz sortiran, to jest, da ne mogu postojati neka dva elementa (nesusedna) koji su u inverziji.

2.2 Korektnost algoritama

Algoritmi, zajedno sa kompjuterom i operativnim sistemom su često tehnologija koja je ciljano birana i korišćena za rešavanje problema.

Ako bismo zanemarili kapacitete kompjutera: brzina CPU i raspoloživost memorije, ostaje pitanje da li bi posmatrani algoritam ikad završio i da li bi na kraju dao traženi rezultat. To je pitanje **korektnosti algoritama**.

Često algoritam koji je na raspolaganju nije jednostavan i za dokazivanje njegove korektnosti treba jak matematičko–logički aparat.

U prethodnom odeljku u zadatku 5. je postavljeno pitanje dokazivanja korektnosti PROGRAM1 za ispitivanje da li je ulazni niz sortiran.

U ostatku knjige ćemo korektnost objasniti bez prevelikog zalaženja u detalje matematičko–logičke preciznosti.

Korektnost algoritama i izračunljivost funkcija su se pojavili kao koncepti u teorijskim kompjuterima još pre nego što su kompjuteri stvarno napravljeni.

2.3 Efikasnost (kompleksnost) algoritama

Kompjuterski resursi ipak nisu neograničeni.

CPU najnovijih kompjutera su veoma brzi, odnosno, rade na velikim frekvencijama. Ali, tehnologija izrade CPU dozvoljava ograničene frekvencije procesora. Najbolje se **efikasnost** algoritma meri **brzinom izvršavanja** programa na **realnom kompjuteru** za zadati ulaz.

Efikasnost može da se meri i **upotrebom memorije**. U početku kompjuterske ere memorija je bila veoma ograničen resurs, a danas se velika količina memorije može jeftino kupiti. Ipak, dobro napisani programi se ne smeju razbacivati memorijom i smatra se za dva jednako brza algoritma da je efikasniji onaj koji troši manje memorije.

Algoritmi koje ćemo posmatrati u ovoj knjizi nisu veliki potrošači memorije, kompleksnost algoritama ćemo posmatrati kao brzinu izvršavanja za ulaz određene veličine.

Tako ćemo videti da je za ulazni niz dužine n brzina izvršavanja SELECTION SORT otprilike jednaka $c_1 n^2$, a brzina izvršavanja MERGE SORT otprilike $c_2 n \log n$.

Pri tome je $c_1 < c_2$, jer je SELECTION SORT jednostavniji od MERGE SORT, odnosno, ima manji *overhead* (administraciju) i troši manje memorije. Memorija u ovom slučaju nije kritičan resurs.

Eksperimentalno smo proverili (Tabela 3.7) da za ulaze veličine do približno $n = 1000$ važi $c_1 n^2 < c_2 n \log n$, odnosno, da je SELECTION SORT efikasniji (brži). Ipak, za ulaz veličine $n = 100000$ vreme izvršavanja SELECTION SORT je bilo $60.3s$, a vreme izvršavanja MERGE SORT je bilo $2.4s$. Za još veće nizove, razlika u kompleksnosti je puno veća(u korist MERGE SORT).

Razlog za to je što niz n^2 brže teži beskonačnosti od niza $n \log n$.

Apsolutna vrednost razlike vremena izvršavanja za ulaze male veličine n nije velika. Efikasnost algoritama je značajna za velike ulaze.

Precizno gledano, posmatraćemo asimptotsku efikasnost u zavisnosti od veličine ulaza n. Matematički precizno, upoređivaćemo efikasnost (vreme izvršavanja) algoritama za $n \to \infty$.

Vratimo se na PROGRAM1 da bi analizirali njegovu efikasnost.

```
1:  function PROGRAM1(A)
2:      n ← length(A)
3:      flag ← true
4:      for j ← 1 to n − 1 do
5:          if A[j] > A[j + 1] then
6:              flag ← false
7:          end if
8:      end for
9:      return flag
10: end function
```

6. Zadatak: U zavisnosti od n, **veličine ulaznog niza** A, koliko je vreme izvršavanja algoritma datog levo?

Rešenje: Da bismo odgovorili na ovo pitanje moramo posmatrati šta se dešava u procesu kodiranja ovog algoritma u program nekog programskog jezika i kompajliranja na mašinski jezik.
Jedna linija pseudo koda se pretvara u nekoliko linija asemblerskog koda. Vreme izvršavanja tih linija je uvek približno isto.

Stoga ćemo za linije koda $2,3,4,5,6,9$ označiti vreme izvršavanja c_2, c_3, c_4, c_5, c_6, c_9. Napravićemo izraz za vreme izvršavanja datog programa u zavisnosti od uvedenih veličina i n.

U tabeli koja sledi imamo koliko se puta koja od ovih linija izvršava

vreme izvršavanja	c_2	c_3	c_4	c_5	c_6	c_9
broj izvršavanja	1	1	n	$n-1$	m	1

S obzirom da ne znamo m: broj koliko puta će se naći susednih brojeva u inverziji (jedino znamo da je $0 \leq m \leq n-1$), razlikujemo tri slučaja:

Best case	$T(n) = c_2 + c_3 + nc_4 + (n-1)c_5 + 0\,c_6 + c_9 = \Theta(n)$
Worst case	$T(n) = c_2 + c_3 + nc_4 + (n-1)c_5 + (n-1)\,c_6 + c_9 = \Theta(n)$
Average case	$T(n) = c_2 + c_3 + nc_4 + (n-1)c_5 + \left\lfloor \frac{n-1}{2} \right\rfloor c_6 + c_9 = \Theta(n)$

U Average[3] case smo pretpostavili da je u pola slučajeva upoređivani par u inverziji.

Oznaka $\Theta(n)$ se koristi kada je funkcija koja se posmatra približno jednaka $const \cdot n$. Strogo formalno objašnjenje asimptotskog ponašanje $\Theta(n)$ funkcija ćemo dati u sledećem odeljku.

Vidimo da brzina (efikasnost) algoritma nije ista za sve ulazne nizove dužine n. Mi ćemo razlikovati tri slučaja: Best, Worst i Average. S tim što se prosek vremena izvršavanja može računati na razne načine. Ako se zna raspodela slučajne promenljive koja predstavlja statističko ponašanje ulaza, onda se Average case dobija kao očekivanje slučajne promenljive koja predstavlja vreme izvršavanja.

Nekad ćemo samo na osnovu Best i Worst case zaključiti asimptotski red Average case.

2.4 Asimptotsko ponašanje nizova koji teže beskonačnosti

Ako posmatramo kompleksnost algoritma za veličinu ulaza $n \in \mathbb{N}$, očigledno je da vreme izvršavanja $T(n)$ ne može biti negativan broj i da će monotono težiti beskonačnosti kad $n \to \infty$.

U ovom odeljku ćemo posmatrati funkcije nad skupom prirodnih brojeva $f : \mathbb{N} \to \mathbb{R}$, odnosno nizove. Nizovi koje ćemo posmatrati su nenegativni, monotoni i neograničeni, odnosno, teže beskonačnosti, $\lim_{n\to\infty} f(n) = \infty$. Interesuje nas kojom brzinom teže beskonačnosti.

Iako ti nizovi predstavljaju vreme izvršavanja za veličinu ulaza n, u zadacima ćemo zbog jednostavnosti dozvoliti da do nekog n_0 mogu biti negativni ili na delovima opadajući.

Uvešćemo relacije kojima poredimo brzine teženja beskonačnosti.

DEFINICIJA 2 *Veliko teta i veliko o ponašanje. (Θ i O)*

Klasu funkcija $\Theta(g)$ i klasu funkcija $O(g)$ definišemo:

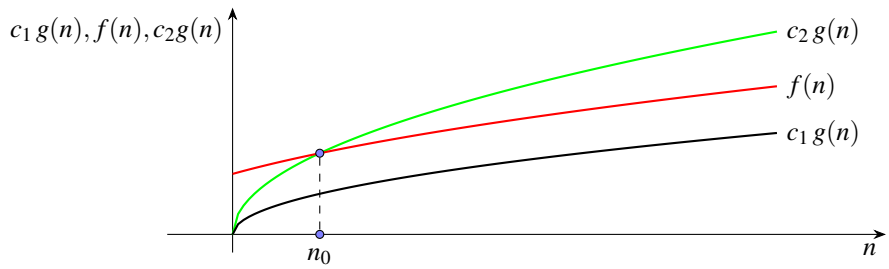
$$\Theta(g) = \{f \mid (\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \in \mathbb{N})(\forall n)\ (n \geq n_0) \Rightarrow (0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n))\},$$

$$O(g) = \{f \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n)(n \geq n_0) \Rightarrow (0 \leq f(n) \leq c\, g(n))\}.$$

Umesto da pišemo $f \in \Theta(g)$, pišemo $f = \Theta(g)$ i čitamo: funkcija f se ponaša kao $\Theta(g)$ (kao veliko teta od g).

Umesto da pišemo $f \in O(g)$, pišemo $f = O(g)$ i čitamo: funkcija f se ponaša kao $O(g)$ (kao veliko o od g).

[3] *best* = najbolji; *worst* = najgori; *average* = prosečan; *case* = slučaj, EN



Slika 2.1: Veliko teta i veliko o ponašanje: $f = O(g)$, $f = \Theta(g)$

Na primer, $3n^2 = \Theta(n^2)$, za $c_1 = 1, c_2 = 4, n_0 = 1$.
Takođe, $\ln n = O(n)$, za $c_1 = 1, n_0 = 1$.

7. Zadatak: Pokazati da nije $\ln n$ veliko teta od n.

Najbolje ćemo brzinu teženja ka besknačnosti, odnosno poređenje asimptotskog ponašanja razumeti kroz primere.

8. Zadatak: Pokazati da je $\frac{2}{3}n^2 - 2n = \Theta(n^2)$.

Rešenje: Treba naći n_0, c_1, c_2 tako da počev od n_0 važi $0 < c_1 n^2 \leq \frac{2}{3}n^2 - 2n \leq c_2 n^2$.
Za desnu nejednakosti očigledno je dovoljno uzeti $c_2 = \frac{2}{3}$. Da bismo našli c_1, podelimo levu nejednakost sa n^2. Dobijamo

$$0 < c_1 \leq \frac{2}{3} - \frac{2}{n}, \text{ odakle zaključujemo da možemo uzeti } n \geq 4 =: n_0.$$

Sad možemo uzeti za c_1 bilo koji broj koji zadovoljava $0 < c_1 \leq \frac{2}{3} - \frac{2}{4} = \frac{1}{6}$, recimo $c_1 := \frac{1}{6}$.

