Osnove programiranja
kroz programski jezik C – Deo II

Predrag Janičić i Filip Marić

26. januar 2015

Sadržaj

Deo I

Osnove razvoja softvera

Glava 1

Životni ciklus razvoja softvera

Pod *razvojem softvera* često se ne misli samo na neposredno pisanje programa, već i na procese koji mu prethode i slede. U tom, širem smislu razvoj softvera se naziva i *životni ciklus razvoja softvera*. Razvoj softvera se razlikuje od slučaja do slučaja, ali u nekoj formi obično ima sledeće faze i podfaze:

Planiranje: Ova faza obuhvata prikupljanje i analizu zahteva od naručioca softvera, razrešavanje nepotpunih, višesmislenih ili kontradiktornih zahteva i kreiranje precizne specifikacije problema i dizajna softverskog rešenja. Podfaze ove faze su:

- Analiza i specifikovanje problema (obično je sprovodi analitičar, koji nije nužno informatičar, ali mora da poznaje relevantne poslovne ili druge procese);
- Modelovanje rešenja (obično je sprovodi projektant, koji mora da razume i specifikaciju problema i da je u stanju da izabere adekvatna softverska rešenja, npr. programski jezik, bazu podataka, relevantne biblioteke, strukture podataka, matematičku reprezentaciju problema, algoritamska rešenja, itd);
- Dizajn softverskog rešenja (sprovode je programeri).

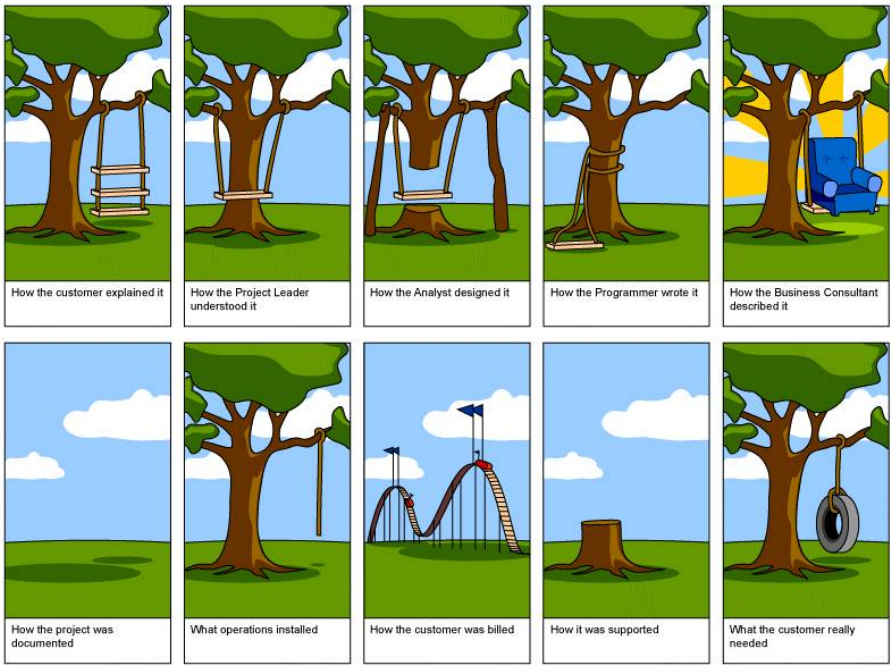
Realizacija: Ova faza obuhvata implementiranje dizajniranog softverskog rešenja u nekom konkretnom programskom jeziku. Implementacija treba da sledi opšte preporuke, kao i preporuke specifične za realizatora ili za konkretan projekat. Pouzdanost i upotrebljivost softverskog proizvoda proverava se analizom efikasnosti i ispravnosti, a za naručioca se priprema i dokumentacija. Podfaze ove faze su:

- Implementiranje (kodiranje, pisanje programa) (o nekim aspektima ove faze govori glava 2);
- Analiza efikasnosti i ispravnosti (o nekim aspektima ovih faza govore redom glava 3 i glava 4);
- Izrada dokumentacije (obično korisničke dokumentacije – koja opisuje korišćenje programa i tehničke dokumentacije — dokumentacije koja opisuje izvorni kôd);

Eksploatacija: Ova faza počinje nakon što je ispravnost softvera adekvatno proverena i nakon što je softver odobren za upotrebu. Puštanje u rad uključuje instaliranje, podešavanja u skladu sa specifičnim potrebama i zahtevima korisnika ali i testiranje u realnom okuženju i sveukupnu evaluaciju sistema. Organizuje se obuka za osnovne i napredne korisnike i obezbeđuje održavanje kroz koje se ispravljaju greške ili dodaju nove manje funkcionalnosti. U održavanje se obično uloži više od tri četvrtine ukupnog rada u čitavom životnom ciklusu softvera. Podfaze ove faze su:

- Obuka i tehnička podrška;
- Puštanje u rad;
- Održavanje.

Postoje i međunarodni standardi, kao što je ISO/IEC 12207 i ISO/IEC 15504, koji opisuju životni ciklus softvera, kros precizno opisane postupke izbora, implementacije i nadgledanja razvoja softvera. Kvalitet razvijenog posla često se ocenjuje prema nivou usklađenosti sa ovim standardima.



Slika 1.1: Faze razvoja softvera ilustrovane na šaljiv način

Faze razvoja softvera i moguće probleme na šaljiv način ilustruje čuvena karikatura prikazana na slici 1.1.

Za razvoj softvera relevantni su i procesi istraživanja tržišta, nabavke softvera, naručivanja softvera, razmatranje ponuda i slični, ali u ovom tekstu će biti reči samo o razvoju softvera od faze planiranja. Faza planiranja će biti ukratko opisana u nastavku ovog poglavlja, dok će faza realizacije i njene podfaze biti opisane detaljnije u narednim glavama.

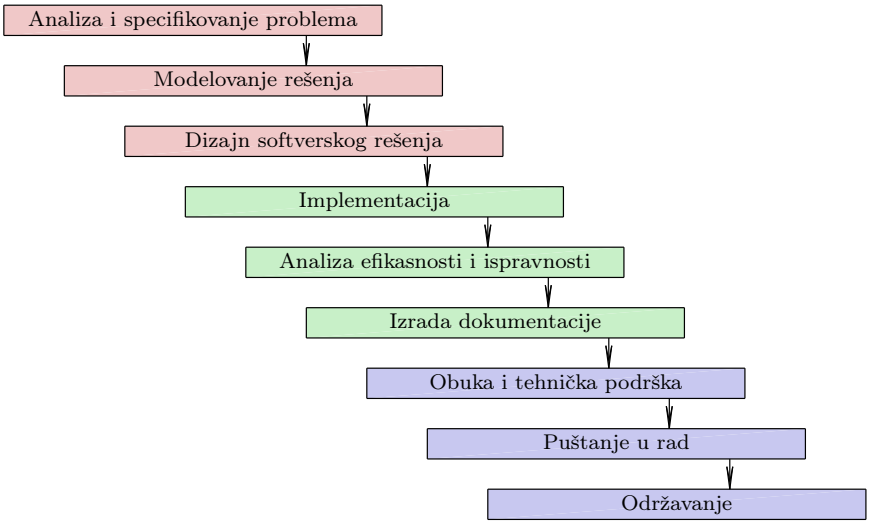
1.1 Metodologije razvoja softvera

I u teoriji i u praksi postoje mnoge metodologije razvoja softvera. U praksi su one često pomešane i često je teško razvrstati stvarne projekte u postojeće metodologije. U nastavku je opisano nekoliko bitnih metodologija i ideja na kojima su zasnovane.

Metodologija vodopada. U strogoj varijanti metodologije vodopada (eng. waterfall), na sledeću fazu u razvoju softvera prelazi se tek kada je jedna kompletno završena. Ova metodologija ilustrovana je slikom 1.2. Metodologija se smatra primenljivom ako su ispunjeni sledeći uslovi:

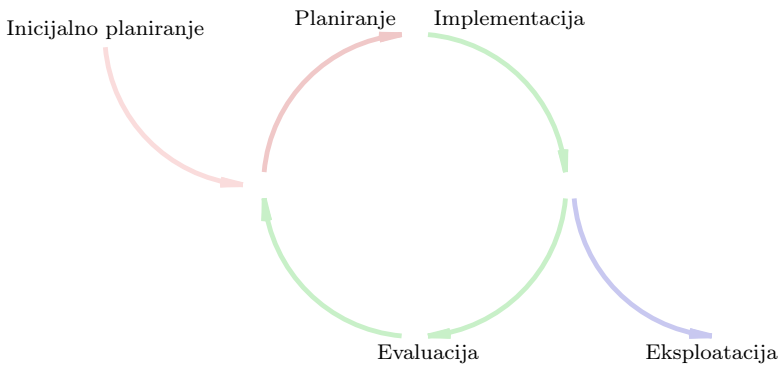
- svi zahtevi poznati su unapred;
- zahtevi nemaju nerazrešene, potencijalno rizične faktore (npr. rizike koji se odnose na cenu, tempo rada, efikasnost, bezbednost, itd);
- priroda zahteva se ne menja bitno u toku razvoja;
- zahtevi su u skladu sa očekivanjima svih relevantnih strana (investitori, korisnici, realizatori, itd.);
- pogodna arhitektura rešenja može biti opisana i podrobno shvaćena;
- na raspolaganju je dovoljno vremena za rad u etapama.

Ovaj model ne predviđa modifikovanje prethodnih faza jednom kada su završene i ova osobina je predmet najčešćih kritika. Naime, izrada aplikacija često traje toliko dugo da se zahtevi promene u međuvremenu i završni proizvod više nije sasvim adekvatan a ponekad ni uopšte upotrebljiv.



Slika 1.2: Ilustracija za metodologiju vodopada

Metodologija iterativnog i inkrementalnog razvoja. U ovoj metodologiji, dizajn se opisuje u iteracijama i projekat se gradi inkrementalno. Iteracije obično donose više detalja i funkcionalnosti, a inkrementalnost podrazumeva dodavanje jednog po jednog modula, pri čemu i oni mogu biti modifikovani ubuduće. U jednom trenutku, više različitih faza životnog ciklusa softvera može biti u toku. U ovoj metodologiji vraćanje unazad je moguće. Ova metodologija ilustrovana je slikom 1.3.



Slika 1.3: Ilustracija za iterativnu metodologiju

Metodologija rapidnog razvoja. U ovoj metodologiji (engl. rapid application development), faza planiranja je svedena na minimum zarad brzog dobijanja prototipova u iteracijama. Faza planiranja je pomešana sa fazom implementacije što olakšava izmene zahteva u hodu i brže implementiranje. Proces razvoja kreće sa razvojem preliminarnog modela podataka i algoritama, razvijaju se prototipovi na osnovu kojih se definišu, preciziraju ili potvrđuju ili zahtevi naručioca ili korisnika. Ovaj postupak se ponavlja iterativno, sve do završnog proizvoda.

Slaba strana ove metodologije je u tome što može dovesti do niza prototipova koji nikada ne dostižu do zadovoljavajuće finalne aplikacije. Čest izvor takvih problema su grafički korisnički interfejsi (engl. graphical user interface; GUI). Naime, korisnici napredak u razvoju aplikacije doživljavaju prvenstveno kroz napredak grafičkog korisničkog interfejsa. To podstiče programere, pa i vođe projekata, da se u prototipovima usredsređuju na detalje grafičkog interfejsa umesto na druge

segmente aplikacije (kao što su, na primer, poslovni procesi i obrada podataka). Čak i mnogi razvojni alati privilegovano mesto u razvoju softvera daju razvoju grafičkih interfejsa. Ovakav razvoj aplikacije često dovodi do niza prototipova sa razrađenim korisničkim interfejsom, ali bez adekvatnih obrada koje stoje iza njega.

Spiralna metodologija. Ova metodologija kombinuje analizu rizika sa nekim osobinama metodologije vodopada i metodologije rapidnog razvoja. Spirala koja ilustruje ovu metodologiju prolazi u iteracijama kroz nekoliko faza kao što su planiranje i dizajn, implementacija, analiza rizika, evaluacija tekuće verzije implementacije. U svakoj iteraciji analiziraju se relevantni faktori rizika (npr. rizici koji se odnose na cenu, tempo rada, efikasnost, bezbednost, itd). Ako neki rizik ne može biti eliminisan, naručilac mora da odluči da li se sa projektom nastavlja ili ne. Ukoliko se sa projektom nastavlja, ulazi se u sledeću iteraciju.

Ova metodologija se obično ne koristi u projektima u kojima analiza rizika može da ugrozi isplativost projekta.

Metodologija ekstremnog razvoja. Ovo je jedna od tzv. *agilnih* metodologija u kojima se koristi pojednostavljeni iterativni razvoj u čijem centru su pojedinačni programeri. U metodologiji ekstremnog programiranja, faze se sprovode u veoma malim koracima i prva iteracija može da dovede, do svesno nepotpune ali funkcionalne celine, već za jedan dan ili nedelju (za razliku od nekoliko meseci ili čak godina u metodologiji vodopada).

Osnovni principi koje agilni razvoj prati su sledeći:

- Ljudi i njihova komunikacija su bitniji od procesa i alata – samoorganizovanje, motivacija programera i njihove međusobne interakcije su jako važne (podstiče se, na primer, rad u parovima). Projekat ne može da uspe ako ljudi nisu kvalitetni. Izgradnja tima važnija je od izgradnje okruženja.
- Naručiocu je mnogo bolje tokom sastanaka prikazivati softver koji nešto radi (ne sve i ne savršeno) nego mu prikazivati samo planove i dokumente. Dokumentacija mora da postoji, ali previše dokumentacije može biti gore nego premalo (naime, osnovni cilj projekta obično nije dokumentacija, već softver).
- Zahtevi ne mogu da se u potpunosti utvrde na samom početku pa naručilac treba da konstantno bude uključen u razvojni tim.
- Zahtevi se menjaju i mnogo je bolje na njih odgovarati odmah nego se striktno držati unapred zacrtanog plana. Vizija i glavni ciljevi moraju da postoje, ali treba biti fleksibilan i dopustiti promene tokom razvoja.