9. Zadatak: Pokazati da je $100n - 1000 = O(n^2)$.

10. Zadatak: Pokazati da je $100n + 1000 = O(n^2)$.

Rešenje: Da, za, recimo, $c_1 = 20$ i $n_0 = 10$, jer je

$$0 \leq 100n + 1000 \leq c_1 n^2 \Leftrightarrow 0 \leq \frac{100}{n} + \frac{1000}{n^2} \leq c_1 \Leftarrow n \geq 10 =: n_0.$$

11. Zadatak: Da li je $100n + 1000 = \Theta(n^2)$?

DEFINICIJA 3 *Klasa funkcija* $\Omega(g)$*:*

$$\Omega(g) = \{f | (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n)\ (n \geq n_0) \Rightarrow (0 \leq c\,g(n) \leq f(n))\}.$$

Umesto da pišemo $f \in \Omega(g)$, pišemo $f = \Omega(g)$ i čitamo: funkcija f se ponaša kao $\Omega(g)$ (kao veliko omega od g).
Na primer, $3n^2 = \Omega(n^2)$. $(c_1 = 1, n_0 = 1)$

12. Zadatak: Dati vezu Ω, Θ i O ponašanja.

Rešenje: Očigledno važi: $f = \Theta(g) \Leftrightarrow (f = \Omega(g) \wedge f = O(g))$.
Na primer, na slici 2.1 važi da je $f = \Omega(g)$, $f = O(g)$, $f = \Theta(g)$.

13. Zadatak: Pokazati da je $\frac{2}{3}n^2 - 2n = \Omega(n^2)$

14. Zadatak: Da li je $100n + 1000 = O(n)$?

Rešenje: Da, očigledno, za, recimo, $c_1 = 101$ i $n_0 = 1001$. Vidimo da je tvrđenje zadatka 10 pregrubo. Važi i $100n + 1000 = o(n^2)$ (malo o).

15. Zadatak: Da li je $100n + 1000 = \Theta(n)$?

16. Zadatak: Pokazati da je $n \ln n + n = O(n^2)$?

Rešenje: Da, jer je niz $\frac{n \ln n + n}{n^2}$ konvergentan (konvergira ka nuli), zato je ograničen, to jest postoji c_2 takvo da počev od nekog n_0 važi $0 \leq \frac{n \ln n + n}{n^2} \leq c_2$, što je ekvivalentno sa $0 \leq n \ln n + n \leq c_2 n^2$.

17. Zadatak: Pokazati da je $n \ln n + n = \Omega(n)$?

Asimptotske oznake imaju svoju analogiju sa brojevima:

$$\begin{aligned} f = \Omega(g) &\quad \leftrightarrows \quad f \geq g \\ f = O(g) &\quad \leftrightarrows \quad f \leq g \\ f = \Theta(g) &\quad \leftrightarrows \quad f = g \end{aligned}$$

DEFINICIJA 4 *Kažemo da je za nenegativni niz g klasa funkcija $o(g)$ (čitamo malo o od g):*

$$o(g) = \{f | (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n)\ (n \geq n_0) \Rightarrow (0 \leq f(n) < cg(n)\}$$

Kao i do sad, $f \in o(g)$ pišemo $f = o(g)$.
U matematičkoj analizi je česta upotreba oznake malo o. Razlika je što mi posmatramo samo nenegativne funkcije prirodnih brojeva, odnosno nenegativne nizove.
Za (počev od nekog n_0) nenegativne neograničene nizove f i g važi

$$f = o(g) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Analogija sa brojevima bi bila: $f = o(g) \leftrightarrows f < g$.

Iz definicije je očigledno da $f = o(g) \Rightarrow f = O(g)$. Obrnuto ne mora biti tačno.

Napomenimo još da za asimptotske oznake važi **tranzitivnost**:

$$f = O(g) \wedge g = O(h) \Rightarrow f = O(h)$$
$$f = \Theta(g) \wedge g = \Theta(h) \Rightarrow f = \Theta(h)$$
$$f = \Omega(g) \wedge g = \Omega(h) \Rightarrow f = \Omega(h)$$
$$f = o(g) \wedge g = o(h) \Rightarrow f = o(h)$$

Uvešćemo relaciju $\prec$ na sledeći način:

Za nizove f i g koji teže ka beskonačnosti kažemo da je f **asimptotski manja beskonačnost** od g, pišemo $f \prec g$, ako je $f = o(g)$.

Na primer $n\ln n + n \prec n^2$ jer je $n\ln n + n = o(n^2)$.

Tranzitivnost malog o daje shemu za poređenje niza beskonačnih veličina, slika 2.2.

Po toj shemi granična vrednost količnika neke od funkcija sa funkcijom desno od nje u nizu je 0. Često se u upotrebi pojavljuju i proizvodi nekih od funkcija sa slike 2.2.

$$\cdots \prec \log\log n \prec \log n \prec n^{\alpha} \prec n^{\beta} \prec a^n \prec b^n \prec n! \prec n^n \prec \cdots$$
$$(\text{za } 0 < \alpha < \beta \text{ i } 1 < a < b)$$

Slika 2.2: Poređenje beskonačnih veličina

2.5 Rekurzija

Rekurzija je deo algoritma koji poziva samog sebe sa promenjenim ulazom ili stanjem neke globalne promenljive. Da se ne bi beskonačno vrteo u petlji, algoritam mora imati kraj rekurzije.

Strogo formalno dokazivanje korektnosti rekurzivnih programa ima posebne tehnike.

Pisanje rekurzivnih programa daje jasnoću i eleganciju programu / algoritmu.

Svaki poziv programa zahteva od operativnog sistema odvajanje memorijskih resursa. Ako su ti resursi veliki i / ili je velik broj rekurzivnih poziva (drvo rekurzije), rekurzivni program je spor.

Ako je rekurzivni poziv na kraju algoritma, to se zove **repna rekurzija**. Posle povratka iz repne rekurzije algoritam ne koristi ostale promenljive i resurse iz svog nivoa. Poznati kompajleri umeju da detektuju repnu rekurziju i da optimizuju kod, čime značajno ubrzavaju izvođenje programa.

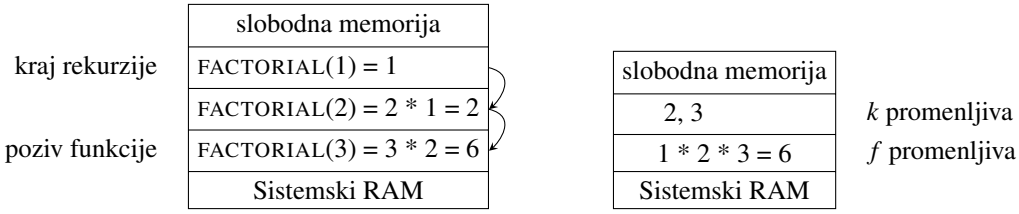
Kompajleri i interpreteri mogu da koriste brojače dubine rekurzije i da ne dozvole dubinu veću od unapred definisane. Time se sprečavaju greške u pisanju rekurzivnih procedura i preterano korišćenje kompjuterskih resursa.

Iako su rekurzivni programi elegantni za pisanje i razumevanje, u produkcionim programima se izbegavaju. Postoji način da se program derekurzivira. Često se za to koriste stekovi, apstraktni tip podataka koji ćemo proučavati u glavi 5. Na žalost, takvi programi su obično teški za razumeti i prepravljati. Za dobijanje na efikasnosti programa najbolja varijanta je da se za problem koji se rešava nađe nerekurzivno rešenje.

Primer rekurzivnog algoritma na kome se lako vidi primena rekurzije je algoritam za računanje faktorijala $n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$.

```
function FACTORIAL(n)
    if n ≤ 1 then
        return 1
    else
        return n · FACTORIAL(n − 1)
    end if
end function
```

```
function FACTORIAL(n)
    f ← 1
    for k ← 2 to n do
        f ← f · k
    end for
    return f
end function
```



Slika 2.3: Prikaz memorije pri rekurzivnom i iterativnom pozivu FACTORIAL(3)

Levo je prikazana rekurzivna, a desno iterativna verzija programa FACTORIAL.

Za oba programa, po definiciji, FACTORIAL(0) = 1.

U rekurzivnoj verziji program poziva samog sebe sa smanjenom vrednošću ulaza. Rekurzija završava kad je ulazna vrednost 1. Onda se program vraća iz rekurzije množeći vraćenu vrednost sa ulazom na prethodnom nivou sve dok ne dođe do nivoa prvog poziva funkcije i vrati traženu vrednost, slika 2.3 levo.

U iterativnoj verziji promenljivoj f se dodeli vrednost 1, a onda se u petlji redom množi sa brojevima od 2 do vrednosti ulaza i na kraju vrati traženu vrednost, slika 2.3 desno.

Ako je kompajleru uključena opcija optimizacije koda, on u rekurzivnoj verziji treba da prepozna repnu rekurziju. Tako kompajliran program se izvršava, za veliko n, puno brže od neoptimizovanog programa, jednako brzo kao iterativna verzija. U svakom slučaju, iterativna verzija je brža i zauzima manje memorije.

Ovde je još važno naglasiti da tip podataka u koji se želi smestiti vrednost faktorijala može biti integer samo do FACTORIAL(13), jer se za vrednosti veće od 13 pojavljuje overflow[4]. Ako se koristi double floating point tip podataka, onda faktorijal može da se izračuna do FACTORIAL(170). Zato što niz $n!$ brzo teži beskonačnosti i izračunata vrednost izlazi iz opsega, vidi sliku 2.2.

18. Zadatak: Fibonačijev niz čine redom brojevi $0,1,1,2,3,5,8,13,\ldots$ Prva dva su redom 0 i 1, svaki sledeći je zbir prethodna dva. Napisati rekurzivni i iterativni algoritam za računanje n-tog Fibonačijevog broja. (Dajući im redne brojeve počev od nule: $0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2, \ldots$)

Rešenje:

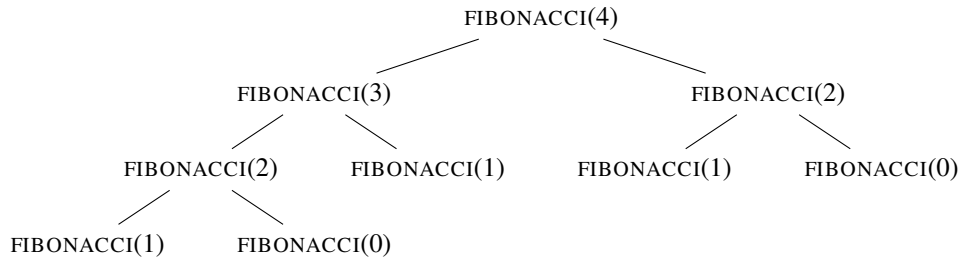
[4] overflow - programska greška kad izračunata vrednost izraza ne može da se smesti u osnovni tip podataka

```
function FIBONACCI(n)
    if n ≤ 1 then
        return n
    else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
    end if
end function
```

```
function FIBONACCI(n)
    if n ≤ 1 then
        return n
    else
        f0 ← 0
        f1 ← 1
        for k ← 2 to n do
            f ← f0 + f1
            f0 ← f1
            f1 ← f
        end for
    end if
    return f
end function
```

Vidimo koliko je rekurzivna verzija elegantnija i kraća od iterativne verzije. Ipak, izvršavanje rekurzivne verzije za veliko n traje mnogo duže.

Na slici 2.4 vidimo rekurzivne pozive funkcije FIBONACCI za $n = 4$. Zauzeće memorije je $\Theta(n)$, jer je proporcionalno dubini rekurzije.



Slika 2.4: Rekurzivni pozivi FIBONACCI(n) za $n = 4$

Broj poziva funkcije FIBONACCI je manji od broja čvorova punog binarnog drveta sa n nivoa, koji iznosi

$$2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} = 2^n - 1.$$

Vreme izvršavanja funkcije u svakom čvoru je približno isto, neka iznosi c.

Onda je $T(n) =$ ukupno vreme izvršavanja funkcije FIBONACCI(n) manje od $c \cdot (2^n - 1)$, odnosno, $T(n) = O(2^n)$.

Postoji zatvorena formula za računanje Fibonačijevih brojeva

$$\text{FIBONACCI}\,(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n . \tag{2.1}$$

Pošto se formula (2.1) računa u floating point tipu podataka, zbog greške zaokruživanja nekad se neće dobiti tačna vrednost. U tom slučaju funkcija round[5] može da dovede do tačnog broja za sve

[5] round - zaokružiti, EN

n za koje se (2.1) može izračunati.

Bez ulaženja u detalje analize vremena izvršavanja rekurzivne funkcije FIBONACCI, pažljivim posmatranjem drveta poziva rekurzivne funkcije FIBONACCI (n) može se zaključiti da je vreme izvršavanja rekurzivne verzije reda veličine FIBONACCI (n): $T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \approx \Theta(1.618^n)$.

Naravno da je to mnogo lošije od vremena izvršavanja iterativne verzije: $\Theta(n)$, što je lošije od računanja pomoću formule (2.1) koje je $\Theta(1)$ (iako $g(n) = 1$ ne teži beskonačnosti, klasa $\Theta(1)$ može da se definiše preko definicije 2).

19. Zadatak: Napisati hronološki pozive rekurzivne funkcije FIBONACCI(5) i prebrojati ih.

20. Zadatak: Na kompjuteru je izmereno vreme izvršavanja rekurzivne funkcije FIBONACCI za $n = 40$ i za $n = 42$: $T(40) = 5.5492s$ i $T(42) = 14.5920s$. Ako je vreme izvršavanja približno jednako $T(n) = c \cdot q^n$, izračunati q i c.

Rešenje: $q = \sqrt{T(42)/T(40)} = 1.6216, \quad c = T(40)/q^{40} = 2.2210 \cdot 10^{-8}$

21. Zadatak: Koristeći podatke iz prethodnog zadatka predvideti koliko bi vremena trebalo da se izvrši FIBONACCI (50) i FIBONACCI (100).

22. Zadatak: Dokazati da je t_n, broj poziva rekurzivne funkcije FIBONACCI (n), jednak $t_n = 2 \cdot$FIBONACCI $(n+1) - 1$ za $n = 1, 2, 3, \ldots$

Rešenje: Označimo n-ti Fibonačijev broj FIBONACCI $(n) =: f_n$. Imamo $f_1 = 1$, $f_2 = 1$, $f_3 = 2, \ldots$
Tvrđenje je za $n = 1$ očigledno tačno, $t_1 = 2f_2 - 1 = 1$.
Za $n \geq 2$ dokaz ćemo dati shemom matematičke indukcije:

$$\begin{array}{r|l} n = 2 & t_2 = 3 = 2 \cdot f_3 - 1 \\ \hline n \mapsto n+1 & t_{n+1} = t_n + t_{n-1} + 1 = 2f_{n+1} - 1 + 2f_n - 1 + 1 = 2f_{n+2} - 1 \end{array} \tag{2.2}$$

Jednakost $t_n = t_{n-1} + t_{n-2} + 1$ se vidi sa slike 2.4 (za $n = 3$). Za računanje f_n broj rekurzivnih poziva je jednak broju poziva za računanje f_{n-1} plus broju poziva za računanje f_{n-2} plus 1.

23. Zadatak: Dokazati da je vreme izvršavanja rekurzivne funkcije FIBONACCI $T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Glava 3

Sortiranje

Kad govorimo o nizu u programiranju, mislimo na konačan niz susednih memorijskih lokacija istog tipa. Da bi se mogli sortirati, elementi niza treba da su uporedivi relacijom poretka. Zadatak algoritama za sortiranje je da elemente ulaznog niza poređa po veličini, obično od najmanjeg do najvećeg. Niz je ulazno izlazni parametar, prosleđujemo ga po adresi, algoritam pišemo kao proceduru. Mi ćemo izložiti samo neke algoritme za sortiranje i izvršiti njihovo poređenje.

3.1 BUBBLE SORT

PROGRAM1 iz prethodne glave se može doraditi tako da se izvrši zamena, swap $(A, j, j+1)$, ako je upoređivani par $(j, j+1)$ u inverziji. To se ponavlja u **repeat-until** petlji. Ponavljanje prestaje kad se dobije **sortiran niz**.

Tako dobijen algoritam za sortiranje se naziva BUBBLE SORT[1]. Služi za sortiranje nizova. Performanse su mu slabe, pa se izučava kao reper za poređenje sa drugim algoritmima za sortiranje i za učenje programiranja. Njegov kod je PROGRAM2.

```
1:  procedure PROGRAM2(A)
2:      n ← length(A)
3:      repeat
4:          flag ← true
5:          for j ← 1 to n − 1 do
6:              if A[j] > A[j + 1] then
7:                  swap(A, j, j + 1)
8:                  flag ← false
9:              end if
10:         end for
11:     until flag
12: end procedure
```

24. Zadatak: Da li će i zašto dobijeni algoritam ikad izaći iz repeat-until petlje?

Rešenje:
Program u for petlji "gura" veće elemente na koje nailazi usput prema kraju. Prvi put će **for** petlja "odgurati" najveći element do kraja, sledeći put će odgurati drugi po veličini element, i tako dalje.
Dakle: sigurno će se repeat-until petlja završiti posle najviše n ciklusa, gde je n dužina ulaznog niza.
Može se reći da sortirani deo (na kraju niza) **inkrementalno raste** posle svakog prolaza kroz **repeat-until** petlju.

[1] bubble - mehurić, EN - najveći elementi niza isplivavaju na kraj, kao mehurići u vodi

Korektnost algoritma: Posle k izvođenja repeat-until petlje, barem k elemenata sa kraja će biti sortirani.

Da bi se algoritam ubrzao, može se smanjiti gornja granica for petlje (PROGRAM2A): Između linije 10 i 11 ubacimo liniju

$n \leftarrow n-1$.

Program se može još ubrzati ako se uvede promenljiva *newn* koja "pamti" poslednji element koji je u for petlji zamenjen, jer su svi ostali iza njega sortirani. Rešenje je PROGRAM2B.

```
procedure PROGRAM2A(A)
    n ← length(A)
    repeat
        flag ← true
        for j ← 1 to n − 1 do
            if A[j] > A[j+1] then
                swap(A, j, j+1)
                flag ← false
            end if
        end for
        n ← n − 1
    until flag
end procedure
```

```
procedure PROGRAM2B(A)
    n ← length(A)
    repeat
        newn ← 0
        for j ← 1 to n − 1 do
            if A[j] > A[j+1] then
                swap(A, j, j+1)
                newn ← j
            end if
        end for
        n ← newn
    until n = 0
end procedure
```

Zajedničko ime za PROGRAM2, PROGRAM2A, PROGRAM2B i ostale verzije sortiranja u kojima se zamenjuju mesta samo susednim elementima je BUBBLE SORT.

Zamena elemenata niza (swap) od svih komandi iz gornjih programa troši najviše vremena, a potom poređenje elemenata niza. Zato je zanimljivo videti koliko zamena i koliko poređenja se vrši u dobijenim programima za sortiranje.

25. Zadatak: Dokazati da PROGRAM2B ima $\leq$ poređenja nego PROGRAM2A.
Dokazati da PROGRAM2A ima $\leq$ poređenja nego PROGRAM2.

26. Zadatak: Koliko poređenja (linija 6) i koliko zamena (linija 7) će BUBBLE SORT programi izvršiti za ulaz $[5,2,4,6,1,3]$?

Rešenje: Za PROGRAM2 broj prolazaka kroz **repeat-until** petlju zavisi od sadržaja ulaznog niza. U sledećoj tabeli imamo zapisano koji element se menjao sa kojim u prolazu 1, 2, 3, 4, 5.

broj prolaza	1	2	3	4	5
menja se:	5 sa 2	5 sa 1	4 sa 1	2 sa 1	
	5 sa 4	5 sa 3	4 sa 3		
	6 sa 1				
	6 sa 3				

Iz petlje se izlazi kad u nekom prolazu ne bude zamena. U našem slučaju to je prolaz broj 5.

Broj poređenja za PROGRAM2:
Kreće se od prvog elementa i on se poredi sa 5 iza njega. Pošto je bilo 5 prolazaka kroz **repeat-until** petlju, PROGRAM2 ima ukupno $5 \cdot 5 = 25$ poređenja.

Broj zamena za PROGRAM2:
Pažljivim posmatranjem vidimo da se svaki element niza menja sa svim elementima sa kojima je u inverziji. Za naš niz ima ukupno 9 zamena (inverzija).

Broj poređenja nije isti za svaki ulaz, a ni za svaku verziju BUBBLE SORT. Ali, za svaki ulaz, broj zamena je isti za sva tri programa i jednak je ukupnom broju inverzija ulaznog niza, u konkretnom slučaju: 9.

Ako se ulaz $[5,2,4,6,1,3]$ propusti kroz PROGRAM2A i PROGRAM2B, pažljivim prebrojavanjem broja poređenja dobićemo:

algoritam	br. poređenja	br. zamena
PROGRAM2	25	9
PROGRAM2A	15	9
PROGRAM2B	14	9

27. Zadatak: Napisati u programskom jeziku C proceduru swap (A,i,j) za zamenu elemenata niza. Napisati proceduru exchange (A,i,j) koja pre zamene proverava da li su to različiti elementi.