Agilni razvoj vezan je često i za automatizovane testove kojima se sa jedne strane preciziraju zahtevi koje određene softverske komponente (na primer, funkcije) treba da zadovolje, a sa druge strane proverava se koliko te komponente zaista ispunjavaju date zahteve. Testovi se često pišu pre implementacije samih funkcija, u njima se vrše pozivi funkcija za određene karakteristične ulaze i proverava se da li se vraćeni rezultati poklapaju sa očekivanim izlazima. Testovi se pišu na osnovu opštih pravila za testiranje softvera o kojima će biti više reči u narednim poglavljima.

U ekstremnom razvoju, programiranje se obično vrši u paru. Dva programera istovremeno rešavaju problem, jedan piše kôd dok drugi proverava njegovo pisanje i pokušava da pronađe i ukaže na eventualne greške i nedostatke. Tokom razvoja svi unapred zadati testovi treba da budu zadovoljeni, ali tokom razvoja programeri mogu i da dodaju nove testove. Implementacija se završava tek kada oba programera budu zadovoljna kodom, kada kôd prolazi sve testove i kada programeri ne mogu da se sete novih testova koje bi trebalo dodati.

Dizajn i arhitektura koda se ne zadaju unapred, već se oni javljaju tokom samog kodiranja i kasnije, u procesu refaktorisanja koda (izmene njegove unutrašnje strukture bez izmene funkcionalnosti). Dakle, isti ljudi koji rade kodiranje brinu o dizajnu. Nekompletni, ali funkcionalni sistem se konstantno demonstrira naručiocu (i ostatku tima, ali i krajnjim korisnicima). Tada se kreće sa pisanjem testova za naredni važan deo sistema.

1.2 Planiranje

Faza planiranja može da obuhvati sledeće zadatke:

- analizu i specifikovanje problema,
- modelovanje rešenja,
- dizajn softverskog rešenja.

Analiza se pre svega bavi samim problemom tj. njegovom preciznom postavkom i formulacijom, dok se modelovanje i dizajn bave rešenjem problema koji je definisan u fazi analize. U fazi planiranja sistema obično se koriste različite dijagramske tehnike i specijalizovani alati koji podržavaju kreiranje ovakvih dijagrama (tzv. CASE alati – Computer Aided Software Engineering).

1.2.1 Analiza i specifikovanje problema

S obzirom na to da se softver obično piše za naručioce, u procesu analize i specifikovanja problema vrši se intenzivna komunikacija analitičara sa njima. Kada se softver pravi po narudžbini, naručioci mogu da budu krajnji korisnici ili njihovi predstavnici, ali čest slučaj u velikim kompanijama je da ulogu naručioca preuzimaju radnici zaposleni u odeljenju prodaje ili marketinga (koji imaju ideju kakav proizvod bi kasnije mogli da prodaju).

Tokom sastanaka često se najpre vrši analiza postojećih rešenja (na primer, postojećeg poslovnog procesa u kompaniji koja uvodi informacioni sistem) i razmatraju se mogućnosti njihovog unapređenja uvođenjem novog softvera. Kako naručioci obično nemaju potrebno informatičko obrazovanje, oni često nisu svesni svih mogućnosti koje novi softver može da im pruži. Jedan od zadataka analitičara je da ovakve mogućnosti uočava i da o njima komunicira sa naručiocima. Naručioci formulišu zahteve (engl. requirements) koje softver koji se proizvodi treba da zadovolji. Zahtevi su često neprecizni, pa čak i kontradiktorni, i zadatak analitičara je da ih u saradnji sa naručiocima precizira i uobliči.

Pored precizne analize zahteva, zadatak analize je i da se naprave procene obima, cene i vremena potrebnog da se projekat realizuje. Preciznije, analiza treba da proceni:

- obim posla koji će da se radi (potrebno je precizno definisati šta projekat treba da obuhvati, a šta ne);
- rizike koji postoje (i da definiše odgovarajuće reakcije projektnog tima u slučaju da nešto pođe drugačije nego što je planirano);
- resurse (ljudske, materijalne) koji su potrebni;
- očekivanu cenu realizacije projekta i njegovih delova;
- plan rada (po fazama) koji je neophodno poštovati.

Obim posla često se izražava u terminima broja potrebnih čovek-meseci (1 čovek-mesec podrazumeva da jedan čovek na projektu radi mesec dana).

Rezultat analize je precizna specifikacija problema. Specifikacijom je potrebno što preciznije opisati problem, prirodu ulaznih podataka i oblik u kome se žele rešenja — izlazni rezultati. Specifikacija programa bavi se pitanjem šta program treba da uradi, kojom brzinom, koja mu je maksimalna dozvoljena veličina, itd.

Kada je problem precizno specifikovan, prelazi se na sledeće faze u kojima se modeluje i dizajnira rešenje specifikovanog problema.

1.2.2 Modelovanje rešenja

Modelovanje rešenja obično sprovodi projektant, koji mora da razume specifikaciju problema i da je u stanju da izradi matematičke modele problema i da izabere adekvatna softverska rešenja, npr. programski jezik, bazu podataka, relevantne biblioteke, strukture podataka, matematičku reprezentaciju problema, algoritamska rešenja, itd).

1.2.3 Dizajn softverskog rešenja

Tokom dizajna vrši se preciziranje rešenja imajući u vidu konkretnu hardversku i softversku platformu na kojoj rešenje treba da bude realizovano. Koncepti koji su u ranijim fazama nezavisni od konkretnih tehnologija se razrađuju i dobija se opšti plan kako sistem treba da bude izgrađen na konkretnoj tehnologiji. Tokom dizajna često se koriste neki unapred ponuđeni uzorci (engl. design patterns) za koje je praksa pokazala da predstavljaju kvalitetna rešenja za određenu klasu problema.

U jednostavnijim slučajevima (na primer kada softver treba da radi autonomno, bez korisnika i korisničkog interfejsa), dizajn može biti dat i u neformalnom tekstualnom oblik ili u vidu jednostavnog dijagrama toka podataka tj. tokovnika (engl. data flow diagram). U kompleksnijim slučajevima mogu se koristiti napredniji sistemi poput UML (Unified Modeling Language). UML je standardizovana grafička notacija (kaže se i *grafički jezik*) koja omogućava modelovanje podataka, modelovanje poslovnih procesa i modelovanje softverskih komponenti. UML može biti korišćen i u drugim fazama životnog ciklusa softvera.

Osnovni pojmovi dizajna softvera su

- Apstrahovanje (engl. abstraction) – apstrahovanje je proces generalizacije kojim se odbacuju nebitne informacije tokom modelovanja nekog entiteta ili procesa i zadržavaju samo one informacije koje su bitne za sam softver. Tokom apstrakcije vrši se i identifikacija srodnih entiteta. Na primer, apstrahovanjem se uočava da boja očiju studenta nema nikakvog značaja u informacionom sistemu fakulteta i ta informacija se onda odbacuje prilikom predstavljanja studenta u sistemu. Apstrahovanje podrazumeva i apstrahovanje kontrole programa, i apstrahovanje podataka. Objekti i klase u objektno-orijentisanom programiranju kombinuju ova dva oblika apstrahovanja.

- Profinjavanje (engl. refinement) – profinjavanje je proces razvoja programa odozgo-naniže. Nerazrađeni koraci se tokom profinjavanja sve više preciziraju dok se na samom kraju ne dođe do sasvim preciznog opisa u obliku izvršnog programskog koda. U svakom koraku jedna ili više funkcija tj. naredbi se razlaže na detaljnije funkcije tj. naredbe. Na primer, poziv `pozovi_prijatelja()` može da se razloži na `idi_do_telefona()`, `podigni_slusalicu()`, `okreni_broj()`, itd. Apstrahovanje i profinjavanje međusobno su suprotni procesi.

- Modularnost (engl. modularity) – softver se deli na komponente koje se nazivaju moduli. Svaki modul ima precizno definisanu funkcionalnost i poželjno je da moduli što manje zavise jedni od drugih kako bi mogli da se koriste i u drugim programima.

- Sakrivanje informacija (engl. information hiding) – moduli treba da budu dizajnirani tako da njihove unutrašnje informacije ne mogu biti dostupne iz drugih modula (tako se smanjuje zavisnost između modula i postiže se unutrašnja koherentnost svakog modula). Korisnicima modula važno je samo koju funkcionalnost im taj modul pruža, ali ne i način na koji je ta funkcionalnost realizovana.

- Arhitektura softvera (engl. software architecture) – arhitektura se odnosi na celokupnu strukturu softvera i načine na koje ta struktura obezbeđuje integritet i uspeh sistema. Dobra arhitektura treba da obezbedi željeni ishod projekta (dobre performanse, kvalitetan softver, poštovanje rokova i uklapanje u planirane troškove).

- Podela strukture (engl. structural partitioning) – struktura programa treba da bude podeljena i vertikalno i horizontalno. Horizontalna podela definiše hijerarhiju modula za svaku krupniju funkcionalnost softvera. Vertikalna podela sugeriše da kontrola i podela posla treba da bude distribuirana odozgo naniže u strukturi programa.

Jedna od najznačajnijih veština potrebnih za razvoj dobrih programa je strukturna dekompozicija.[1]

Dizajn u vidu tokovnika

Dizajn u vidu tokovnika predstavlja niz transformacija (zadatih u obliku funkcija) kojima se dati ulaz prevodi u željeni izlaz. Ovakav dizajn predstavlja se u obliku dijagrama toka podataka (tzv. tokovnika). Ovi dijagrami ilustruju kako podaci teku kroz sistem i kako se izlaz izvodi iz ulaza kroz niz funkcionalnih transformacija. Tokovnici su obično koristan i intuitivan način za opis sistema. Obično ih je moguće

[1] Delom zasnovano na poglavlju „Function-oriented design“ knjige Ian Sommerville: „Software Engineering“.

razumeti i bez prethodne obuke. Mnogi alati koji se koriste tokom dizajna softvera (npr. CASE alati) omogućuju kreiranje tokovnika. Notacija koja se koristi u tokovnicima nije standardizovana, ali različite notacije su često slične. Na primer,

- Zaobljeni pravougaonici predstavljaju funkcije koje transformišu ulaz u izlaz. Ime u pravougaoniku označava ime funkcije.
- Pravougaonici označavaju skladišta podataka. I njima je poželjno dati ilustrativna imena.
- Krugovi predstavljaju korisnike koji interaguju sa sistemom dajući ulaz ili prihvatajući izlaz.
- Strelice označavaju smer toka podataka. Na strelicama se označava vrsta podataka koja se prenosi.

Tokovnici opisuju funkcijske transformacije, ali ne sugerišu kako se one mogu implementirati. Na primer, sistem opisan na ovaj način može biti implementiran u obliku jednog programa sa više funkcija ili u obliku niza programa koji međusobno komuniciraju.

Funkcijski-orijentisan dizajn

U funkcijski-orijentisanom dizajnu svaki softverski modul je odgovoran samo za jedan zadatak i sprovodi ga sa minimalnim uticajem na druge delove. Ovo omogućava jednostavnije održavanje sistema.

U funkcijski-orijentisanom dizajnu sistem se dekomponuje na skup funkcija koje intereaguju sa zajedničkim centralnim stanjem sistema. Funkcije takođe mogu da imaju i svoje lokalno stanje, ali ono se čuva samo tokom izvršavanja funkcije. Ovakav dizajn se neformalno praktikovao još od početaka programiranja. Programi su dekomponovani u potprograme koji su funkcijski po svojoj prirodi.

Funkcijski orijentisan dizajn sakriva detalje algoritama u same funkcije, ali stanje sistema nije sakriveno. Ovo može da dovede do problema jer funkcije (svojim sporednim efektima) mogu da promene stanje na način na koji to druge funkcije ne očekuju. Izmena jedne funkcije i načina na koji ona menja stanje sistema može da proizvede efekat na ponašanje ostalih funkcija. Dakle, funkcijski-orijentisan dizajn može da bude uspešan u slučajevima kada je stanje sistema minimalno i kada je razmena informacija između funkcija eksplicitna. Sistemi u kojima odgovor zavisi od pojedinačnog unosa (u kojima istorija ranijih uslova nije relevantna da bi se dobio odgovor na tekući unos) su prirodno funkcijski-orijentisani. Na primer, softver koji kontroliše bankomate je takav – usluga koja se pruža korisniku ne zavisi od prethodnih transakcija. Sistem se implementira kao neprekidna petlja u kojoj se akcije pokreću kada se ubaci kartica. Sistem je implementiran kroz funkcije poput `izbaci_novac`, `proveri_broj_racuna`, `izdaj_potvrdu` i slično. Stanje programa je minimalno.

Strukturna dekompozicija

Ponekad strukturna i funkcionalna podela nisu isto iako tradicionalno računarstvo često polazi od pretpostavke da jesu. Strukturna dekompozicija se odnosi na podelu sistema na „delove" tj. na komponente. Funkcionalna dekompozicija se odnosi na podelu funkcija sistema. Često postoji tesna veza između delova i funkcionalnosti koju svaki od delova pruža, ali ponekad strukturna i funkcionalna podela ipak nisu isto.

Pored dijagrama toka podataka često je korisno napraviti i strukturni model sistema koji prikazuje kako je funkcija implementirana time što se prikazuje koje druge funkcije ona poziva. Strukturni dijagrami predstavljaju grafički način da se predstavi ovakva hijerarhija. Funkcije se često predstavljaju pravougaonicima, dok se hijerarhija prikazuje povezivanjem pravougaonika linijama. Ulaz i izlaz (koji se implementiraju kao parametri funkcija ili kao deljene globalne promenljive) prikazuju se označenim strelicama. Strelica koja ulazi u pravougaonik predstavlja ulaz, a ona koja izlazi iz njega predstavlja izlaz. Skladišta podataka se predstavljaju kao zaokruženi pravougaonici, a korisnički ulazi kao krugovi. Kreiranje strukturnog dijagrama na osnovu tokovnika zahteva elaboraciju, inventivnost i kreativnost dizajnera. Ipak postoje neka opšta uputstva, neka od njih su nabrojana u nastavku.