3.2 INSERTION SORT

```
1:  procedure INSERTION SORT(A)
2:      for j ← 2 to length(A) do
3:          key ← A[j]
4:          i ← j − 1
5:          while i > 0 & A[i] > key do
6:              A[i + 1] ← A[i]
7:              i ← i − 1
8:          end while
9:          A[i + 1] ← key
10:     end for
11: end procedure
```

Algoritam polazi od drugog elementa i ide do kraja: pretpostavlja da je niz od početka do posmatranog elementa (brojač j) sortiran.
Upoređuje se posmatrani element sa elementima prema početku (brojač i), pomeraju se elementi (linija 6) sortiranog dela koji su veći od posmatranog i kad se nađe mesto, upiše se posmatrani element (linija 9).
U kodu sa leve strane poređenje ključeva se vrši u liniji 5, ako je $i > 0$. Upisivanja se vrše u linijama 3, 6 i 9.

Pre **for** petlje sortiran podniz je samo prvi element. U poslednjoj liniji **for** petlje *key* se upisuje na ispravno $(i+1)$-vo mesto, pa je deo od početka do j-tog elementa sortiran. Sortirani deo se inkrementalno povećava u **for** petlji koja završava sa poslednjim elementom niza. Time je pokazana korektnost algoritma.

28. Zadatak: Primeniti algoritam INSERTION SORT na ulaz $[5,2,4,6,1,3]$ i prebrojati broj poređenja i upisivanja elemenata u niz, odnosno u privremenu promenljivu.

Uporediti dobijeni algoritam sa algoritmom iz zadatka 26 po broju poređenja i broju upisivanja.

Rešenje: Ako se propusti ulaz $[5,2,4,6,1,3]$ kroz algoritam, pažljivim prebrojavanjem dobijamo da se poređenje ključeva vršilo 12 puta, a upisivanja (linije 3, 6, i 9 zajedno) 19 puta.

Znajući da je za jednu zamenu iz BUBBLE SORT (PROGRAM2) potrebno izvršiti 3 pisanja, za ovaj ulaz INSERTION SORT je bolji jer je BUBBLE SORT imao $9 \cdot 3 = 27$ upisivanja.

29. Zadatak: Kako izgleda permutacija elemenata skupa $\{1,2,3,4,5,6\}$ koja je najbolji, a kako permutacija koja je najgori slučaj za broj poređenja i za broj pisanja INSERTION SORT algoritma?

30. Zadatak: Napraviti analizu vremena izvršavanja INSERTION SORT algoritma za niz dužine n.

Dodelimo linijama 2,3,4,5,6,7,9 redom vremena izvršavanja $c_2, c_3, c_4, c_5, c_6, c_7, c_9$. Broj izvršavanja određene linije ćemo napisati desno kao komentar. "**end**" linijama ne dodeljujemo vreme.

```
1:  procedure INSERTION SORT(A)
2:      for i ← 2 to length(A) do                ▷ n
3:          key ← A[i]                           ▷ n − 1
4:          j ← i − 1                            ▷ n − 1
5:          while j > 0 & A[j] > key do          ▷ Σ_{i=2}^{n} t_i
6:              A[j + 1] ← A[j]                  ▷ Σ_{i=2}^{n} (t_i − 1)
7:              j ← j − 1                        ▷ Σ_{i=2}^{n} (t_i − 1)
8:          end while
9:          A[j + 1] ← key                       ▷ n − 1
10:     end for
11: end procedure
```

Dobijamo vreme izvršavanja algoritma:

$$T(n) = c_2 n + (c_3 + c_4)(n-1) + c_5 \sum_{2}^{n} t_i + (c_6 + c_7) \sum_{2}^{n} (t_i - 1) + c_9 (n-1),$$

gde je t_i broj koliko puta će se i-ti element pomeriti prema početku plus jedan - kad **while** petlja dobije **false**. Unutrašnjost petlje računamo da se izvršava $t_i - 1$ puta. t_i je najmanje 1, a najviše i.

Best case $t_i = 1$
$T_B(n) = c_2 n + (c_3 + c_4)(n-1) + c_5(n-1) + c_9(n-1) = \Theta(n)$

Worst case $t_i = i$
$T_W(n) = c_2 n + (c_3 + c_4)(n-1) + c_5(n(n+1)/2 - 1) + (c_6 + c_7)n(n-1)/2 + c_9(n-1) = \Theta(n^2)$

31. Zadatak: Znajući da je worst-case vreme izvršavanja INSERTION SORT algoritma $O(n^2)$, n je veličina ulaza, da li to znači da će za svaki ulazni niz veličine n vreme izvršavanja biti $O(n^2)$?

32. Zadatak: Znajući da je worst-case vreme izvršavanja INSERTION SORT algoritma $\Theta(n^2)$, n je veličina ulaza, da li to znači da će za svaki ulazni niz veličine n vreme izvršavanja biti $\Theta(n^2)$?

Rešenje: Ne. Zato što $\Theta(n^2)$ ponašanje daje i gornju i donju granicu. Donja granica neće biti ispunjena jer za sortiran niz veličine n (best-case za INSERTION SORT) vreme izvršavanja je $\Theta(n)$.

33. Zadatak: Za sortiran niz dužine n, koji algoritam je brži: BUBBLE SORT ili INSERTION SORT?

34. Zadatak: Naći asimptotsku ocenu za $T(n)$, vreme izvršavanja INSERTION SORT algoritma za ulaz dužine n.

Rešenje: U zadatku 30. vidimo da je za $T(n)$ Best case $T_B(n) = \Theta(n)$ i Worst case $T_W(n) = \Theta(n^2)$. Naša procena vremena izvršavanja na osnovu toga je: $T(n) = \Omega(n)$ i $T(n) = O(n^2)$.

3.3 SELECTION SORT

```
1:  procedure SELECTION SORT(A)
2:      n ← length(A)
3:      for i ← 1 to n − 1 do
4:          i_min ← i
5:          for j ← i + 1 to n do
6:              if A[j] < A[i_min] then
7:                  i_min ← j
8:              end if
9:          end for
10:         exchange(A, i, i_min)
11:     end for
12: end procedure
```

Ovaj algoritam od prvog do poslednjeg elementa bira najmanji iza njega, $A[i_{min}]$, i menja mu mesto sa trenutnim elementom, $A[i]$.
Razlika između funkcije exchange$(A,i,i_{\min})$ i swap$(A,i,i_{\min})$ je u tome što se u exchange[a] prvo proveri da li je u pitanju isti elemenat, da se ne bi vršilo nepotrebno pisanje.
Ovaj algoritam je po broju upisivanja (zamena) elemenata niza u proseku bolji od BUBBLE SORT i INSERTION SORT. Kad se sortiranje vrši na mediju na kome je upisivanje "skupo" (spore memorije), preporučuje se ovaj algoritam.

[a] *exchange* = zamena, EN

Ovo je, kao BUBBLE SORT i INSERTION SORT, **inkrementalno sortiranje**. Ideja inkrementalnog sortiranja je da se deo niza sa početka ili kraja održava sortiranim i da se taj deo uvećava.

Korektnost algoritma se vidi iz toga što posle svakog prolaska kroz spoljnu **for** petlju (linije 3-11) deo niza od početka do i ostaje sortiran, a na kraju petlje poslednji element niza nije manji od bilo kog elementa pre njega, jer je upoređivan sa njima.

35. Zadatak: Primeniti algoritam SELECTION SORT na ulaz $[5,2,4,6,1,3]$, prebrojati broj poređenja (linija 6), broj zamena elemenata niza (linija 10 sa $i \neq i_{\min}$) i broj upisivanja indeksa (linije 4 i 7).

36. Zadatak: Napraviti analizu brzine rada SELECTION SORT algoritma za niz dužine n.

Rešenje:
Označimo sa t_i, $i = 1,2,\ldots,n$, broj izvršavanja linije 5 u i-tom prolasku kroz petlju iz linije 3.
Označimo sa s_i, $i = 1,2,\ldots,n$, broj izvršavanja linije 7 u i-tom prolasku kroz petlju iz linije 3.

```
1:  procedure SELECTION SORT(A)
2:      n ← length(A)                          ▷ 1
3:      for i ← 1 to n do                      ▷ n+1
4:          i_min ← i                          ▷ n
5:          for j ← i+1 to n do                ▷ Σ_{i=1}^{n} t_i
6:              if A[j] < A[i_min] then        ▷ Σ_{i=1}^{n} (t_i − 1)
7:                  i_min ← j                  ▷ Σ_{i=1}^{n} (s_i − 1)
8:              end if
9:          end for
10:         if i ≠ i_min then                  ▷ n
11:             swap(A, i, i_min)              ▷ r
12:         end if
13:     end for
14: end procedure
```

Napisali smo ponovo algoritam sa brojem ciklusa izršavanja desno u komentaru.

Način implementacije pseudo koda SELECTION SORT algoritma, brzina i sposobnost procesora i memorije može da utiče na brzinu rada algoritma. Uz male razlike izvršna verzija koja se dobije kompajliranjem programa napisanog na osnovu ovog pseudo koda treba da bude otprilike kao u ovoj analizi.

Liniji algoritma i dodeljujemo vreme izvršavanja c_i.

Vreme za ulazni niz A dužine n je

$$T(n) = c_2 + c_3(n+1) + c_4 n + c_5 \sum_{i=1}^{n} t_i + c_6 \sum_{i=1}^{n} (t_i - 1) + c_7 \sum_{i=1}^{n} (s_i - 1) + \\ + c_{10} n + c_{11} r$$

Poznato je: $t_i = n - i + 1$, $s_i \leq t_i$, $r \leq n$, odatle dobijamo dobijamo

$$T(n) = c_2 + c_3(n+1) + c_4 n + c_5 \left(\frac{n^2}{2} + \frac{n}{2}\right) + c_6 \left(\frac{n^2}{2} - \frac{n}{2}\right) + c_7 \sum_{i=1}^{n} (s_i - 1) + \\ + c_{10} n + c_{11} r$$

Worst case

$$s_i = t_i, \quad r = n$$

$$\begin{aligned} T(n) &= c_2 + c_3(n+1) + c_4 n + c_5 \left(\frac{n^2}{2} + \frac{n}{2}\right) + (c_6 + c_7)\left(\frac{n^2}{2} - \frac{n}{2}\right) + c_{10} n + c_{11} n \\ &= n^2(c_5 + c_6 + c_7)/2 + n\left(c_3 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_{10} + c_{11}\right) + c_2 + c_3 \\ &= \Theta(n^2) \end{aligned}$$

Best case

$$s_i = 0, \quad r = 0$$

$$\begin{aligned} T(n) &= c_2 + c_3(n+1) + c_4 n + c_5 \left(\frac{n^2}{2} + \frac{n}{2}\right) + c_6 \left(\frac{n^2}{2} - \frac{n}{2}\right) + c_{10} n \\ &= n^2 \frac{c_5 + c_6}{2} + n\left(c_3 + c_4 + \frac{c_5 - c_6}{2} + c_{10}\right) + c_2 + c_3 \\ &= \Theta(n^2) \end{aligned}$$

Naravno da je **Average case** $T(n) = \Theta(n^2)$.

37. Zadatak: Naći permutaciju brojeva $\{1, 2, \ldots, n\}$ koja je za SELECTION SORT najgori slučaj.

38. Zadatak: Za niz brojeva $[n, (n-1), \ldots, 2, 1]$ naći broj zamena algoritma SELECTION SORT.

Za parno n biće $n/2$, za neparno n biće $(n-1)/2$. Za oba slučaja $\lfloor n/2 \rfloor$.

3.4 MERGE SORT

```
procedure SORT(A, p, r)
    ▷ Procedura koja se rekurzivno poziva
    if p < r then
        q ← ⌊(p+r)/2⌋
        SORT(A, p, q)
        SORT(A, q+1, r)
        MERGE(A, p, q, r)
    end if
end procedure
procedure MERGE SORT(A)
    ▷ Glavna procedura koju poziva korisnik
    n ← length (A)
    SORT(A, 1, n)
end procedure
```

Ideja MERGE SORT algoritma je da kad niz A ima deo od p do q koji je sortiran, isto kao i deo od $q+1$ do r, da se izvrši spajanje po veličini (MERGE) u sortirani podniz od p do r. Slika 3.1.

To ima smisla ako je $p < r$, a najveća efikasnost se postiže ako su pomenuti delovi jednaki po veličini.

Sortiranost u oba dela se postiže na isti način, rekurzivno primenjujući SORT na deo od p do q i SORT na deo od $q+1$ do r.

Ako je $p = r$, onda je to niz od jednog elementa, on se smatra sortiranim. Za $p = r$ procedura SORT se vraća iz rekurzije.

Neparan broj elemenata se ne može podeliti na dva jednaka dela, zato $q \leftarrow \lfloor (p+r)/2 \rfloor$.

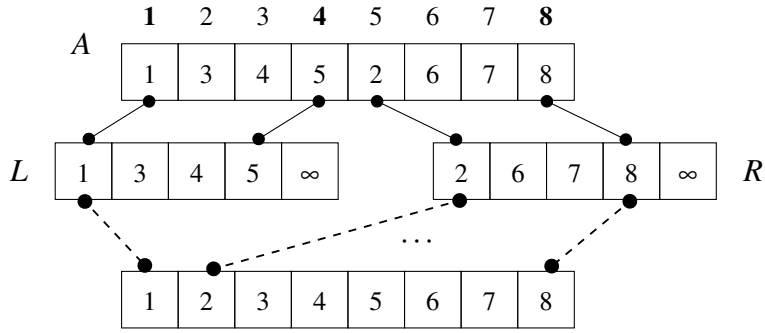
Za korisnike je napravljena glavna (umotavajuća) procedura MERGE SORT koja će sortirati ceo niz A, pozivajući rekurzivni SORT sa $p = 1$ i $r =$ length (A).

Sad nam preostaje da napišemo proceduru MERGE[2] koja bi uradila pomenuto spajanje po veličini.

U realizaciji ćemo koristiti takozvani **džoker simbol** ∞ koji je veći ili jednak od svih elemenata korišćenog tipa. Ova tehnika omogućava pojednostavljenje koda uz neznatni utrošak memorije. Ti simboli ne postoje u svakom tipu podataka. Za tip podataka int, može se koristiti maxint[3].

Prvo ćemo podniz od p do q prebaciti u niz L, dodati mu džoker simbol na kraj. Potom isto to za podniz od $q+1$ do r u niz R.

Džoker simboli služe da ne bismo morali komplikovati algoritam proverama da li je niz L ili R iscrpljen. Pošto su džoker simboli postavljeni na kraj nizova L i R, i pošto za sve x važi $x \leq \infty$, neće se prebaciti džoker simboli iz nizova L i R u podniz A od p do r.



Slika 3.1: Spajanje sortiranih podnizova od 1 do 4 i od 5 do 8: MERGE ([1,3,4,5,2,6,7,8],1,4,8)

[2] *merge* - spojiti, stopiti, EN

[3] maxint - najveći broj koji se može smestiti u 32-bitni integer = $2^{31} - 1 = 2.147.483.647$

```
1:  procedure MERGE(A, p, q, r)
2:      ▷ spaja sortirane podnizove niza A od p do q i od q+1 do r
3:      for k ← p to q do                    ▷ Prebacujemo prvi podniz u L
4:          L[k − p + 1] ← A[k]
5:      end for
6:      L[q − p + 2] ← ∞                     ▷ Dodajemo džoker simbol na kraj niza L
7:      for k ← q + 1 to r do                ▷ Prebacujemo drugi podniz u R
8:          R[k − q] ← A[k]
9:      end for
10:     R[r − q + 1] ← ∞                     ▷ Dodajemo džoker simbol na kraj niza R
11:     i ← 1                                ▷ Priprema
12:     j ← 1
13:     for k ← p to r do                    ▷ Spajanje po veličini
14:         if L[i] ≤ R[j] then
15:             A[k] ← L[i]
16:             i ← i + 1
17:         else
18:             A[k] ← R[j]
19:             j ← j + 1
20:         end if
21:     end for
22: end procedure
```

Glavna radnja procedure MERGE je spajanje po veličini, odnosno, vraćanje elemenata iz nizova L i R u ispravnom redosledu: linije 13-21. Brojač i pokazuje do kog smo elementa stigli u nizu L, a brojač j u nizu R.

Neka je $n = r - p + 1$ broj elemenata podniza od rednog broja p do r uključujući i njih i neka važi $p \leq q \leq r$. Neka je vreme izvršavanja linije k jednako c_k, a broj izvršavanja linije k neka je n_k, za $k \in I = \{3,4,6,7,8,10,11,12,13,14,15,16,18,19\}$.

Možemo primetiti da će ukupan broj izvršavanja linije 4 i linije 8 biti n, isto kao par linija 15 i 18, odnosno 16 i 19.

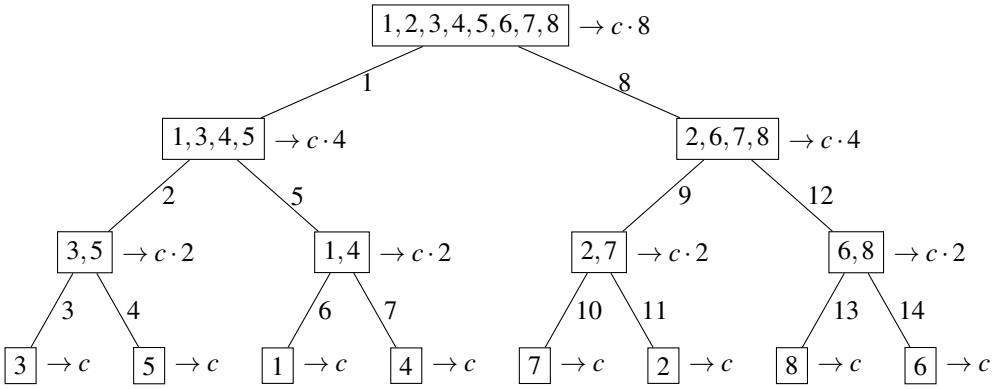
$$\left.\begin{array}{c} n_3 + n_7 = n+1+1, \\ n_4 + n_8 = n, \\ n_6 = n_{10} = n_{11} = n_{12} = 1, \\ n_{13} = n+1, \\ n_{14} = n \\ n_{15} + n_{18} = n \\ n_{16} + n_{19} = n. \end{array}\right\} \quad \begin{aligned} T_M(n) = & \sum_{k \in I} c_k n_k \; = \\ = \; & (c_3 + c_7)(n+2) + (c_4 + c_8)n + \\ & + c_6 + c_{10} + c_{11} + c_{12} + c_{13}(n+1) + \\ & + c_{14} n + (c_{15} + c_{18})n + (c_{16} + c_{19})n \; = \\ = \; & \Theta(n) \end{aligned}$$

Mogli bi pojednostaviti formulu ako dodamo da je $c_4 = c_8$, $c_{15} = c_{18}$, $c_{16} = c_{19}$, no, rezultat ostaje da je $T_M(n)$, vreme izvršavanja procedure MERGE(A, p, q, r), klase $\Theta(n)$, gde je $n = r - p + 1$.

U proceduri SORT (A, p, r) osim MERGE imamo još 4 komande koje se izvršavaju (najviše) jednom. Ne računajući rekurzivne pozive, vreme izvršavanja SORT (A, p, r) je $T_S(n) = \Theta(n)$, gde je $n = r - p + 1$.

Pri tome je nebitno kakav je sadržaj podniza od p do r, $T(n)$ će biti isto za svaki sadržaj podniza. Pošto se ∞ smatra veći ili jednak od svih elemenata, neće se komandom iz linije 15 ili 18 prebaciti džoker simbol ∞ u rezultujući podniz (osim ako je već bio u nizu A).

Pretpostavimo da je broj elemenata niza od p do r jednak 2^k. Onda će se moći deliti podniz na dva jednaka dela dok se ne dođe do jednog elementa. Tu počinje povratak iz rekurzije. Kad se to uradi i za suseda, onda nastupa MERGE: u početku spaja dva elementa, pa četiri, sve do $n = 2^k$.



Slika 3.2: MERGE SORT ([3,5,1,4,7,2,8,6]) za niz sa $n = 2^3 = 8$ elemenata

Neka je $n = 2^k$, $k \in N$ broj elemenata niza A kao na slici 3.2. Kad se pozove SORT (A, 1, 8), za $A = [3,5,1,4,7,2,8,6]$, rekurzivno će se pozivati SORT (A, 1, 4), SORT (A, 1, 2) i SORT (A, 1, 1). Tu se SORT vraća iz rekurzije, pa se poziva SORT (A, 2, 2), gde se isto vraća iz rekurzije. Onda će se, konačno, u okviru poziva SORT (A, 1, 2) izvršiti MERGE ([3,5,1,4,7,2,8,6], 1, 1, 2). Na slici je prikazan samo deo niza A od p do r posle poziva SORT (A, p, r) i vreme izvršavanja desno.

Pošto je $T_S(n) = \Theta(n)$, možemo ga aproksimirati sa $c \cdot n$. Vidimo na slici 3.2 da na svakom nivou rekurzije imamo ukupan zbir vremena izvršavanja SORT procedure $8 \cdot c$, odnosno, u opštem slučaju $2^k \cdot c = c \cdot n$. Pošto imamo $k+1 = \log_2 n + 1$ nivoa, vreme izvršavanja MERGE SORT (A) za niz A dužine $n = 2^k$ je $T(n) = (\log_2 n + 1) \cdot c \cdot n = c \cdot n \cdot \log_2 n + c \cdot n = \Theta(n \log n)$.