- Mnogi sistemi, naročito poslovni, za koje funkcionalni dizajn ima smisla mogu da se shvate kao trofazni sistemi. U prvoj fazi se prihvata ulaz i vrši se njegova provera, zatim se u drugoj fazi vrši obrada ulaza, a zatim se u trećoj fazi generiše izlaz (često u formi nekog izveštaja koji se smešta u neku datoteku). Usput se obično modifikuje i neko globalno skladište podataka. U prvoj iteraciji

strukturnog dijagrama obično se prepoznaju 3-4 funkcije koje odgovaraju ulazu, obradi, promeni globalnog skladišta podataka i izlazu.

- Ako se zahteva provera podataka, funkcije koje vrše proveru su podređene funkciji koja prihvata ulaz. Funkcije za formatiranje izlaza, štampu i pisanje na disk podređene su funkciji koja generiše izlaz.
- Funkcije pri vrhu hijerarhije imaju zadatak da kontrolišu i koordinišu skup funkcija nižeg nivoa.
- Cilj procesa dizajna je da se dođe do veoma slabo spregnutih funkcija, od kojih svaka ima visok stepen unutrašnje kohezije. Funkcije zato treba da rade jednu i samo jednu stvar.
- Svakom čvoru u strukturnom dijagramu odgovara između 2 i 7 podređenih čvorova. Ako postoji samo jedan, to govori da funkcija predstavljena tim čvorom ima slab stepen kohezije. Podređena komponenta možda ne treba da bude zasebna funkcija. Ako postoji previše podređenih funkcija to može da znači da je dizajn previše razrađen za ovaj nivo apstrakcije.

Pitanja i zadaci za vežbu

Pitanje 1.1. *Navesti faze razvoja softvera i ko ih obično sprovodi?*

Pitanje 1.2. *Opisati metodologije razvoja softvera.*

Pitanje 1.3. *Koje dve vrste dokumentacije treba da postoje?*

Glava 2

Principi pisanja programa

Programi napisani na višem programskom jeziku sredstvo su komunikacije između čoveka i računara ali i između ljudi samih. Razumljivost, čitljivost programa, iako nebitna za računar, od ogromne je važnosti za kvalitet i upotrebljivost programa. Naime, u održavanje programa se obično uloži daleko više vremena i truda nego u njegovo pisanje a održavanje sistema često ne rade oni programeri koji su napisali program. Pored toga, razumljivost programa omogućava lakšu analizu njegove ispravnosti i složenosti. Preporuke za pisanje obično nisu kruta pravila i ona najpre predstavljaju zapravo ideje kojima se treba rukovoditi u pisanju programa, u aspektima formatiranja, nazubljivanja, imenovanja promenljivih i funkcija itd.

U daljem tekstu će, kao na jedan primer konvencija za pisanje programa, biti ukazivano na preporuke iz teksta *Linux Kernel Coding Style*, Linusa Torvaldsa, autora operativnog sistema Linux koji je napisan na jeziku C. Nekoliko saveta i preporuka u nastavku preuzeto je iz znamenite knjige *The Practice of Programming* autora Brajana Kernigena i Roba Pajka. Preporuke navedene u nastavku se često odnose na sve programske jezike, ali ponekad samo na jezik C.

2.1 Timski rad i konvencije

Za svaki obimniji projekat potrebno je usaglasiti konvencije za pisanje programa. Da bi ih se lakše pridržavalo, potrebno je detaljno motivisati i obrazložiti pravila. Ima različitih konvencija i one često izazivaju duge i zapaljive rasprave između programera. Mnogi će, međutim, reći da nije najvažnije koja konvencija se koristi nego koliko strogo se nje pridržava. Strogo i konzistentno pridržavanje konvencije u okviru jednog projekta izuzetno je važno za njegovu uspešnost. Jedan isti programer treba da bude spreman da u različitim timovima i različitim projektima koristi različite konvencije.

Kako bi se olakšalo baratanje programom koji ima na stotine datoteka koje menja veliki broj programera, u timskom radu obično se koriste *sistemi za kontrolu verzija* (kao što su CVS, SVN, Git, Bazaar, Mercurial, Visual SourceSafe). I ovi sistemi nameću dodatna pravila i omogućavaju dodatne konvencije koje tim treba da poštuje.

2.2 Vizualni elementi programa

Prva ideja o programu formira se na osnovu njegovog izgleda – njegovih vizualnih elemenata, kao što su broj linija u datoteci, broj karaktera u liniji, nazubljivanje, grupisanje linija i slično. Vizualni elementi programa i njegovo formatiranje često su od ključne važnosti za njegovu čitljivost. Formatiranje, konkretno nazubljivanje, u nekim jezicima (npr. python) čak utiče na značenje programa.

Formatiranje i vizualni elementi programa treba da olakšaju razumevanje koda koji se čita, ali i pronalaženje potrebnog dela koda ili datoteke sa delom programa. Formatiranje i vizualni elementi programa treba da olakšaju i proces pisanja programa. U tome, pomoć autoru programa pružaju alati u okviru kojih se piše program – specijalizovani editori teksta ili integrisana razvojna okruženja (engl. IDE, Integrated Development Environment) koja povezuju editor, kompilator, debager i druge alate potrebne u razvoju softvera.

2.2.1 Broj karaktera u redu

U modernim programskim jezicima dužina reda programa nije ograničena. Ipak, predugi redovi mogu da stvaraju probleme. Na primer, predugi redovi mogu da zahtevaju horizontalno „skrolovanje“ kako bi se video njihov kraj, što može da drastično oteža čitanje i razumevanje programa. Takođe, ukoliko se program štampa, dugi redovi mogu da budu presečeni i da naruše formatiranje. Zbog ovih i ovakvih problema, preporučuje se pridržavanje nekog ograničenja – obično 80 karaktera u redu. Konkretna preporuka za 80 karaktera u redu je istorijska i potiče od ograničenja na starim ekranima i štampačima i drugim uređajima. Ipak, ona je i danas široko prihvaćena kao pogodna. Ukoliko red programa ima više od 80 karaktera, to najčešće ukazuje na to da kôd treba reorganizovati uvođenjem novih funkcija ili promenljivih. Broj 80 (ili bilo koji drugi) kao ograničenje za broj karaktera u redu ne treba shvatati kruto već kao načelnu preporuku koja može biti narušena ako se tako postiže bolja čitljivost.

2.2.2 Broj naredbi u redu, zagrade i razmaci

Red programa može da bude prazan ili da sadrži jednu ili više naredbi. Prazni redovi mogu da izdvajaju blokove blisko povezanih naredbi (na primer, blok naredbi za koje se može navesti komentar o tome šta je njihova svrha). Ako se prazni redovi koriste neoprezno, mogu da naruše umesto da poprave čitljivost. Naime, ukoliko ima previše praznih linija, smanjen je deo kôda koji se može videti i sagledavati istovremeno na ekranu. Po jednoj konvenciji, zagrade koje označavaju početak i kraj bloka navode se u zasebnim redovima (u istoj koloni), a po drugoj, otvorena zagrada se navodi u nastavku prethodne naredbe, a zatvorena u zasebnom redu ili u redu zajedno sa ključnom rečju `while` ili `else`. Torvalds preporučuje ovu drugu konvenciju, uz izuzetak da se otvorena vitičasta zagrada na početku definicije funkcije piše u zasebnom redu.

Naredni primer prikazuje deo koda napisan sa većim brojem praznih redova i prvom konvencijom za zagrade:

```
do
{

  printf("Unesi ceo broj: ");
  scanf("%d", &i);

  if (duzina == alocirano)
  {

    alocirano += KORAK;
    a = realloc(a, alocirano*sizeof(int));
    if (a == NULL)
      return 1;
  }

  a[duzina++] = i;
}
while (i != -1);
```

Isti deo kôda može biti napisan sa manjim brojem praznih redova i drugom konvencijom za zagrade. Ovaj primer prikazuje kompaktnije zapisan kôd koji je verovatno čitljiviji većini iskusnih C programera:

```
do {
  printf("Unesi ceo broj: ");
  scanf("%d", &i);
  if (duzina == alocirano) {
    alocirano += KORAK;
    a = realloc(a, alocirano*sizeof(int));
    if (a == NULL)
      return 1;
  }
```

```
  a[duzina++] = i;
} while (i != -1);
```

Jedan red može da sadrži i više od jedne naredbe. To je prihvatljivo samo (a tada može da bude i preporučljivo) ako se radi o jednostavnim inicijalizacijama ili jednostavnim dodelama vrednosti članovima strukture, na primer:

```
...
int i=10; double suma=0;
tacka.x=0; tacka.y=0;
```

Ukoliko je u petlji ili u `if` bloku samo jedna naredba, onda nisu neophodne zagrade koje označavaju početak i kraj bloka i mnogi programeri ih ne pišu. Međutim, iako nisu neophodne one mogu olakšati razumevanje kôda u kojem postoji višestruka `if` naredba. Dodatno, ukoliko se u blok sa jednom naredbom i bez vitičastih zagrada u nekom trenutku doda druga naredba lako može da se previdi da postaje neophodno navesti i zagrade.

Veličina blokova kôda je takođe važna za preglednost, pa je jedna od preporuka da vertikalno rastojanje između otvorene vitičaste zagrade i zatvorene vitičaste zagrade koja joj odgovara ne bude veće od jednog ekrana.

Obično se preporučuje navođenje razmaka oko ključnih reči i oko binarnih operatora, izuzev `.` i `->`. Ne preporučuje se korišćenje razmaka kod poziva funkcija, unarnih operatora, izuzev operatora `sizeof` i operatora kastovanja. Ne preporučuje se navođenje nepotrebnih zagrada, posebno u okviru povratne vrednosti. Na primer:

```
if (uslov)  {
   *a = -b + c + sizeof (int) + f(x);
   return -1;
}
```

2.2.3 Nazubljivanje teksta programa

Nazubljivanje teksta programa nebitno je kompilatoru, ali je skoro neophodno programeru. Nazubljivanje naglašava strukturu programa i olakšava njegovo razumevanje. Red programa može biti uvučen u odnosu na početnu kolonu za nekoliko blanko karaktera ili nekoliko tab karaktera. Tab karakter može da se u okviru editora interpretira na različite načine (tj. kao različit broj belina), te je preporučljivo u programu sve tab karaktere zameniti razmacima (za šta u većini editora postoji mogućnost) i čuvati ga u tom obliku. Na taj način, svako će videti program (na ekranu ili odštampan) na isti način.

Ne postoji kruto pravilo za broj karaktera za jedan nivo uvlačenja. Neki programeri koriste 4, a neki 2 – sa motivacijom da u redovima od 80 karaktera može da stane i kôd sa dubokim nivoima. Torvalds, sa druge strane, preporučuje broj 8, jer omogućava bolju preglednost. Za programe koji imaju više od tri nivoa nazubljivanja, on kaže da su ionako sporni i zahtevaju prepravku.

2.3 Imenovanje promenljivih i funkcija

Imenovanje promenljivih i funkcija veoma je važno za razumljivost programa i sve je važnije što je program duži. Pravila imenovanja mogu da olakšaju i izbor novih imena tokom pisanja programa. Imena promenljivih i funkcija (pa i datoteka programa) treba da sugerišu njihovu ulogu i tako olakšaju razumevanje programa.

Globalne promenljive, strukture i funkcije treba da imaju opisna imena, potencijalno sačinjena od više reči. U *kamiljoj* notaciji (popularnoj među Java i C++ programerima), imena od više reči zapisuju se tako što svaka nova reč (sem eventualno prve) počinje velikim slovom, na primer, `brojKlijenata`. U notaciji sa podvlakama (popularnoj među C programerima), sve reči imena se pišu malim slovima a reči su razdvojene podvlakama, na primer, `broj_klijenata`. Imena makroa i konstanti pišu se obično svim velikim slovima, a imena globalnih promenljivih počinju velikim slovom.

Lokalne promenljive, a posebno promenljive koje se koriste kao brojači u petljama treba da imaju kratka i jednostavna, a često najbolje, jednoslovna imena – jer se razumljivost lakše postiže sažetošću. Imena za brojače u petljama su često `i`, `j`, `k`, za pokazivače `p` i `q`, a za niske `s` i `t`. Preporuka je i da

se lokalne promenljive deklarišu što kasnije u okviru funkcije i u okviru bloka u kojem se koriste (a ne u okviru nekog šireg bloka).

Jedan, delimično šaljiv, savet za imenovanje (i globalnih i lokalnih) promenljivih kaže da broj karaktera u imenu promenljive treba da zavisi od broja linija njenog dosega i to tako da bude proporcionalan logaritmu broja linija njenog dosega.

Za promenljive i funkcije nije dobro koristiti generička imena kao `rezultat`, `izracunaj(...)`, `uradi(...)` i slično, već sugestivnija, kao što su, na primer, `kamata`, `izracunaj_kamatu(...)`, `odstampaj_izvestaj_o_kamati(...)`.

Imena funkcija dobro je da budu bazirana na glagolima, na primer, `izracunaj_kamatu(...)` umesto `kamata(...)` i `get_time(...)` umesto `time(...)`. Za funkcije koje vraćaju istinitosnu vrednost, ime treba da jasno sugeriše u kom slučaju se vraća vrednost *tačno*, na primer, bolje je ime `is_prime(...)` nego `check_prime(...)`.

Mnoge promenljive označavaju neki broj entiteta (na primer, broj klijenata, broj studenata, broj artikala) i za njih se može usvojiti konvencija po kojoj imena imaju isti prefiks ili sufiks (na primer, `br_studenata` ili `num_students`. U ovom smislu, znatno dalje ide *mađarska notacija*.[1] Postoje zapravo dve varijante mađarske notacije: sistemska i aplikacijska. U sistemskoj mađarskoj notaciji ime promenljive počinje uvek slovom ili slovima koja odgovaraju njenom tipu, na primer, `uBrojKlijenata` može da bude promenljiva tipa `unsigned int`. Ideja je da se na ovaj način tip promenljive vidi na svakom mestu gde se ona koristi (pa ne mora da se proverava njena deklaracija). U aplikacijskoj mađarskoj notaciji, ime promenljive se kreira na osnovu njene ulogu u programu, a ne na osnovu tipa. Na primer, imena promenljivih mogu da počinju sa `rw` i `col` i označavaju poziciju kurzora u dokumentu. Jedan par takvih promenljivih mogu da budu, na primer, `rwOutputWindow` i `colOutputWindow`. Ove dve promenljive su istog tipa, ali nema smisla vrednost jedne pridružiti drugoj i to sugerišu izabrani prefiksi (te se u kodu lako mogu primetiti neka pogrešna izračunavanja). Mađarska notacija (posebno „sistemska“) ima mnogo kritičara (uključujući Torvaldsa) koji smatraju da je informacija o tipovima ionako poznata kompilatora a da dodatni prefiksi više zbunjuju programera nego što mu pomažu.

I programeri kojima to nije maternji jezik, iako to nije zahtev projekta, često imenuju promenljive i funkcije na osnovu reči engleskog jezika. To je posledica istorijskih razloga i dominacije engleskog jezika u programerskoj praksi, kao i samih ključnih reči skoro svih programskih jezika (koje su na engleskom). Prihvatljivo je (ako nije zahtev projekta drugačiji) imenovanje i na maternjem jeziku i na engleskom jeziku — jedino je neprihvatljivo mešanje ta dva. Imenovanje na bazi engleskog i komentari na engleskom mogu biti pogodni ukoliko postoji i najmanja mogućnost da se izvorni program koristi u drugim zemljama, ili od strane drugih timova, ili da se učini javno dostupnim i slično. Naime, u programiranju (kao i u mnogim drugim oblastima) engleski jezik je opšteprihvaćen u svim delovima sveta i tako se može osigurati da program lakše razumeju svi.

Kvalitet imenovanja promenljivih i funkcija može se „testirati“ na sledeći zanimljiv način: ako se kôd može pročitati preko telefona tako da ga sagovornik na drugoj strani razume, onda je imenovanje dobro.

2.4 Pisanje izraza

Za dobrog programera neophodno je da poznaje sva pravila programskog jezika jer će verovatno češće i više raditi na tuđem nego na svom kôdu. S druge strane, programer u svojim programima ne mora i ne treba da koristi sva sredstva izražavanja tog programskog jezika, već može i treba da ih koristi samo delom, oprezno i uvek sa ciljem pisanja razumljivih programa. Ponekad programer ulaže veliku energiju u pisanje najkonciznijeg mogućeg kôda što ponekad može da bude protraćen trud, jer glavni cilj treba da bude pisanje jasnih programa. Sve ovo se odnosi na mnoge aspekte pisanja programa, uključujući pisanje izraza.

Preporučuje se pisanje izraza u jednostavnom i intuitivno jasnom obliku. Na primer, umesto:

```
(!(c < '0') && !(c > '9'))
```

bolje je:

```
(c >= '0' && c <= '9')
```

[1] Aplikacijsku mađarsku notaciju razvio je Čarls Simonji, poreklom iz Mađarske, koji je tokom osamdesetih godina bio jedan od vodećih programera u kompaniji Microsoft. Konvencija se koristila i još se koristi u mnogim proizvodima ove kompanije, ali su je prihvatili i mnogi drugi programeri.

Zagrade, čak i ako nisu neophodne, mogu da olakšaju čitljivost. Prethodni primer može da se zapiše na sledeći način:

```
 ((c >= '0') && (c <= '9'))
```

a naredba:

```
prestupna = g % 4 == 0 && g % 100 != 0 || g % 400 == 0;
```

može čitljivije da se napiše na sledeći način:

```
prestupna = ((g%4 == 0) && (g % 100 != 0)) || (g % 400 == 0);
```

Razmak oko operatora ne mora da se piše i njegovo uklanjanje može da naglasti prioritet i popravi čitljivost. U navedenom primeru zato piše `g%4 == 0` umesto `g % 4 == 0`.

Suviše komplikovane izraze treba zameniti jednostavnijim i razumljivijim. Kernigen i Pajk navode primer:

```
 *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

i bolju, jednostavniju varijantu:

```
if (2*k < n-m)
  *xp = c[k+1];
else
  *xp = d[k--];
*x += *xp;
```

Kernigen i Pajk navode i primer u kojem je moguće i poželjno pojednostaviti komplikovana izračunavanja. Umesto:

```
subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

bolji je (ekvivalentan) kôd:

```
subkey = subkey >> (bitoff & 0x7);
```

Zbog komplikovanih, a u nekim situacijama i nedefinisanih, pravila poretka izračunavanja i dejstva sporednih efekata (kao, na primer, kod operatora inkrementiranja i dekrementiranja), dobro je pojednostaviti kôd kako bi njegovo izvršavanje bilo jednoznačno i jasno. Na primer, umesto:

```
str[i++] = str[i++] = ' ';
```

bolje je:

```
str[i++] = ' ';
str[i++] = ' ';
```

Poučan je i sledeći čuveni primer: nakon dodele `a[a[1]]=2;`, element `a[a[1]]` nema nužno vrednost 2 (ako je na početku vrednost `a[1]` bila jednaka 1, a vrednost `a[2]` različita od 2). Navedeni primer pokazuje da treba biti veoma oprezan sa korišćenjem indeksa niza koji su i sami elementi niza ili neki komplikovani izrazi.

2.5 Korišćenje idioma

Idiomi su ustaljene jezičke konstrukcije koje predstavljaju celinu. Idiomi postoje u svim jezicima, pa i u programskim. Tipičan idiom u jeziku C je sledeći oblik `for`-petlje:

```
for (i = 0 ; i < n; i++)
  ...
```

Kernigen i Pajk zagovaraju korišćenje idioma gde god je to moguće. Na primer, umesto varijanti

```
i=0;
while (i <= n-1)
  a[i++] = 1.0;

for (i = 0; i<n; )
  a[i++] = 1.0;

for (i = n; --i >= 0; )
  a[i] = 1.0;
```

smatraju da je bolja varijanta:

```
for (i = 0 ; i < n; i++)
  a[i] = 1.0;
```

jer je najčešća i najprepoznatljivija. Štaviše, Kernigen i Pajk predlažu, delom i ekstremno, da se, bez dobrog razloga i ne koristi nijedna forma `for`-petlji osim navedene, zatim sledeće, za prolazak kroz listu (videti poglavlje 7.1):

```
for (p = list ; p != NULL ; p = p->next)
  ...
```

i sledeće, za beskonačnu petlju:

```
for (;;)
  ...
```

Glavni argument za korišćenje idioma je da se kôd brzo razume a i da svaki drugi („neidiomski“) konstrukt privlači dodatnu pažnju što je dobro, jer se bagovi češće kriju u njima. Kao dodatne primere idioma, Kernigen i Pajk navode:

```
while ((c = getchar()) != EOF)
  ...
```

i

```
p = malloc(strlen(buf)+1);
strcpy(p, buf);
```

2.6 Korišćenje makroa

Makroe sa argumentima obrađuje pretprocesor i u fazi izvršavanja, za razliku od funkcija, nema kreiranja stek okvira, prenosa argumenata i slično. Zbog uštede memorije i računarskog vremena, makroi sa argumentima su nekada smatrani poželjnom alternativom funkcijama. Danas, u svetu mnogo bržih računara nego nekada, smatra se da loše strane makroa sa argumentima prevazilaze dobre strane i da makroe treba izbegavati. U loše strane makroa sa argumentima spada to što ih obrađuje pretprocesor (a ne kompilator), nema provera tipova argumenata, debager ne može da prati definiciju makroa i slično. Postoje i dodatne loše strane ilustrovane primerima u nastavku (od kojih su neki diskutovani i u prvom delu ove knjige, u poglavlju **??**).

Ukoliko se u tekstu zamene neki argument pojavljuje više puta, postoji mogućnost da on bude izračunat više puta. Na primer, ukoliko postoji makro definicija:

```
#define  max(A, B)  ((A) > (B) ? (A) : (B))
```

ukoliko je negde u programu navedeno `max(a++, b++)`, ovaj tekst biće (u fazi pretprocesiranja) zamenjen tekstom `((a++) > (b++) ? (a++) : (b++))`, što će dovesti do toga da se veća od vrednosti `a` i `b` inkrementira dva puta (a što verovatno nije bilo planirano). Slično je i sa makroom:

```
#define  isUpperCase(c)  ((c) >= 'A' && (c) <= 'Z')
```

Naime, ako u programu postoji naredba `while(isUpperCase(c = getchar())) ...`, pretprocesor će je zameniti sa `while((c = getchar()) >= 'A' && (c = getchar()) <= 'Z') ...`, pa će u svakom prolasku kroz petlju biti sa standardnog ulaza učitano dva karaktera (i biće proveravani na nepotpun način).[2]

Važno je voditi računa i o zagradama u tekstu zamene, kako bi bio očuvan poredak primene operacija. Na primer, ukoliko se definicija

```
#define kvadrat(x)  x*x
```

primeni na `kvadrat(a+2)`, tekst zamene će biti `a+2*a+2`, a ne `(a+2)*(a+2)`, kao što je verovatno željeno i, u tom slučaju bi trebalo koristiti:

```
#define kvadrat(x)  (x)*(x)
```

2.7 Korišćenje konstanti

Konstantne vrednosti, veličina nizova, pozicije karaktera u niskama, faktori za konverzije i druge slične vrednosti koje se pojavljuju u programima često se zovu *magični brojevi* (jer obično nije jasno odakle dolaze, na osnovu čega su dobijeni). Kernigen i Pajk kažu da je, osim 0 i 1, svaki broj u programu kandidat da se može smatrati magičnim, te da treba da ima ime koje mu je pridruženo. Na taj način, ukoliko je potrebno promeniti vrednost magične konstante (na primer, maksimalna dužina imena ulice) – to je dovoljno uraditi na jednom mestu u kôdu.

Magičnom broju može biti pridruženo simboličko ime pretprocesorskom direktivom `#define`. U tom slučaju, pretprocesor zamenjuje sva pojavljivanja tog imena konkretnom vrednošću pre procesa kompilacije, te kompilator (pa i debager) nema nikakvu informaciju o simboličkom imenu koje je pridruženo magičnoj konstantni niti o njenom tipu. Zbog toga se preporučuje da se magične konstante uvode kao konstantne promenljive ili korišćenjem nabrojivih tipova (Deo I, poglavlje **??**). Na primer, u narednoj deklaraciji

```
char imeUlice[50];
```

pojavljuje se magična konstanta 50, te se u nastavku programa broj 50 verovatno pojavljuje u svakoj obradi imena ulica. Promena tog ograničenja zahtevala bi mnoge izmene koje ne bi mogle da se sprovedu automatski (jer se broj 50 možda pojavljuje i u nekom drugom konktekstu). Zato je bolja, na primer, sledeća varijanta:

```
#define MAKS_IME_ULICE 50
char streetName[MAKS_IME_ULICE];
```

ili, još bolje:

```
const unsigned int MAKS_IME_ULICE = 50;
char streetName[MAKS_IME_ULICE];
```

U većim programima, konstante od značaja za čitav program (ili veliki njegov deo) obično se čuvaju u zasebnoj datoteci zaglavlja (koju koriste sve druge datoteke kojima su ove konstante potrebne).

Konstante se u programima mogu koristiti i za kôdove karaktera. To je loše ne samo zbog narušene čitljivosti, već i zbog narušene prenosivosti – naime, nije na svim računarima podrazumevana ASCII karakterska tabela. Dakle, umesto, na primer:

[2] U standardnoj biblioteci postoji funkcija `isupper` koja vraća ne-nula vrednost ukoliko argument predstavlja veliko slovo. U principu, uvek je bolje koristiti funkcije iz sistemske biblioteke nego implementirati svoju verziju.

```
if (c >= 65 && c <= 90)
  ...
```

bolje je pisati

```
if (c >= 'A' && c <= 'Z')
  ...
```

a još bolje koristiti funkcije iz standardne biblioteke, kad god je to moguće:

```
if (isupper(c))
  ...
```

Slično, zarad bolje čitljivosti treba pisati `NULL` (za nultu vrednost pokazivača) i `'\0'` (za završnu nulu u niskama) umesto konstante `0`.

U programima ne treba koristiti kao konstante ni veličine tipova – zbog čitljivosti a i zbog toga što se mogu razlikovati na različitim računarima. Zato, na primer, za dužinu tipa `int` nikada ne treba pisati 2 ili 4, već `sizeof(int)`. Za promenljive i elemente niza, bolje je pisati `sizeof(a)` i `sizeof(b[0])` umesto `sizeof(int)` (ako su promenljiva `a` i niz `b` tipa `int`), zbog mogućnosti da se promenljivoj ili nizu u nekoj verziji programa promeni tip.

2.8 Pisanje komentara

Čak i ako se autor pridržavao mnogih preporuka za pisanje jasnog i kvalitetnog kôda, ukoliko kôd nije dobro komentarisan njegovo razumevanje može i samom autoru predstavljati teškoću već nekoliko nedelja nakon pisanja. Komentari treba da olakšaju razumevanje kôda i predstavljaju njegov svojevrsni dodatak.

Postoje alati koji olakšavaju kreiranje dokumentacije na osnovu komentara u samom kodu i delom je generišu automatski (npr. Doxygen).

Komentari ne treba da objašnjavaju ono što je očigledno: Komentari ne treba da govore *kako* kôd radi, već *šta* radi (i zašto). Na primer, naredna dva komentara su potpuno suvišna:

```
k += 1.0;  /* k se uvecava za 1.0 */


return OK; /* vrati OK */
```

U prvom slučaju, komentar ima smisla ako objašnjava zašto se nešto radi, na primer:

```
k += 1.0;  /* u ovom slucaju kamatna stopa veca je za 1.0 */
```

U slučajevima da je neki deo programa veoma komplikovan, potrebno je u komentaru objasniti zašto je komplikovan, kako radi i zašto je izabrano takvo rešenje.