Ako n nije 2^k, $k \in N$, može se videti da je $T(n) = \Theta(m \log m)$, gde je $m = 2^k$, $k = \lceil \log_2 n \rceil$, odnosno prvi stepen dvojke veći on n. U tom slučaju, takođe važi $T(n) = \Theta(n \log n)$.

Na slici 3.2 na granama su navedeni redosledi rekurzivnih poziva procedure SORT.

MERGE SORT je još davne 1945. godine isprogramirao Fon Nojman na prvom na svetu elektronskom programabilnom računaru ENIAC. Algoritam i analiza MERGE SORT spajanjem odozdo prema gore su Goldstajn i Fon Nojman objavili u izveštaju o računarima 1948. godine.

Za rekurzivno pozivanje ovog programa je pogodno statički alocirati memoriju za globalne nizove L i R koju bi koristile sve instance u svim rekurzivnim pozivima procedure MERGE.

39. Zadatak: Koliko puta će biti pozvana procedura MERGE za sortiranje niza dužine $n = 2^k$?

40. Zadatak: Napisati hronološki pozive MERGE koji se izvrše pri pozivanju MERGE SORT ([5,2,4,6,1,3]).

Rešenje:

MERGE([5,2,4,6,1,3],1,1,2)

MERGE([2,5,4,6,1,3],1,2,3)

MERGE([2,4,5,6,1,3],4,4,5)

MERGE([2,4,5,1,6,3],4,5,6)

MERGE([2,4,5,1,3,6],1,3,6)

3.5 QUICK SORT

Ovo je često korišten algoritam sortiranja koji se pokazao brzim i jednostavnim.

Sortiranje se vrši divide & conquer[4] tehnikom, isto kao kod MERGE SORT.

Koristi se funkcija PARTITION koja grupiše elemente odabranog podniza (od p do r) premeštanjem, tako da jedan element (recimo poslednji) bude na svom mestu po redosledu, da ispred njega budu manji ili jednaki od njega, iza njega veći od njega osobina ($\star$).

Redni broj elementa koji zadovolja ($\star$) vraća funkcija PARTITION. U nekim varijantama ne postoji granični element već se elementi dele na manje i veće. Tada se vraća redni broj prvog elementa grupe većih od posmatranog (ili poslednjeg elementa grupe manjih), osobina ($\star\star$).

Neka je funkcija PARTITION vratila q, redni broj odabranog elementa. Onda se rekurzivno poziva ista procedura za podnizove od p do $q-1$ i od $q+1$ do r, gde su p i r granice posmatranog podniza. Prvi put se uzima $p = 1$ i $r = n$. Povratak iz rekurzije je kad se pozove SORT (A, p, r), gde je $p \geq r$. Vidi procedure SORT i QUICK SORT.

```
procedure SORT(A, p, r)
    ▷ Procedura koja se rekurzivno poziva
    if p < r then
        q ← PARTITION(A, p, r)
        SORT(A, p, q − 1)
        SORT(A, q + 1, r)
    end if
end procedure
procedure QUICK SORT(A)
    ▷ Glavna procedura koju poziva korisnik
    n ← length (A)
    SORT(A, 1, n)
end procedure
```

QUICK SORT je razvio Toni Hoar[5] 1959. godine, objavljen je 1961. Kada se dobro implementira, u proseku je brži od MERGE SORT. QUICK SORT vrši **sortiranje u mestu**: zamenom elemenata niza, bez prebacivanja elemenata niza u nove nizove kao kod MERGE SORT.

[4] *divide & conquer* - podeli pa osvoji, EN

[5] Tony Hoare, britanski informatičar, 1934 -

Implementacija funkcije PARTITION može da se uradi na više načina. Mi ćemo dati Lomuto shemu, u kojoj se za izabrani element uzima poslednji u zadatom podnizu (r-ti).

```
function PARTITION(A, p, r)
    ▷ zamenama srediti niz u skladu sa (⋆)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1 do
        if A[j] ≤ x then
            i ← i + 1
            exchange(A, i, j)
        end if
    end for
    exchange(A, i, r)
    return i + 1
end function
```

Ova verzija funkcije PARTITION primenjena u QUICK SORT ima asimptotski **Worst case** za već **sortiran niz**. Naime, za sortiran niz funkcija PARTITION će se pozivati sa $p = 1$ i, redom, $r = n, n-1, \ldots, 2$. Posle svakog izvođenja partition u prvom sledećem rekurzivnom pozivu SORT biće redom $n-2, n-3, \ldots, 1$ poređenja. To daje ukupan broj poređenja, a samim tim i ukupno vreme izvršavanja QUICK SORT, $T(n) = \Theta(n^2)$.

Procedura exchange prvo utvrđuje da ako se traži zamena dva ista elementa - ne treba vršiti zamenu (swap). Njenim korišćenjem, ako je niz sortiran, skraćuje se ukupno vreme. To ne bi bio slučaj sa obrnuto sortiranim nizom, koji u ovoj verziji (Lomuto shema) takođe ima vreme izvršavanja $T(n) = \Theta(n^2)$, ali ima puno zamena (swap) unutar exchange.

41. Zadatak: Napisati funkciju PARTITION koja koristi dva brojača: sa početka i kraja podniza koji se sortira. Oni se pomeraju jedan prema drugom, poredeći se sa srednjim elementom, dok ne otkriju inverziju. Ako je u tom trenutku prvi brojač još uvek ispred drugog, ovi elementi su u pogrešnom redosledu - treba swap. Kada se na kraju ukrste brojači, razmena se ne vrši, pronađen je element sa rednim brojem q tako da je zadovoljena osobina ($\star\star$). (Hoare shema)

Za svaku verziju funkcije PARTITION postoji Worst case niz dužine n.

Postoji mogućnost randomizovanja (permutovanja na slučajan način) redosleda elemenata niza koja bi učinila ovaj algoritam u proseku podjednako kompleksnim za sve ulaze iste dužine.

3.6 Sortiranje indeksa i stabilnost algoritama za sortiranje

U praksi se sortiranje najčešće koristi za slogove (records) koji sadrže više zapisa. Na primer, kada se vrste u spreadsheet tabeli sa više kolona sortiraju po jednoj koloni. Elemente kolone po kojoj se sortira nazivamo **ključevi**. Ključeve možemo formirati i spajanjem dva ili više ključa.

3.6.1 Sortiranje stringova

Programski jezik C ima osnovni tip podataka karakter (char) koji zauzima jedan bajt. Nizovi karaktera su stringovi, njima se predstavljaju reči (engleskog jezika). Reči mogu biti različitih dužina, zato, radi lakšeg baratanja, C kompajleri na kraj niza karaktera stavljaju NUL[6] karakter. Na taj način se ne mora pamtiti dužina stringa, string se pamti kao adresa prvog (nultog) karaktera.

Karakter sa manjim rednim brojem prethodi karakteru sa većim rednim brojem. U ASCII standardu karakteri koji predstavljaju brojeve prethode velikim slovima koja prethode malim slovima. Postoji standardna C biblioteka za rad sa nul–terminated[7] stringovima (#include <string.h>).

Leksikografski, za proveru prethođenja dva stringa preskaču se identični karakteri i posmatra prvi različit. Kraći string čiji je sadržaj jednak početku dužeg stringa prethodi dužem. Provera **relacije prethođenja** (u leksikografskom redosledu) se postiže poređenjem redom karaktera kao brojeva od 0 do 255 i dogovorom da se NUL karakter ne koristi u stringovima. Na taj način nul–terminated stringovi omogućavaju lakše određivanje prethodnika.

42. Zadatak: Napisati u programskom jeziku C funkciju zcmp koja za ulaz ima dva stringa s1 i s2 (adrese) i maksimalnu dužinu stringa n, a kao rezultat daje: negativan broj - ako prvi string prethodi drugom, nulu - ako su prvih n karaktera jednaki, pozitivan broj - ako drugi string prethodi prvom.

Levo je dat program za testiranje sa prototipom funkcije zcmp.

```
#include <stdio.h>
#include <stdlib.h>
#define maxn 30

// Ovaj program koristiti za testiranje.
// Izlaz: ab = -97, ac = 17, bc = 17.

int zcmp(char*, char*, size_t);

void main (){
   char a[] = "Petr";
   char b[] = "Petra";
   char c[] = "Petar";
   printf("ab = %d, ", zcmp(a,b,maxn));
   printf("ac = %d, ", zcmp(a,c,maxn));
   printf("bc = %d. \n", zcmp(b,c,maxn));
}
```

Rešenje: Broj koji će funkcija vratiti je razlika rednih brojeva u ASCII kodu prvih karaktera koji se razlikuju na adresama s1 i s2. Nula se vraća ako su n karaktera jednaki.

Ova funkcija je implementirana u standardnoj biblioteci string kao strncmp.

```
int zcmp(char *s1, char *s2, size_t n){
    unsigned int i = 0;
    int y = 0;
    while(i<n &&
          !(y=s1[i]-s2[i]) && s1[i]){i++;};
    return y;
}
```

43. Zadatak: Dokazati da za relaciju prethođenja stringova važi tranzitivnost.

Redosled stringova iz rezultata primene zcmp funkcije (ili strncmp) u programu za testiranje je: ab = -97 $\Rightarrow$ "Petr" prethodi "Petra", ac = 17 $\Rightarrow$ "Petar" prethodi "Petr". Posledica tranzitivnosti je: "Petar" prethodi "Petra", što se i vidi iz bc = 17.

Zaključak: redosled je "Petar", "Petr", "Petra" (c, a, b).

Vidimo da funkciju zcmp odnosno strncmp, ako smo u pretprocesorskim direktivama imali #include <string.h>, možemo koristiti za sortiranje stringova koji sadrže reči engleskog jezika. Na-

[6] NUL = \0. Bajt sadržaja 0. 1 char = 1 byte = 8 bits = 2^8 = 256 permutacija, od \0 do \255.

[7] nul–terminated, stringovi koji završavaju NUL karakterom, EN

ime, ako imamo niz A (adresa) nul–terminated stringova, umesto logičke operacije $A[i] \leq A[j]$ koristimo strcmp $(A[i], A[j]) \leq 0$. C funkcija strcmp radi isto što i strncmp, ali bez argumenta za maksimalnu dužinu stringa.

3.6.2 Lokalizacija

Redosled zapisa karaktera u kompjuteru ne mora odgovarati redosledu reči (stringova) nekog jezika. Recimo, postoje digrafi, glasovi zapisani pomoću dva karaktera, koji utiču na redosled reči nekog jezika. U našoj latinici to su dž, lj, nj.