Komentari treba da budu koncizni. Kako ne bi trošili preterano vreme, komentari treba da budu što je moguće kraći i jasniji, da ne ponavljaju informacije koje su već navedene drugde u komentarima ili su očigledne iz kôda. Previše komentara ili predugi komentari predstavljaju opasnost za čitljivost programa.

Komentari treba da budu usklađeni sa kôdom. Ako se promeni kôd programa, a ne i prateći komentari, to može da uzrokuje mnoge probleme i nepotrebne izmene u programu u budućnosti. Ukoliko se neki deo programa promeni, uvek je potrebno proveriti da li je novo ponašanje u skladu sa komentarima (za taj ali i druge delove programa). Usklađenost kôda i komentara je lakše postići ako se pridržava saveta da komentari ne treba da govore ono što je očigledno iz kôda.

Komentarima treba objasniti ulogu datoteka i globalnih objekata. Komentarima treba, na jednom mestu, tamo gde su definisani, objasniti ulogu datoteka, globalnih objekata kao što su funkcije, globalne promenljive i strukture. Funkcije treba komentarisati pre same definicije a Torvalds savetuje da se izbegavaju komentari unutar tela funkcije. Čitava funkcija može da zaslužuje komentar (pre prvog reda), ali ako pojedini njeni delovi zahtevaju komentarisanje, onda je moguće da funkciju treba reorganizovati i/ili podeliti na nekoliko funkcija. Ni ovo pravilo nije kruto i u specifičnim situacijama prihvatljivo je komentarisanje delikatnih delova funkcije („posebno pametnih ili ružnih“).

Loš kôd ne treba komentarisati, već ga popraviti. Često kvalitetno komentarisanje *kako* i *zašto* neki loš kôd radi zahteva više truda nego pisanje tog dela koda iznova tako da je očigledno kako i zašto on radi.

Komentari treba da budu laki za održavanje: Treba izbegavati stil pisanja komentara u kojem i mala izmena komentara zahteva dodatni posao u formatiranju. Na primer, promena narednog opisa funkcije zahteva izmene u tri reda komentara:

```
/**************************************
* Funkcija area racuna povrsinu trougla *
**************************************/
```

Komentari mogu da uključuju standardne fraze. Vremenom se nametnulo nekoliko na bazi fraza koje se često pojavljuju u okviru komentara. Njih je lako pronaći u kôdu, a mnoga razvojna okruženja prepoznaju ih i prikazuju u istaknutoj boji kako bi privukli pažnju programera kao svojevrsna lista stvari koje treba obaviti. Najčešći markeri su:

TODO marker: označava zadatke koje tek treba obaviti, koji kôd treba napisati.

FIXME marker: označava deo kôda koji radi ali treba ga popraviti, u smislu opštijeg rešenja, lakšeg održavanja, ili bolje efikasnosti.

HACK marker: označava deo kôda u kojem je, kako bi radio, primenjen neki trik i loše rešenje koje se ne prihvata kao trajno, te je potrebno popraviti ga.

BUG marker: označava deo kôda koji je gotov i očekuje se da radi, ali je pronađen bag.

XXX marker: obično označava komentar programera za sebe lično i treba biti obrisan pre nego što kôd bude isporučen drugima. Ovim markerom se obično označava neko problematično ili nejasno mesto u kôdu ili pitanje programera.

Uz navedene markere i prateći tekst, često se navodi i ime onoga ko je uneo komentar, kao i datum unošenja komentara.

2.9 Modularnost

Veliki program je teško ili nemoguće razmatrati ako nije podeljen na celine. Podela programa na celine (na primer, datoteke i funkcije) neophodna je za razumevanje programa i nametnula se veoma rano u istoriji programiranja. Svi savremeni programski jezici su dizajnirani tako da je podela na manje celine ne samo moguća već tipičan način podele određuje sam stil programiranja (na primer, u objektno orijentisanim jezicima neki podaci i metode za njihovu obradu se grupišu u takozvane klase). Podela programa na module treba da omogući:

Razumljivost: podela programa na celine popravlja njegovu čitljivost i omogućava onome ko piše i onome ko čita program da se usredsredi na ključna pitanja jednog modula, zanemarujući u tom trenutku i iz te perspektive sporedne funkcionalnosti podržane drugim modulima.

Upotrebljivost: ukoliko je kôd kvalitetno podeljen na celine, pojedine celine biće moguće upotrebiti u nekom drugom kontekstu. Na primer, proveravanje da li neki trinaestocifreni kôd predstavlja mogući JMBG (jedinstveni matični broj građana) može se izdvojiti u zasebnu funkciju koja je onda upotrebljiva u različitim programima.

Obično se program ne deli u funkcije i onda u datoteke tek onda kada je kompletno završen. Naprotiv, podela programa u dodatke i funkcije vrši se u fazi pisanja programa i predstavlja jedan od njegovih najvažnijih aspekata.

2.9.1 Modularnost i podela na funkcije

Za većinu jezika osnovna je funkcionalna dekompozicija ili podela na funkcije. U principu, funkcije treba da obavljaju samo jedan zadatak i da budu kratke. Tekst jedne funkcije treba da staje na jedan ili dva ekrana (tj. da ima manje od pedesetak redova), radi dobre preglednosti. Duge funkcije poželjno je podeliti na manje, na primer, one koje obrađuju specijalne slučajeve. Ukoliko je brzina izvršavanja kritična, kompilatoru se može naložiti da *inlajnuje* funkcije (da prilikom kompilacije umetne kôd kratkih funkcija na pozicije gde su pozvane).

Da li funkcija ima razuman obim često govori broj lokalnih promenljivih: ako ih ima više od, na primer, 10, verovatno je funkciju poželjno podeliti na nekoliko manjih. Slično važi i za broj parametara funkcije.

2.9.2 Modularnost i podela na datoteke

Veliki programi sastoje se od velikog broja datoteka koje bi trebalo da budu organizovane na razuman način u direktorijume. Jednu datoteku treba da čine definicije funkcija koje su međusobno povezane i predstavljaju nekakvu celinu.

Datoteke zaglavlja obično imaju sledeću strukturu:

- definicije tipova;
- definicije konstanti;
- deklaracije globalnih promenljivih (uz navođenje kvalifikatora `extern`);
- deklaracije funkcija (uz navođenje kvalifikatora `extern`).

a izvorne datoteke sledeću strukturu:

- uključivanje sistemskih datoteka zaglavlja;
- uključivanje lokalnih datoteka zaglavlja;
- definicije tipova;
- definicije konstanti;
- deklaracije/definicije globalnih promenljivih;
- definicije funkcija.

Organizacija u datoteke treba da bude takva da izoluje delove koji se trenutno menjaju i razdvoji ih od delova koji su stabilne, zaokružene celine. U fazi razvoja, često je preporučljivo čak i napraviti privremeni deo organizacije kako bi modul koji se trenutno razvija bio izolovan i razdvojen od drugih delova.

Program treba deliti na datoteke imajući u vidu delom suprotstavljene zahteve. Jedna datoteka ne treba da bude duža od nekoliko, na primer - dve ili tri, stotine linija. Ukoliko logička struktura programa nameće dužu datoteku, onda vredi preispitati postojeću organizaciju podataka i funkcija. S druge strane, datoteke ne treba da budu prekratke i treba da predstavljaju zaokružene celine. Preterana usitnjenost (u preveliki broj datoteka) može da oteža upravljanje programom i njegovu razumljivost.

Rad sa programima organizovanih u više datoteka olakšavaju integrisana razvojna okruženja i program `make` (videti Deo I, **??**).

2.9.3 Primer

TODO

Pitanja i zadaci za vežbu

Pitanje 2.1. *Navedite barem jedan alat za kontrolu verzija.*

Pitanje 2.2. *Šta su CVS, SVN, Git, Bazaar, Mercurial, Visual SourceSafe?*

Pitanje 2.3. *Navedite dva načina za imenovanje funkcije koja bi trebalo da se zove "izracunavanje finalne ocene".*

Pitanje 2.4. *Koje ime bi, u kamiljoj notaciji, nosila promenljiva* `int broj_cvorova`*?*

Pitanje 2.5. *Koje ime bi, u mađarskoj notaciji, nosile promenljive* `float* X` *i* `int* broj_cvorova` *?*

Pitanje 2.6. *Zaokružiti deklaracije koje su u skladu sa mađarskom notacijom:*

```
char cKontrolniKarakter;
char cKontrolni_karakter;
char kontrolni_karakter;
int iKoeficijent;
int iKamatnaStopa;
int kamatna_stopa;
```

Pitanje 2.7. *Navedite dva načina za imenovanje promenljive koja označava ukupan broj klijenata.*

Pitanje 2.8. *Koliko se preporučuje da najviše ima karaktera u jednoj liniji programa i koji su razlozi za to?*

Pitanje 2.9. *Ukoliko neka linija našeg programa ima 300 karaktera, šta nam to sugeriše?*

Pitanje 2.10. *Kada je prihvatljivo da jedna linija programa ima više naredbi?*

Pitanje 2.11. *U kojim situacijama se obično neka linija programa ostavlja praznom?*

Pitanje 2.12. *U kojem stilu programiranja nazubljivanje teksta programa ima najviše smisla?*

Pitanje 2.13. *Koliko belina zamenjuje jedan tab karakter?*

Pitanje 2.14. *Zašto za nazubljivanje teksta programa nije preporučljivo koristiti tabulator?*

Pitanje 2.15. *Napisati sledeće naredbe na prihvatljiviji način:*

```
if ( !(c == 'y' || c == 'Y') )
   return;

length = (length < BUFSIZE) ? length : BUFSIZE;

flag = flag ? 0 : 1;
```

Pitanje 2.16. *Šta su „magične konstante“ i da li one popravljaju ili kvare kvalitet programa?*

Pitanje 2.17. *Kako se izbegava korišćenje „magičnih konstanti“ u programu?*

Pitanje 2.18. *Šta je, kako bi kod bio lakši za održavanje, bolje koristiti umesto deklaracije* `char ImeKorisnika[50]` *?*

Pitanje 2.19. *Navedite barem jedan alat za automatsko generisanje tehničke dokumentacije.*

Pitanje 2.20. *Kojim se komentarom obično označava mesto u kodu na kojem treba dodati kôd za neki podzadatak?*

Pitanje 2.21. *Koji se marker u okviru komentara u kodu obično koristi za označavanje potencijalnih propusta i/ili grešaka koje naknadno treba ispraviti?*

Pitanje 2.22. *Koji je preporučeni obim jedne datoteke programa?*

Glava 3

Efikasnost programa i složenost izračunavanja

Pored svojstva ispravnosti programa, veoma je važno i pitanje koliko program zahteva vremena (ili izvršenih instrukcija) i memorije za svoje izvršavanje. Često se algoritmi ne izvršavaju isto niti zahtevaju isto memorije za sve ulazne vrednosti, pa je potrebno naći način za opisivanje i poređenje *efikasnosti* (ili *složenosti*) različitih algoritama. Obično se razmatraju:

- vremenska složenost algoritma;
- prostorna (memorijska) složenost algoritma.

Vremenska i prostorna složenosti mogu se razmatrati

- u terminima konkretnog vremena/prostora utrošenog za neku konkretnu ulaznu veličinu;
- u terminima asimptotskog ponašanja vremena/prostora kada ulazna veličina raste.

U nekim situacijama vremenska ili prostorna složenost programa nisu mnogo važne (ako se zadatak izvršava brzo, ne zahteva mnogo memorije, ima dovoljno raspoloživog vremena itd), ali u nekim je vredna ušteda svakog sekunda ili bajta. I u takvim situacijama dobro je najpre razviti najjednostavniji program koji obavlja dati zadatak, a onda ga modifikovati ako je potrebno da se uklopi u zadata vremenska ili prostorna ograničenja.

Vremenska složenost algoritma određuje i njegovu praktičnu upotrebljivost tj. najveću dimenziju ulaza za koju je moguće da će se algoritam izvršiti u nekom razumnom vremenu. Analogno važi i za prostornu složenost.

U nastavku teksta najčešće će se govoriti o vremenskoj složenosti algoritama, ali u većini situacija potpuno analogna razmatranja mogu se primeniti na prostornu složenost.

3.1 Merenje i procenjivanje korišćenih resursa

Vreme izvršavanja programa može biti procenjeno ili izmereno za neke konkretne ulazne vrednosti i neko konkretno izvršavanje.

3.1.1 Merenje utrošenog vremena

Najjednostavnija mera vremenske efikasnosti programa (ili neke funkcije) je njegovo neposredno vreme izvršavanja za neke konkretne vrednosti. U standardnoj biblioteci raspoloživa je, kroz datoteku zaglavlja `time.h` jednostavna podrška za merenje vremena. Struktura `time_t` služi za reprezentovanje vremena, ali standard ne propisuje na koji način se reprezentuje vreme. Obično se vreme reprezentuje kao broj sekundi od početka godine 1970-e. Funkcija `time`:

```
time_t time(time_t *time);
```

vraća vreme, izraženo u sekundama, proteklo od početka *epohe*, obično od početka godine 1970-e.

Funkcija `difftime`:

```
double difftime(time_t end, time_t start);
```

vraća razliku između dva vremena, izraženu u sekundama. Ta razlika je obično celobrojna (ako je tip `time_t` celobrojan), ali je, zbog opštosti, povratni tip ipak `double`.

Struktura `clock_t` služi za reprezentovanje vremena rada tekućeg procesa, ali standard ne propisuje na koji način se reprezentuje. Funkcija `clock`:

```
clock_t clock();
```

vraća trenutno vreme, ali zbog različitosti mogućih implementacija, jedino ima smisla razmatrati razliku dva vremena dobijena ovom funkcijom. Razlika predstavlja broj vremenskih „tikova“, specifičnih za dati sistem, pa se broj sekundi između dva vremena može dobiti kao razlika dve vrednosti tipa `clock_t` podeljena konstantom `CLOCKS_PER_SEC`. Vreme izmereno na ovaj način je vreme koje je utrošio sam program (a ne i drugi programi koji se istovremeno izvršavaju) i može se razlikovati od proteklog apsolutnog vremena (može biti kraće ako, na primer, ima više programa koji rade istovremeno, ili duže ako, na primer, program koristi više raspoloživih jezgara procesora).