Unicode standard je predvideo za većinu jezika i alfabeta način zapisivanja reči u stringove i definisao njihovo leksikografsko prethođenje. Korišćenjem UTF-8 očuvana je kompatibilnost sa 7-bitnim ASCII standardom, ali se izgubila jednoznačna korespodencija: 1 kompjutersko slovo (karakter) $\leftrightarrow$ 1 znak alfabeta (slovo). U string biblioteci u C-u postoji funkcija strcoll koja slično funkciji strcmp vraća negativan ili pozitivan broj ili nulu ako prvi string prethodi drugom ili drugi prethodi prvom ili su jednaki. (Vidi zadatak 42.)

Da bi u kompjuterskom programu odredili standard za poređenje prethođenja stringova, u operativnom sistemu ili okruženju u kome se koristi program se podešavaju globalne promenljive koje daju informacije o šest informacija lokalizacije (vidi tabelu 3.1). Moguće je pojedinačno podešavanje, a postoji mogućnost da se svih šest informacija lokalizacije postavi na istu grupu lokalnih simbola. Za to se koristi globalna promenljiva LC_ALL.

Format informacije za lokalizaciju je:

```
language[_territory][.codeset][@modifier].
```

Na primer: `sr_RS.UTF-8@latin` je oznaka za srpski jezik, u Republici Srbiji, kodiran u UTF-8 u latiničnoj varijanti.

```c
#include <stdio.h>
#include <locale.h>                    // Biblioteka za lokalizaciju
#include <string.h>

void main (){
    char a[] = "njujork";
    char b[] = "nujork";
    char new_locale[] = "sr_RS";
    setlocale (LC_ALL, new_locale);    // Srpski jezik

    printf("strcoll(njujork,nujork) = %d\n", strcoll(a,b));
    printf("strcmp(njujork,nujork) = %d\n", strcmp(a,b));
}
```

Prethodni C program komandom setlocale podešava svih šest informacija po pravilima srpskog jezika. Ako se to ne uradi, podrazumevana lokalizacija je C standard. Izlaz ovog programa će biti:

```
ab = 1
abz = -11
```

To znači da po ASCII standardu slovo j prethodi slovu u, pa stoga i "njujork" prethodi "nujork" (abz = -11). Po srpskoj latinici slovo n prethodi slovu (digrafu) nj pa "nujork" prethodi "njujork" (ab = 1).

LC_COLLATE	Definiše redosled poređenja stringova
LC_CTYPE	Definiše klasifikaciju karaktera, velika - mala slova
LC_MESSAGES	Definiše kako se kaže DA i NE
LC_MONETARY	Definiše pravila za pisanje cena i oznaku valute
LC_NUMERIC	Definiše pravila za pisanje brojeva
LC_TIME	Pravila i simboli za datum i vreme

Tabela 3.1: Šest informacija lokalizacije

Posmatrajmo tabelu studenata sa dva odseka koji su polagali ispite i dobili ocene:

rbr	mat. k.	god.	brind	prezime	ime	pol	ocena	datum
1	SW	2015	033	Petrović	Aleksandar	m	8	20210620
2	SW	2016	102	Nikolić	Marko	m	7	20210703
3	SW	2016	050	Milovanović	Vanja	m	10	20210620
4	SW	2017	004	Aleksić	Dunja	ž	5	20210620
5	SW	2017	052	Petrović	Aleksandra	ž	9	20210620
6	SW	2018	008	Milenović	Slavko	m	6	20220703
7	AI	2012	048	Milovanović	Vanja	ž	6	20210622
8	AI	2013	035	Petrović	Nataša	ž	7	20210704
9	AI	2014	044	Žunić	Milica	ž	5	20210622
10	AI	2015	006	Janković	Milija	m	7	20210622
11	AI	2016	158	Stošić	Milica	ž	8	20210704

Vidimo da su vrste tabele sortirane po prvoj koloni (rbr) ako se uzima da je njen sadržaj ceo broj. Ako bi se sadržaj te kolone posmatrao kao string, sortiranje bilo drugačije: 1, 10, 11, 2, . . . , 9.

Cifre su u ASCII (i UTF-8) kodu od rednog broja 48 pa nadalje (0, 1, . . . , 9). Ako želimo da sortiranje stringova koristimo za sortiranje brojeva treba ih dopuniti početnim nulama tako da svi sadrže isti broj karaktera, kao što je u koloni brind.

Slično je i sa datumima: ako želimo da budu hronološki složeni, treba da ih pišemo u formatu GGGGMMDD (godina, dan, mesec), kao u koloni datum.

Najčešće sortiranje imena (telefonski imenik) je "prezime ime" po leksikografskom redosledu. U nekim jezicima ime se sastoji od nekoliko reči. Obično je poslednja reč prezime, a ostale ime. Neke nacije imaju srednje ime (middle name). U našem jeziku srednje ime obično bude ime ili početno slovo imena jednog od roditelja. Ako se želi napraviti string za sortiranje po prezimenu, onda se, obično, prezime prebaci na početak i odvoji zarezom od ostatka.

Stranci koji koriste naša dokumenta u polje prezime stavljaju svoje last name ili family name[8], a ostala imena u polje ime.

DEFINICIJA 5 *Kažemo da je algoritam za sortiranje **stabilan** ako elemente (vrste) sa istom vrednošću ključa ne invertuje.*

MERGE SORT je stabilan algoritam. Naime, u liniji 14 algoritma MERGE u slučaju kad se pojave

[8] last = poslednje, family = porodično, name = ime, EN

isti ključevi, zbog znaka $\leq$ prednost se daje elementu iz levog podniza, dakle sa manjim indeksom, pa ne može doći do invertovanja.

Ako imamo odvojena polja za prezime i ime, a želimo da sortiramo po pravilu telefonskog imenika, možemo, koristeći stabilan algoritam za sortiranja prvo sortirati po koloni ime a potom po koloni prezime.

44. Zadatak: Umesto spajanja indeksa *mat. k.* + *god.* + *brind*, kako se može postići željeno sortiranje sortiranjem po jednom indeksu?

Rešenje: Prvo se sortiraju po brind sve vrste, zatim stabilnim algoritmom za sortiranje po indeksu god. i na kraju stabilnim algoritmom za sortiranje po indeksu mat. k.

45. Zadatak: Koji su još od do sad obrađivanih algoritama stabilni? Objasniti.

3.7 Poređenje algoritama za sortiranje

Ovde ćemo analizirati red vremena izvršavanja, red potrošnje memorije i stabilnost algoritama za sortiranje datih u ovoj glavi.

U sledećoj tabeli je data asimptotska vrednost vremena izvršavanja za ulaz veličine n za sledeće algoritme: BUBBLE SORT (B), INSERTION SORT (I), SELECTION SORT (S), MERGE SORT (M), QUICK SORT (Q).

Oznake su: Best case: B, Average case: A i Worst case: W. Takođe je dat podatak o redu veličine dodatnog memorijskog prostora potrebnog za sortiranje (M) i podatak o stabilnosti posmatranog algoritma (S).

	B	A	W	M	S
B	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	DA
I	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	DA
S	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	NE
M	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n)$	DA
Q	$\Theta(n \ln n)$	$\Theta(n \ln n)$	$\Theta(n^2)$	$\Theta(\ln n)$	NE

Tabela 3.2: Asimptotska efikasnost algoritama za sortiranje

Stabilnost je korisna osobina algoritama za sortiranje. Ako se stabilan algoritam primeni na podatke prethodno sortirane po jednom ključu, on očuvava redosled sortiranja kod istih ključeva novog sortiranja.

Naša implementacija QUICK SORT algoritma nije stabilna. Ako se ovaj algoritam modifikuje sa ciljem da dobije stabilnost, gube se performanse.

U praksi, primenjen na uobičajene podatke, QUICK SORT je najbrži od posmatranih algoritama.

Algoritme iz prethodnih zadataka smo testirali na nizu random[9] brojeva obima $10^1, 10^2, 10^3, 10^4, 10^5$. Mereno je vreme potrebno da se niz sortira algoritmom kodiranim u C-u na relativno sporom kompjuteru.

n	B	I	S	M	Q
10	7.30E-05	6.08E-05	6.10E-05	0.00034	0.0003
100	0.00021	6.10E-05	0.0001	0.00235	0.0008
1000	0.01873	0.00417	0.00625	0.0233	0.0067
10000	1.92922	0.35455	0.60198	0.23819	0.0669
100000	192.796	35.4994	60.2424	2.38437	0.6754

Tabela 3.3: Vreme $[s]$ sortiranja niza dužine n aloritmima B, I, S, M, Q

Vidimo da je QUICK SORT najbrži za veliko n iako je na nizu do veličine 1000 brži INSERTION

[9] *random* = slučajni, EN

SORT. U praksi se za nizove obima manjeg od neke granice u rekurziji sa QUICK SORTA prelazi na INSERTION SORT. Time se štedi memorija i vreme, jer je INSERTION SORT brži za male nizove i troši manje memorije.

Glava 4

Pretraživanje

46. Zadatak: Napisati algoritam koji za dati niz nalazi najmanji i najveći element. Odrediti red broja poređenja u datom algoritmu (prema veličini niza n).

Rešenje je program MINIMAXI.

47. Zadatak: Napisati algoritam koji za dati niz nalazi najmanji i najveći element a da pri tome ima najviše $1.5n$ poređenja, gde je n veličina niza.

Rešenje je program MINIMAXI1.

```
1:  function MINIMAXI(A)
2:      n ← length(A)
3:      min ← A[1]
4:      max ← A[1]
5:      for i ← 2 to n do
6:          if A[i] < min then
7:              min ← A[i]
8:          end if
9:          if A[i] > max then
10:             max ← A[i]
11:         end if
12:     end for
13:     return [min, max]
14: end function
```

Očigledno je broj poređenja (linije 6 i 9): $n-1+n-1 = 2n-2 = O(n)$ (sa $c_1 = 2, n_0 = 1$).

Odnosno: za svaki element (osim prvog, a to nije bitno) se vrši po dva poređenja.

Objašnjenje zadatka 47:

Za svaka dva elementa (osim prvog za neparno n) se vrši 3 poređenja, što daje ukupni broj poređenja $\frac{3}{2}n = 1.5n$ za n parno, A ukupni broj poređenja je i manji za n neparno.

Odd[1] funkcija vraća true ako je n neparan broj.

```
1:  function MINIMAXI1(A)
2:      n ← length(A)
3:      if odd(n) then
4:          max ← A[1]
5:          min ← A[1]
6:          j ← 2
7:      else
8:          if A[1] > A[2] then
9:              max ← A[1]
10:             min ← A[2]
11:         else
12:             min ← A[1]
13:             max ← A[2]
14:         end if
15:         j ← 3
16:     end if
17:     while j < n do
18:         if A[j] > A[j+1] then
19:             if A[j] > max then
20:                 max ← A[j]
21:             end if
22:             if A[j+1] < min then
23:                 min ← A[j+1]
24:             end if
25:         else
26:             if A[j] < min then
27:                 min ← A[j]
28:             end if
29:             if A[j+1] > max then
30:                 max ← A[j+1]
31:             end if
32:         end if
33:         j ← j+2
34:     end while
35:     return [min, max]
36: end function
```

[1] *odd* = neparno, EN; *even* = parno, EN

Glava 5

Apstraktni tipovi podataka

5.1 Matrice

48. Zadatak: Napisati algoritam DET za računanje determinante matrice formata $n \times n$ dovođenjem na gornje-trougaonu determinantu.

```
1:  function DET(A, n)
2:      znak ← 1
3:      for i ← 1 to n − 1 do
4:          pm ← PIVOT(A, n, i)
5:          if ¬pm then
6:              return 0.0
7:          end if
8:          znak ← znak ∗ pm
9:          for k ← i + 1 to n do
10:             α ← A[k, i]/A[i, i]
11:             A[k, i] ← 0.0
12:             for j ← i + 1 to n do
13:                 A[k, j] ← A[k, j] − αA[i, j]
14:             end for
15:         end for
16:     end for
17:     if A[n, n] = 0 then
18:         return 0.0
19:     end if
20:     d ← znak ∗ A[1, 1]
21:     for i ← 2 to n do
22:         d ← d ∗ A[i, i]
23:     end for
24:     return d
25: end function
```

```
function PIVOT(A, n, m)
    i1 ← m; j1 ← m; pm ← 1
    for i ← m to n do
        for j ← m to n do
            if |A[i, j]| > |A[i1, j1]| then
                i1 ← i; j1 ← j
            end if
        end for
    end for
    if A[i1, j1] = 0 then
        return 0
    end if
    if i1 ≠ m then
        pm ← pm ∗ (−1)
        for j ← m to n do
            swap(A[i, j], A[i1, j])
        end for
    end if
    if j1 ≠ m then
        pm ← pm ∗ (−1)
        for i ← 1 to n do
            swap(A[i, j], A[i, j1])
        end for
    end if
    return pm
end function
```

49. Zadatak: Napraviti biblioteku funkcija u programskom jeziku C koje omogućavaju računske operacije sa matricama.

Elementi matrice $A_{m\times n}$ se smeštaju po vrstama u niz dužine $m*n$. Indeksiranje elemata u C-u počinje od 0, tako da element matrice $A[i,j]$ se u nizu nalazi na mestu $A[i*n+j]$.

matrice.h

```
void addmat(double *, double *, double *, int, int);          // sabiranje (I,I,O,I,I)
void multmat(double *, double *, double *, int, int, int);    // mnozenje (I,I,O,I,I,I)
void multscal(double, double *, double *, int, int);          // mnozenje skalarom (I,I,O,I,I)
void transpose(double*, double *, int, int);                  // transponovanje (I,O,I,I)
int inverse(double*, double *, int);                          // inverzna (I/O,O,I)
double det(double*, int);                                     // determinanta (I/O,I)
int printmatrix(double*, int, int);                           // stampanje (I,I,I)
```

50. Zadatak: Koristeći biblioteku iz zadatka 49 napisati program u C-u koji rešava matričnu jednačinu $A+BX=C$, gde su

$$A=\begin{bmatrix}1 & 2 & -3\\ 4 & -7 & -16\\ -7 & -18 & -16\end{bmatrix},\ B=\begin{bmatrix}1 & 2 & 3\\ 4 & 7 & 6\\ 7 & 8 & -16\end{bmatrix},\ C=\begin{bmatrix}-1 & 3 & 4\\ 2 & 2 & 10\\ 16 & 20 & 21\end{bmatrix}.$$

Rešenje: Primenjujući matričnu algebru dobijamo $X=B^{-1}(C-A)$. U C-u to računamo pomoću sledećeg programa

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrice.h"
#define epsilon 1e-12

int main()
{
    int nr,n;

    n = 3;
    double A[]={1,2,-3,4,-7,-16,-7,-18,-16};
    double B[]={1,2,3,4,7,6,7,8,-16};
    double C[]={-1,3,4,2,2,10,16,20,21};
    double *minusA=malloc(n*n*sizeof(double));
    double *CminusA=malloc(n*n*sizeof(double));
    double *Binv=malloc(n*n*sizeof(double));
    double *X=malloc(n*n*sizeof(double));

    if((nr=inverse(B,Binv,n))){
        printf("\nNe postoji B^(-1), rang(B) je %d.",n-nr);
    }
    else{
        multscal(-1,A,minusA,3,3);
        addmat(C,minusA,CminusA,3,3);
        multmat(Binv,CminusA,X,3,3,3);
        printmatrix(X,3,3);
    }
    return 0;
}
```

Kompajliranje i pokretanje gornjeg programa main.c daje rešenje $X =$

```
+-                            -+
|     1.00     2.00     3.00  |
|     0.00     1.00     2.00  |
|    -1.00    -1.00    -0.00  |
+-                            -+

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

51. Zadatak: Analizirati algoritam za računanje inverzne matrice inverse iz zadatka 49.

52. Zadatak: Izračunati broj sabiranja i množenja elemenata matrice A prilikom računanja determinante reda n upotrebom algoritma iz zadatka 48.

Rešenje: U pivotizaciji nema množenja i sabiranja elemenata matrice A. Dovođenje na gornju trougaonu determinantu se vrši Gausovim eliminacijama:

Idući brojačem i po dijagonali od prvog do pretposlednjeg elementa, množenjem i-te vrste brojem $\alpha = A[k,i]/A[i,i]$ i oduzimanje od k-te vrste dobija se gornje trougaona determinanta. Njena vrednost je proizvod elemenata sa glavne dijagonale.

Sabiranja elemenata matrice se vrše u liniji 13 algoritma (oduzimanje brojimo kao sabiranje). Množenja se vrše u liniji 10 (deljenje brojimo kao množenje) i liniji 13, kao i na samom kraju, u liniji 22, kad se množe elementi sa glavne dijagonale.

i	$SAB(i)$	$MNO(i)$
1	$(n-1)(n-1)$	$(n-1)(n-1)+n-1$
2	$(n-2)(n-2)$	$(n-2)(n-2)+n-2$
$\vdots$	$\ddots$	$\cdots$
$n-2$	$2\cdot 2$	$2\cdot 2+2$
$n-1$	1	$1+1$
linija 22	0	$n-1$
$\sum$	$\sum SAB(i)$	$\sum MNO(i)$

Koristeći formulu $\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1)$ i $\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$, dobijamo broj sabiranja i množenja:

$$\sum SAB(i) = \frac{1}{6}(n-1)n(2n+1) = \Theta(n^3)$$

$$\sum MNO(i) = \sum SAB(i) + \frac{1}{2}(n-1)n + n - 1 = \Theta(n^3)$$

53. Zadatak: Izračunati broj sabiranja i množenja elemenata matrice A prilikom računanja inverzne matrice reda n Gaus Žordanovom eliminacijom.

54. Zadatak: Izračunati broj sabiranja i množenja elemenata matrice A prilikom računanja determinante reda n koristeći definiciju:

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = \sum_{\substack{\text{po svim perm.} \\ (i_1,i_2,\ldots i_n)}} (-1)^{\sigma(i_1,i_2,\ldots i_n)} a_{1,i_1} a_{2,i_2} \cdots a_{n,i_n}$$

gde je $\sigma(i_1,i_2,\ldots i_n)$ broj inverzija permutacije $(i_1,i_2,\ldots i_n)$, čija parnost odlučuje znak sabirka.

Rešenje: Očigledno ima $n!$ sabiraka, tako da je za ovaj algoritam

$$\sum SAB(i) = n! - 1$$

$$\sum MNO(i) = n!(n-1)$$

55. Zadatak: Izmeriti na vašem kompjuteru vreme potrebno za izvršavanje jednog sabiranja i jednog množenja u for petlji veličine 10^5.

56. Zadatak: Ako jedno sabiranje traje 3.13E-9, a množenje 3.75E-9, koliko vremena treba da se saberu i pomnože elementi matrice 100×100 pomoću algoritma iz zadatka 48, čija analiza je u zadatku 52, odnosno, preko definicije, analiza u zadatku 54?

Rešenje: Vreme dobijamo po formuli

$$\sum SAB(i) \times 3.13 \times 10^{-9} + \sum MNO(i) \times 3.75 \times 10^{-9}.$$

Za $n = 100$, formule iz zadatka 52 i 54 daju:

	vreme
zad 52	$0.0023s$
zad 54	$3.49 \times 10^{151} s =$ 1.11×10^{144}godina

Očigledno je računanje determinante dovođenjem na gornju trougaonu neuporedivo brže za velike formate matrice.

Štaviše, zbog manjeg broja računskih operacija nagomilavanje greške zaokruživanja je manje i dobija se precizniji rezultat.

Preciznosti rezultata doprinosi i pivotizacija. Dovođenje najvećeg elementa po apsolutnoj vrednosti na dijagonalu utiče na znatno smanjivanje uticaja greške zaokruživanja.

Pivotizacija je istovremeno i test da li je data matrica singularna: kad se procedura rowpivot vrati sa vrednošću nr različitom od 0, onda je matrica singularna, njena determinanta je 0, a rang je $n - nr$.

Posmatrani algoritam prvo vrši Gausove eliminacije na matrici A i istovremeno na matrici *B* koja je u početku bila jedinična matrica.

Pivotizaciju vrši samo po vrstama. Akko je matrica singularna, onda će pri izvršavanju Gausovih eliminacija rowpivot vratiti -1, što će biti signal proceduri inverse da zaključi da je matrica singularna, a da je njen rang trenutna vrednost brojača *i*.

Naravno, isto kao u implementaciji algoritma za računanje determinante u *floating point*[1] aritmetici ne vrši se poređenje po jednakosti dobijene vrednosti sa nulom. Razlog je što mala greška zaokruživanja, koja je neminovna, dovodi do pogrešnog zaključka. Stoga nulom smatramo brojeve čija je apsolutna vrednost manja od epsilon = 10^{-12}.

Algoritam potom vrši Žordanove eliminacije na matrici *A* i *B*, dovodeći elementarnim transformacijama po vrstama matricu *A* na jediničnu i matricu *B* na inverznu od *A*.

Žordanove transformacije se ne moraju vršiti na elementima matrice *A*.

[1] *floating point* = aritmetika pokretnog zareza, EN