Navedene funkcije mogu se koristiti za merenje vremena koje troši neka funkcija ili operacija. Precizniji rezultati se dobijaju ako se meri izvršavanje koje se ponavlja veliki broj puta. S druge strane, umesto u sekundama, vreme koje troši neka funkcija ili operacija često se pogodnije izražava u nanosekundama kao u narednom primeru. U merenjima ovog tipa treba biti oprezan jer i naredbe petlje troše nezanemariv udeo vremena. Dodatno, prilikom kompilacije treba isključiti sve optimizacije (na primer, za kompilator `gcc`, to je opcija `-O0`) kako se ne bi merilo vreme za optimizovanu verziju programa. Naredni program ilustruje merenje vremena koje troši funkcija `f`.

Program 3.1.

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

#define BROJ_ITERACIJA  1000000000
#define NS_U_SEKUNDI    1000000000 /* broj ns u sekundi */

int f(void) {
...
}

int main()  {
  int i;
  double dt;

  clock_t t = clock();
  for(i=0; i<BROJ_ITERACIJA; i++)
     f();

  dt=((double)NS_U_SEKUNDI*(clock() - t))/CLOCKS_PER_SEC/BROJ_ITERACIJA;
  printf("Utroseno vreme: %.2lf ns\n",dt);
  return 0;
}
```

U zaglavlju `time.h` postoje i funkcije koje pružaju podršku za baratanje datumima, ali one nisu relevantne za merenje vremena rada programa pa neće ovde biti opisivane.

3.1.2 Procenjivanje potrebnog vremena

Vreme izvršavanja pojedinih delova koda (na nekom konkretnom računaru) može da se proceni ukoliko je raspoloživa procena vremena izvršavanja pojedinih operacija. Na primer, na računaru sa procesorom Intel Core i7-2670QM 2.2GHz koji radi pod operativnim sistemom Linux, operacija sabiranja dve vrednosti tipa `int` troši oko trećinu nanosekunde, a operacija množenja oko jednu nanosekundu (dakle, za jednu sekundu se na tom računaru može izvršiti oko milijardu množenja celih brojeva). Procena vremena izvršavanja pojedinih operacija i kontrolnih struktura na konkretnom računaru može se napraviti na način opisan u poglavlju 3.1.1.

Procene vremena izvršavanja programa na osnovu procena vremena izvršavanja pojedinačnih instrukcija treba uzimati sa velikom rezervom jer možda ne uzimaju u obzir sve procese koji se odigravaju tokom izvršavanja programa. Dodatno, treba imati na umu da vreme izmereno na jednom konkretnom računaru zavisi i od operativnog sistema pod kojim računar radi, od jezika i od kompilatora kojim je napravljen izvršni program za testiranje, itd.

3.1.3 Profajliranje

Postoje mnogi alati za analizu i unapređenje performansi programa i najčešće se zovu *profajleri* (engl. profiler). Njihova osnovna uloga je da pruže podatke o tome koliko puta je koja funkcija pozvana tokom (nekog konkretnog) izvršavanja, koliko je utrošila vremena i slično. Ukoliko se razvijeni program ne izvršava željeno brzo, potrebno je unaprediti neke njegove delove. Prvi kandidati za izmenu su delovi koji troše najviše vremena.

Za operativni sistem Linux, popularan je sistem `valgrind` koji objedinjuje mnoštvo alata za dinamičku analizu rada programa, uključujući profajler `callgrind`. Profajler `callgrind` se poziva na sledeći način:

```
valgrind --tool=callgrind mojprogram argumenti
```

gde `mojprogram` označava izvršni program koji se analizira, a `argumenti` njegove argumente (komandne linije). Program `callgrind` izvršava zadati program `mojprogram` sa argumentima `argumenti` i registruje informacije o tome koja funkcija je pozivala koje funkcije (uključujući sistemske funkcije i funkcije iz standardne biblioteke), koliko je koja funkcija utrošila vremena itd. Detaljniji podaci se mogu dobiti ako je program preveden u debag režimu (`gcc` kompilatorom, debag verzija se dobija korišćenjem opcije `-g`). Prikupljene informacije program `callgrind` čuva u datoteci sa imenom, na primer, `callgrind.out.4873`. Ove podatke može da na pregledan način prikaže, na primer, program `kcachegrind`:

```
kcachegrind callgrind.out.4873
```

Slika 3.1 ilustruje rad programa `kcachegrind`. Uvidom u prikazane podatke, programer može da uoči funkcije koje troše najviše vremena i da pokuša da ih unapredi i slično.

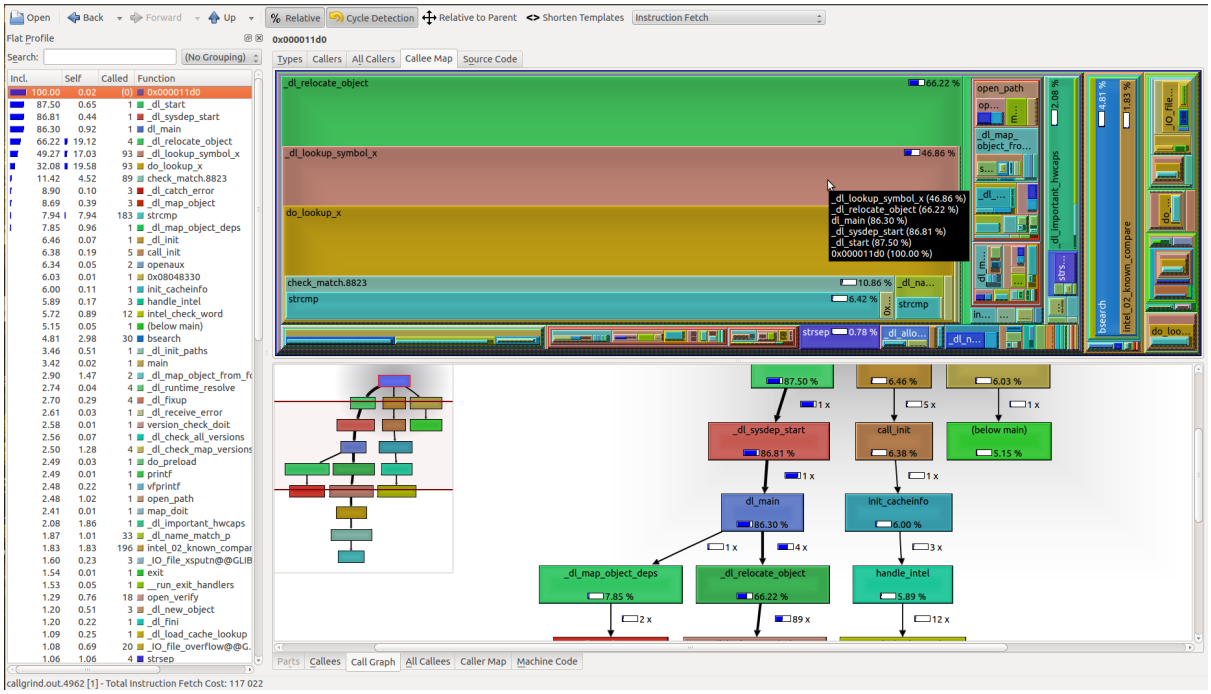
3.2 O notacija i red složenosti algoritma

Vreme izvršavanja programa može biti procenjeno ili izmereno za neke konkretne ulazne vrednosti i neko konkretno izvršavanje. No, vreme izvršavanja programa može biti opisano opštije, u vidu funkcije koja zavisi od ulaznih argumenata.

Često se algoritmi ne izvršavaju isto za sve ulaze istih veličina, pa je potrebno naći način za opisivanje i poređenje efikasnosti različitih algoritama. *Analiza najgoreg slučaja* zasniva procenu složenosti algoritma na najgorem slučaju (na slučaju za koji se algoritam najduže izvršava — u analizi vremenske složenosti, ili na slučaju za koji algoritam koristi najviše memorije — u analizi prostorne složenosti). Ta procena može da bude varljiva, ali predstavlja dobar opšti način za poređenje efikasnosti algoritama. U nekim situacijama moguće je izračunati prosečno vreme izvršavanja algoritma, ali i takva procena bi često mogla da bude varljiva. Analiziranje najboljeg slučaja, naravno, nema smisla. U nastavku će, ako nije rečeno drugačije, biti podrazumevana analiza najgoreg slučaja.

Neka je funkcija $f(n)$ jednaka broju instrukcija koje zadati algoritam izvrši za ulaz veličine n. Tabela 3.1 prikazuje potrebno vreme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu, tj. 0.001 mikrosekundi (μs).

Vodeći član u funkciji $f(n)$ određuje potrebno vreme izvršavanja. Tako, na primer, ako je broj instrukcija $n^2 + 2n$, onda za ulaz dimenzije 1000000, član n^2 odnosi 16.7 minuta dok član $2n$ odnosi samo dodatne dve mikrosekunde. Vremenska (a i prostorna) složenost je, dakle, skoro potpuno određena



Slika 3.1: Ilustracija prikaza u programu `kcachegrind` podataka dobijenih profajliranjem

$n \setminus f(n)$	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	$0.003\,\mu s$	$0.01\,\mu s$	$0.033\,\mu s$	$0.1\,\mu s$	$1\,\mu s$	$3.63\,ms$
20	$0.004\,\mu s$	$0.02\,\mu s$	$0.086\,\mu s$	$0.4\,\mu s$	$1\,ms$	$77.1\,god$
30	$0.005\,\mu s$	$0.03\,\mu s$	$0.147\,\mu s$	$0.9\,\mu s$	$1\,s$	$8.4 \times 10^{15}\,god$
40	$0.005\,\mu s$	$0.04\,\mu s$	$0.213\,\mu s$	$1.6\,\mu s$	$18.3\,min$	
50	$0.006\,\mu s$	$0.05\,\mu s$	$0.282\,\mu s$	$2.5\,\mu s$	$13\,dan$	
100	$0.007\,\mu s$	$0.1\,\mu s$	$0.644\,\mu s$	$10\,\mu s$	$4 \times 10^{13}\,god$	
1,000	$0.010\,\mu s$	$1.0\,\mu s$	$9.966\,\mu s$	$1\,ms$		
10,000	$0.013\,\mu s$	$10\,\mu s$	$130\,\mu s$	$100\,ms$		
100,000	$0.017\,\mu s$	$0.10\,\mu s$	$1.67\,ms$	$10\,s$		
1,000,000	$0.020\,\mu s$	$1\,ms$	$19.93\,ms$	$16.7\,min$		
10,000,000	$0.023\,\mu s$	$0.01\,s$	$0.23\,s$	$1.16\,dan$		
100,000,000	$0.027\,\mu s$	$0.10\,s$	$2.66\,s$	$115.7\,dan$		
1,000,000,000	$0.030\,\mu s$	$1\,s$	$29.9\,s$	$31.7\,god$		

Tabela 3.1: Ilustracija vremena izvršavanja

„vodećim" (ili „dominantnim") članom u izrazu koji određuje broj potrebnih instrukcija. Na upotrebljivost algoritma ne utiču mnogo ni multiplikativni i aditivni konstantni faktori u broju potrebnih instrukcija, koliko asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza. Ovakav pojam složenosti uvodi se sledećom definicijom.

Definicija 3.1. *Ako postoje pozitivna konstanta c i prirodan broj N takvi da za funkcije f i g nad prirodnim brojevima važi*

$$f(n) \leq c \cdot g(n) \quad \textit{za sve vrednosti } n \textit{ veće od } N$$

onda pišemo

$$f = O(g)$$

i čitamo „f je veliko ‚o' od g".

Naglasimo da O ne označava neku konkretnu funkciju, već klasu funkcija i uobičajeni zapis $f = O(g)$ zapravo znači $f \in O(g)$.

Lako se pokazuje da aditivne i multiplikativne konstante ne utiču na klasu kojoj funkcija pripada (na primer, u izrazu $5n^2 + 1$, za konstantu 1 kažemo da je aditivna, a za konstantu 5 da je multiplikativna). Zato se može reći da je neki algoritam složenosti $O(n^2)$, ali se obično ne govori da pripada, na primer, klasi $O(5n^2 + 1)$, jer aditivna konstanta 1 i multiplikativna 5 nisu od suštinske važnosti. Zaista, ako jedan algoritam zahteva $5n^2 + 1$ instrukcija, a drugi n^2, i ako se prvi algoritam izvršava na računaru koji je šest puta brži od drugog, on će biti brže izvršen za svaku veličinu ulaza. No, ako jedan algoritam zahteva n^2 instrukcija, a drugi n, ne postoji računar na kojem će prvi algoritam da se izvršava brže od drugog za svaku veličinu ulaza.

Primer 3.1. *Može se dokazati da važi:*

- $n^2 = O(n^2)$
- $n^2 + 10 = O(n^2)$
- $10 \cdot n^2 + 10 = O(n^2)$
- $10 \cdot n^2 + 8n + 10 = O(n^2)$
- $n^2 = O(n^3)$
- $2^n = O(2^n)$
- $2^n + 10 = O(2^n)$
- $10 \cdot 2^n + 10 = O(2^n)$
- $2^n + n^2 = O(2^n)$
- $3^n + 2^n = O(3^n)$
- $2^n + 2^n n = O(2^n n)$

Važe sledeća svojstva:

- ako važi $f_1 = O(g_1)$ i $f_2 = O(g_2)$, onda važi i $f_1 f_2 = O(g_1 g_2)$.
- ako važi $f_1 = O(g_1)$ i $f_2 = O(g_2)$, onda važi i $f_1 + f_2 = O(|g_1| + |g_2|)$.
- ako važi $f = O(g)$ i a i b su pozitivne konstante, onda važi i $af + b = O(g)$.

Definicija 3.2. *Ako postoje pozitivne konstante c_1 i c_2 i prirodan broj N takvi da za funkcije f i g nad prirodnim brojevima važi*

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ za sve vrednosti } n \text{ veće od } N$$

onda pišemo

$$f = \Theta(g)$$

i čitamo „f je veliko 'teta' od g“.

Ako važi $f = \Theta(g)$, onda važi i $f = O(g)$ i $g = O(f)$.

Primer 3.2. *Može se dokazati da važi:*

- $10 \cdot 2^n + 10 = \Theta(2^n)$
- $2^n + 2^n n = \Theta(2^n n)$

Definicija 3.3. *Ako je $T(n)$ vreme izvršavanja algoritma A (čiji ulaz karakteriše prirodan broj n) i ako važi $T = O(g)$, onda kažemo da je algoritam A* složenosti *ili* reda $O(g)$ *ili da algoritam A* pripada klasi $O(g)$.

Analogno prethodnoj definiciji definiše se kada algoritam A pripada klasi $\Theta(g)$. Takva informacija obezbeđuje da algoritam koji zahteva, na primer, $5n$ koraka jeste reda $\Theta(n)$, ali ne i reda $\Theta(n^2)$. S druge strane, za taj algoritam može da se smatra da je reda n ali i reda $O(n^2)$ (mada ne i reda, na primer, $O(\log n)$). Informacija o složenosti algoritma u terminima Θ (koja daje i gornju i donju granicu) je preciznija nego informacija u terminima O (koja daje samo gornju granicu). Često je jednostavnije složenost algoritma iskazati u terminima O nego u terminima Θ. Obično se za složenost algoritma koristi O notacija i onda kada se složenost može iskazati u terminima Θ. Kada se kaže da algoritam pripada klasi $O(g)$ obično se podrazumeva da je g najmanja takva klasa (ili makar — najmanja za koju se to može dokazati). I O i Θ notacija se koriste i u analizi najgoreg slučaja i u analizi prosečnog slučaja.

Priroda parametra klase složenosti (na primer, n u $O(n)$ ili m u $O(2^{m+k})$) zavisi od samog algoritma. Složenost nekih algoritama zavisi od vrednosti argumenata, a nekih od broja argumenata. Na primer,

složenost funkcije za izračunavanje faktorijela ulazne vrednosti m zavisi od m i jednaka je (za razumnu implentaciju) $O(m)$. Složenost funkcije koja računa prosek k ulaznih brojeva ne zavisi od vrednosti tih brojeva, već samo od toga koliko ih ima i jednaka je $O(k)$. Složenost funkcije koja sabira dva broja je konstantna tj. pripada klasi $O(1)$. Složenost izračunavanja neke funkcije može da zavisi i od više parametara. Na primer, algoritam koji za n ulaznih tačaka proverava da li pripadaju unutrašnjosti m ulaznih trouglova, očekivano ima složenost $O(mn)$.

Za algoritme složenosti $O(n)$ kažemo da imaju linarnu složenost, za $O(n^2)$ kvadratnu, za $O(n^3)$ kubnu, za $O(n^k)$ za neko k polinomijalnu, a za $O(\log n)$ logaritamsku.

3.3 Izračunavanje složenosti funkcija

Izračunavanje (vremenske i prostorne) složenosti funkcija se zasniva na određivanju tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrednosti je najčešće veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primer, u ovom kontekstu pojam „jedinična instrukcija“ se obično pojednostavljuje, te se može smatrati da, na primer, i poredjenje i sabiranje i množenje i druge pojedinačne naredbe troše po jednu vremensku jedinicu. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, multiplikativni faktori ne utiču na red algoritma).

Ukoliko se deo programa sastoji od nekoliko instrukcija bez grananja, onda se procenjuje da je njegovo vreme izvršavanja uvek isto, konstantno, te da pripada klasi $O(1)$. Ukoliko deo programa sadrži petlju koja se izvršava n puta, a vreme izvršavanja tela petlje je konstantno, onda ukupno vreme izvršavanja petlje pripada klasi $O(n)$. Ukoliko deo programa sadrži jednu petlju koja se izvršava m puta i jednu petlju koja se izvršava n puta, a vremena izvršavanja tela ovih petlji su konstantna, onda ukupno vreme izvršavanja petlje pripada klasi $O(m+n)$. Generalno, ukoliko program ima dva dela, složenosti $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je $O(f+g)$. Ukoliko deo programa sadrži dvostruku petlju – jednu koja se izvršava m puta i, unutar nje, drugu koja se izvršava n puta i ukoliko je vreme izvršavanja tela unutrašnje petlje konstantno, onda ukupno vreme izvršavanja petlje pripada klasi $O(m \cdot n)$. Ukoliko deo programa sadrži jedno grananje i ukoliko vreme izvršavanja jedne grane pripada klasi $O(m)$ a druge grane pripada klasi $O(n)$, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(m+n)$.

Analogno se računa složenost za druge vrste kombinovanja linearnog koda, grananja i petlji. Za izračanavanje složenosti rekurzivnih funkcija potreban je matematički aparat za rešavanje *rekurentnih jednačina*, opisan u narednom poglavlju.

3.4 Rekurentne jednačine

Kod rekurzivnih funkcija, vreme $T(n)$ potrebno za izračunavanje vrednosti funkcije za ulaz dimenzije n se može izraziti kao zbir vremena izračunavanja za sve rekurzivne pozive za ulaze manje dimenzije i vremena potrebnog za pripremu rekurzivnih poziva i objedinjavanje rezultata. Tako se, obično jednostavno, može napisati veza oblika

$$T(n) = T(n_1) + \ldots + T(n_k) + c,$$

gde rekurzivna funkcija za ulaz dimenzije n vrši k rekurzivnih poziva za ulaze (ne obavezno različitih) dimenzija $n_1, \ldots, n_k$, dok je vreme potrebno za pripremu poziva i objedinjavanje rezultata c.

U nekim slučajevima iz ovakve *linearne rekurentne relacije* može se eksplicitno izračunati nepoznati niz $T(n)$. U nekim slučajevima eksplicitno rešavanje jednačine nije moguće, ali se može izračunati asimptotsko ponašanje niza $T(n)$.

Homogena rekurentna jednačina prvog reda. Razmotrimo jednačinu oblika

$$T(n) = aT(n-1),$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$. Jednostavno se pokazuje da je rešenje ove jednačine geometrijski niz $T(n) = ca^n$.

Homogena rekurentna jednačina drugog reda. Razmotrimo jednačinu oblika

$$T(n) = aT(n-1) + bT(n-2),$$

za $n > 1$, pri čemu su date vrednosti za $T(0) = c_0$ i $T(1) = c_1$.

Ukoliko nisu navedeni početni uslovi, jednačina ima više rešenja. Zaista, ukoliko nizovi $T_1(n)$ i $T_2(n)$ zadovoljavaju jednačinu, tada jednačinu zadovoljava i njihova proizvoljna linearna kombinacija $T(n) = \alpha T_1(n) + \beta T_2(n)$:

$$\begin{aligned} T(n) &= \alpha T_1(n) + \beta T_2(n) \\ &= \alpha(aT_1(n-1) + bT_1(n-2)) + \beta(aT_2(n-1) + bT_2(n-2)) \\ &= a(\alpha T_1(n-1) + \beta T_2(n-1)) + b(\alpha T_1(n-2) + \beta T_2(n-2)) \\ &= aT(n-1) + bT(n-2)\ . \end{aligned}$$

S obzirom na to da i nula niz (niz čiji su svi elementi nule) trivijalno zadovoljava jednačinu, skup rešenja čini vektorski prostor.

Razmotrimo funkcije oblika t^n i pokušajmo da proverimo da li postoji broj t takav da t^n bude rešenje date jednačine. Za takvu vrednosti bi važilo:

$$t^n = a \cdot t^{n-1} + b \cdot t^{n-2},$$

odnosno, posle množenja sa t^2 i deljenja sa t^n:

$$t^2 = at + b.$$

Dakle, da bi t^n bilo rešenje jednačine, potrebno je da t bude koren navedene kvadratne jednačine, koja se naziva *karakteristična jednačina* za *homogenu rekurentnu jednačinu drugog reda.*

Ako su t_1 i t_2 različiti koreni ove jednačine, može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i t_2^n, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot t_2^n\ ,$$

tj. da ove dve funkcije čine bazu pomenutog vektorskog prostora rešenja. Ako se želi pronaći ono rešenje koje zadovoljava zadate početne uslove (tj. zadovoljava date vrednosti $T(0) = c_0$ i $T(1) = c_1$), onda se vrednosti koeficijenata α i β mogu dobiti rešavanjem sistema dobijenog za $n = 0$ i $n = 1$, tj. rešavanjem sistema jednačina $c_0 = \alpha + \beta$, $c_1 = \alpha \cdot t_1 + \beta \cdot t_2$.

U slučaju da je t_1 dvostruko rešenje karakteristične jednačine, može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i $n \cdot t_1^n$, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot n \cdot t_1^n\ .$$

Koeficijenti α i β koji određuju partikularno rešenje koje zadovoljava početne uslove, takođe se dobijaju rešavanjem sistema za $n = 0$ i $n = 1$.

Primer 3.3. *Neka za vreme izvršavanja $T(n)$ algoritma A (gde n odredjuje ulaznu vrednost za algoritam) važi $T(n+2) = 4T(n+1) - 4T(n)$ (za $n \geq 1$) i $T(1) = 6, T(2) = 20$.*

Složenost algoritma A može se izračunati na sledeći način. Karakteristična jednačina za navedenu homogenu rekurentntu vezu je

$$t^2 = 4t - 4$$

i njen dvostruki koren je $t_1 = 2$. Opšti član niza $T(n)$ može biti izražen u obliku

$$T(n) = \alpha \cdot t_1^n + \beta \cdot n \cdot t_1^n\ .$$

tj.

$$T(n) = \alpha \cdot 2^n + \beta \cdot n \cdot 2^n\ .$$

Iz $T(1) = 6, T(2) = 20$ dobija se sistem

$$\alpha \cdot 2 + \beta \cdot 2 = 6$$

$$\alpha \cdot 4 + \beta \cdot 8 = 20$$

čije je rešenje $(\alpha, \beta) = (1, 2)$, pa je $T(n) = 2^n + 2 \cdot n \cdot 2^n$, odakle sledi da je $T(n) = O(n \cdot 2^n)$.

Homogena rekurentna jednačina reda k. Homogena rekurentna jednačina reda k (gde k može da bude i veće od 2) je jednačina oblika:

$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + \ldots + a_k \cdot T(n-k),$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0$, $T(1) = c_1$, ..., $T(k-1) = c_{k-1}$.

Tehnike prikazane na homogenoj jednačini drugog reda, lako se uopštavaju na jednačinu proizvoljnog reda k. Karakteristična jednačina navedene jednačine je:

$$t^k = a_1 \cdot t^{k-1} + a_2 \cdot t^{k-2} + \ldots + a_k.$$

Ako su rešenja t_1, t_2, ..., t_k sva različita, onda je opšte rešenje polazne jednačine oblika:

$$T(n) = \alpha_1 \cdot t_1^n + \alpha_2 \cdot t_2^n + \ldots + \alpha_k \cdot t_k^n,$$

pri čemu se koeficijenti α_i mogu dobiti iz početnih uslova (kada se u navedeno opšte rešenje za n uvrste vrednosti 0, 1, ..., $k-1$).

Ukoliko je neko rešenje t_1 dvostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n i $n \cdot t_1^n$. Ukoliko je neko rešenje t_1 trostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n, $n \cdot t_1^n$, $n^2 \cdot t_1^n$, itd.

Nehomogena rekurentna jednačina prvog reda. Razmotrimo jednačinu oblika

$$T(n) = aT(n-1) + b,$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$.

Definišimo niz $T'(n) = T(n+1) - T(n)$ za $n \geq 0$. Za $n > 0$ važi $T'(n) = (aT(n)+b) - (aT(n-1)+b) = aT'(n-1)$. Za $n = 0$ važi $T'(0) = T(1) - T(0) = ac + b - c$. Dakle, niz $T'(n)$ zadovoljava homogenu jednačinu prvog reda čije je rešenje $T'(n) = ((ac+b-c)a^n$.

Za $a \neq 1$ važi

$$\begin{aligned} T(n) &= T(0) + \sum_{k=0}^{n-1} T'(k) = c + \sum_{k=0}^{n-1} ((ac+b-c)a^k \\ &= c + (ac+b-c)\frac{a^n - 1}{a-1} = b\frac{a^n-1}{a-1} + ca^n \end{aligned}$$

Drugi način rešavanja ovog tipa jednačina je svođenje na homogenu jednačinu drugog reda. Iz $T(1) = aT(0)+b$, sledi da je $T(1) = ac+b$. Iz $T(n) = aT(n-1)+b$ i $T(n+1) = aT(n)+b$, sledi $T(n+1)-T(n) = (aT(n)+b) - (aT(n-1)+b) = aT(n) - aT(n-1)$ i, dalje, $T(n+1) = (a+1)T(n) - aT(n-1)$, za $n > 0$. Rešenje novodobijene homogene jednačine se može dobiti na gore opisani način (jer su poznate i početne vrednosti $T(0) = c$ i $T(1) = ac+b$).

Nehomogena rekurentna jednačina reda k. Nehomogena rekurentna jednačina reda k ($k > 0$) oblika:

$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + \ldots + a_k \cdot T(n-k) + c,$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0$, $T(1) = c_1$, ..., $T(k-1) = c_{k-1}$, može se rešiti svođenjem na homogenu rekurentnu jednačinu reda $k+1$, analogno gore opisanom slučaju za $k = 1$.

Nehomogena rekurentna jednačina oblika $T(n) = aT(n/b) + cn^k$. Naredna teorema govori o asimptotskom ponašanju rešenja nehomogene rekurentne jednačine oblika $T(n) = aT(n/b) + cn^k$.

Teorema 3.1 (Master teorema)**.** *Rešenje rekurentne relacije*

$$T(n) = aT(n/b) + cn^k \ ,$$

gde su a i b celobrojne konstante ($a \geq 1$, $b \geq 1$) i c i k pozitivne konstante je

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) \ , & \textit{ako je } a > b^k \\ \Theta(n^k \log n) \ , & \textit{ako je } a = b^k \\ \Theta(n^k) \ , & \textit{ako je } a < b^k \end{cases}$$

3.5 Primeri izračunavanja složenosti funkcija

3.5.1 Faktorijel

Izračunajmo vremensku složenost naredne rekurzivne funkcije:

```
unsigned faktorijel(unsigned n) {
    if (n == 0)
        return 1;
    else
        return n*faktorijel(n-1);
}
```

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `faktorijel` za ulaznu vrednost `n`. Za $n = 0$, važi $T(n) = 2$ (jedno poređenje i jedna naredba `return`). Za $n > 0$, važi $T(n) = 4 + T(n-1)$ (jedno poređenje, jedno množenje, jedno oduzimanje, broj instrukcija koje zahteva funkcija `faktorijel` za argument `n-1` i jedna naredba `return`). Dakle,

$$T(n) = 4 + T(n-1) = 4 + 4 + T(n-2) = \ldots = \underbrace{4 + 4 + \ldots + 4}_{n} + T(0) = 4n + 2 = O(n)$$

Dakle, navedena funkcija ima linearnu vremensku složenost.

Nehomogena jednačina $T(n) = 4 + T(n-1)$ je mogla biti rešena i svođenjem na homogenu jednačinu. Iz $T(n) = 4 + T(n-1)$ i $T(n+1) = 4 + T(n)$ sledi $T(n+1) - T(n) = T(n) - T(n-1)$ i $T(n+1) = 2T(n) - T(n-1)$. Karakteristična jednačina $t^2 = 2t - 1$ ima dvostruko rešenje $t_1 = 1$, pa je opšte rešenje oblika $T(n) = a \cdot 1^n + b \cdot n \cdot 1^n = a + nb = O(n)$.

3.5.2 Fibonačijev niz

Za elemente Fibonačijevog niza važi $F(0) = 0$, $F(1) = 1$ i $F(n) = F(n-1) + F(n-2)$, za $n > 1$. Karakteristična jednačina je $t^2 = t + 1$ i njeni koreni su $\frac{1+\sqrt{5}}{2}$ i $\frac{1-\sqrt{5}}{2}$, pa je opšte rešenje oblika

$$F(n) = \alpha \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

Koristeći početne uslove, može se izračunati opšti član niza:

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

Funkcija za izračunavanje n-tog elementa Fibonačijevog niza može se definisati na sledeći način:

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

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `fib` za ulaznu vrednost `n`. Za $n \leq 1$ važi $T(n) = 2$ (jedno poređenje i jedna naredba `return`). Za $n > 1$, važi $T(n) = T(n-1) + T(n-2) + 5$ (jedno poređenje, dva oduzimanja, broj instrukcija koje zahtevaju pozivi funkcije za $n-1$ i $n-2$, jedno sabiranje i jedna naredba `return`). Iz $T(n) = T(n-1) + T(n-2) + 5$ i $T(n+1) = T(n) + T(n-1) + 5$, sledi $T(n+1) = 2T(n) - T(n-2)$. Karakteristična jednačina ove jednačine je $t^3 = 2t^2 - 1$ i njeni koreni su 1, $\frac{1+\sqrt{5}}{2}$ i $\frac{1-\sqrt{5}}{2}$, pa je opšte rešenje oblika

$$T(n) = a \cdot 1^n + b \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + c \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

odakle sledi da je $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

3.5.3 Uzajamna rekurzija

Složenost algoritama u kojima se javlja uzajamna rekurzija može se izračunati svođenjem na prikazane tehnike. To će biti ilustrovano primerom.

Algoritam A izvršava se za vrednost n ($n > 1$) primenom istog algoritma za vrednost $n-1$, pri čemu se za svodjenje problema koristi algoritam B za vrednost $n-1$. Algoritam B izvršava se za vrednost n ($n > 1$) trostrukom primenom istog algoritma za vrednost $n-1$, pri čemu se za svodjenje problema koristi algoritam A za vrednost $n-1$. Algoritmi A i B se za $n = 1$ izvršavaju jednu vremensku jedinicu. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.

Neka je $A(n)$ vreme izvršavanja algoritma A za ulaznu vrednost n i neka je $B(n)$ vreme izvršavanja algoritma B za ulaznu vrednost n. Na osnovu uslova zadatka važi:

$$A(1) = B(1) = 1 \tag{3.1}$$

$$A(n) = A(n-1) + B(n-1) \ \ (n > 1) \tag{3.2}$$

$$B(n) = 3B(n-1) + A(n-1) \ \ (n > 1) \tag{3.3}$$

Iz jednakosti (2) imamo:

$$B(n) = A(n+1) - A(n) \tag{3.4}$$

$$B(n-1) = A(n) - A(n-1) \tag{3.5}$$

pa, uvrštavanjem u jednakost (3), za $n > 1$ važi:

$$A(n+1) - 4A(n) + 2A(n-1) = 0 \tag{3.6}$$

Pomoću ove rekurentne veze i početnih uslova: $A(1) = 1$ i $A(2) = A(1) + B(1) = 2$, možemo odrediti vrednost $A(n)$. Karakteristična jednačina relacije (6) je:

$$x^2 - 4x + 2 = 0$$

Njeni koreni su $x_1 = 2 + \sqrt{2}$ i $x_2 = 2 - \sqrt{2}$, pa je opšte rešenje rekurentne jednačine (6) oblika:

$$A(n) = c_1(2 + \sqrt{2})^n + c_2(2 - \sqrt{2})^n$$

za neke $c_1, c_2 \in \mathbf{R}$. Konstante c_1 i c_2 odredjujemo pomoću početnih uslova, rešavajući sledeći sistem linearnih jednačina po c_1 i c_2:

$$1 = A(1) = c_1(2 + \sqrt{2}) + c_2(2 - \sqrt{2})$$

$$2 = A(2) = c_1(2 + \sqrt{2})^2 + c_2(2 - \sqrt{2})^2$$

Dobija se da je $c_1 = \frac{1}{4}(2 - \sqrt{2})$, a $c_2 = \frac{1}{4}(2 + \sqrt{2})$, pa je konačno:

$$A(n) = \frac{1}{2}((2 + \sqrt{2})^{n-1} + (2 - \sqrt{2})^{n-1})$$

3.6 Klase složenosti P i NP

Neka od najvažnijih otvorenih pitanja matematike i informatike vezana su za složenost izračunavanja i, takozvane, NP-kompletne probleme. Jedan takav problem biće opisan u nastavku teksta.

Šef protokola na jednom dvoru treba da organizuje bal za predstavnike ambasada. Kralj traži da na bal bude pozvan Peru ili da ne bude pozvan Katar (Qatar). Kraljica zahteva da budu pozvani Katar ili Rumunija (ili i Katar i Rumunija). Princ zahteva da ne bude pozvana Rumunija ili da ne bude pozvan Peru (ili da ne budu pozvani ni Rumunija ni Peru). Da li je moguće organizovati bal i zadovoljiti zahteve svih članova kraljevske porodice?

Ako su p, q i r bulovske (logičke) promenljive (koje mogu imati vrednosti *true* ili *false*, tj. $\top$ ili $\bot$), navedeni problem može biti formulisan na sledeći način: da li je *zadovoljiv* logički iskaz

$$(p \vee \neg q) \wedge (q \vee r) \wedge (\neg r \vee \neg p) \ .$$

Zadovoljivost navedenog iskaza može biti odredjena tako što bi bile ispitane sve moguće interpretacije – sve moguće dodele varijablama p, q i r. Ako je u nekoj interpretaciji vrednost datog logičkog iskaza *true*, onda je dati iskaz zadovoljiv. Za izabranu, fiksiranu interpretaciju može se u konstantnom vremenu utvrditi da li je istinitosna vrednost *true*. Za n promenljivih ima 2^n interpretacija, pa je red ovog algoritma za ispitivanje zadovoljivosti logičkih iskaza reda $O(2^n)$ (algoritam je eksponencijalne složenosti). Pitanje je da li postoji algoritam koji navedeni problem rešava u polinomijalnom vremenu.

Definicija 3.4. *Za algoritam sa ulaznom vrednošću n kažemo da je* polinomijalne složenosti *ako je njegovo vreme izvršavanja $O(P(n))$ gde je $P(n)$ polinom po n. Klasa polinomijalnih algoritama označava se sa P.*

Definicija 3.5 (Pojednostavljena definicija klase NP)**.** *Ako neki problem može da se predstavi u vidu najviše eksponencijalno mnogo instanci (u odnosu na dužinu ulaza) i ako za bilo koju instancu može da bude rešen u polinomijalnom vremenu, onda kažemo da problem pripada klasi NP.*

Očigledno je da važi $P \subseteq NP$, ali se još uvek ne zna[1] da li važi $P = NP$.

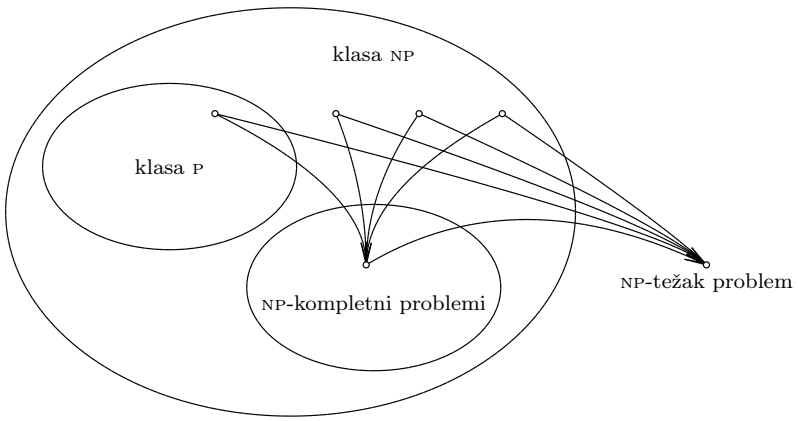
Ako bi se pokazalo da neki problem iz klase NP nema polinomijalno rešenje, onda bi to značilo da ne važi $P = NP$. Ako neki problem iz klase NP ima polinomijalno rešenje, onda to još ne znači da važi $P = NP$. Za probleme iz posebne potklase klase NP (to je klasa NP-kompletnih problema) važi da ako neki od njih ima polinomijalno rešenje, onda važi $P = NP$.

Primer 3.4. *Problem ispitivanja zadovoljivosti logičkih iskaza (SAT) pripada klasi NP, ali se ne zna da li pripada klasi P.*

Definicija 3.6. *Za problem X kažemo da je NP-težak problem ako je svaki NP problem polinomijalno svodljiv na X.*

Definicija 3.7. *Za problem X kažemo da je NP-kompletan problem ako pripada klasi NP i ako je NP-težak.*

Odnos problema iz klase P i NP i NP-teških problema ilustrovan je na slici 3.2. Skup NP-kompletnih problema jeste pravi podskup od NP, a trenutno se ne zna da li je P pravi podskup od NP.



Slika 3.2: Ilustracija odnosa problema iz klase P i NP i NP-teških problema

Teorema 3.2. *Ako bilo koji NP-težak problem pripada klasi P, onda važi $P = NP$.*

Teorema 3.3. *Problem X je NP-kompletan ako*

- *X pripada klasi NP*
- *Y je polinomijalno svodljiv na X, gde je Y neki NP-kompletan problem.*

[1] Matematički institut Klej nudi nagradu od milion dolara za svaki od izabranih sedam najznačajnijih otvorenih matematičkih i informatičkih problema. Problem da li su klase P i NP jednake je prvi na toj listi.

Dakle, ako znamo da je neki problem NP-kompletan, onda na osnovu prethodne teoreme možemo da pokažemo da su i svi NP-problemi na koje ga je moguće svesti takodje NP-kompletni. Dugo se tragalo za pogodnim NP-kompletnim problemom za koji bi moglo da se pronadje polinomijalno rešenje, što bi značilo da važi $P = NP$. Zato je bilo važno otkriti više raznorodnih NP-kompletnih problema. Postavljalo se pitanje da li uopšte postoji ijedan NP-kompletan problem (korišćenjem prethodne teoreme može da se utvrdi da je neki problem NP-kompletan samo ako se za neki drugi problem već zna da je NP-kompletan). Stiven Kuk je 1971. godine dokazao (neposredno, koristeći formalizam Tjuringove mašine) da je problem SAT NP-kompletan. U godinama koje su sledile za mnoge probleme je utvrdjeno da su takodje NP-kompletni (najčešće svodjenjem problema SAT na njih), ali ni za jedan od njih nije pokazano da pripada klasi P, pa se još uvek ne zna da li važi $P = NP$ (mada većina istraživača veruje da ne važi).

$PSPACE$ je klasa problema koji mogu biti rešeni korišćenjem memorijskog prostora koji je polinomijalna funkcija ulaza. Važi $NP \subseteq PSPACE$, ali se ne zna da li važi $NP = PSPACE$ (rasprostranjeno je uverenje da ne važi).

Tokom poslednje decenije, pored napora da se dokaže da ne važi $P = NP$, radi se i na ispitivanju raspodela najtežih problema u pojedinim klasama NP-kompletnih problema. Dosta se radi i na primeni novih pristupa u efikasnijem rešavanju nekih instanci NP-kompletnih problema.

3.7 Popravljanje vremenske složenosti

Ukoliko performanse programa nisu zadovoljavajuće, treba razmotriti zamenu ključnih algoritama – algoritama koji dominantno utiču na složenost. Ukoliko to ne uspeva tj. ukoliko se smatra da je asimptotsko ponašanje najbolje moguće, preostaje da se efikasnost programa popravi na polju broja pojedinačnih izvršenih naredbi (koje ne utiču na asimptotsko ponašanje, ali utiču na ukupno utrošeno vreme). Često ovaj cilj zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva:

Koristiti optimizacije kompilatora. Moderni kompilatori i u podrazumevanom režimu rada generišu veoma efikasan kod ali mogu da primene i dodatne tehnike optimizacije. Ipak, treba biti oprezan sa optimizacijama jer izvršni kôd ne odgovara direktno izvornom (na primer, moguće je da se u izvršnom programu ne predviđa prostor za promenljive za koje je ustanovljeno da se ne koriste), pa je moguće i da optimizacije izmene i očekivano ponašanje programa. Takođe, zbog narušene veze između izvornog i izvršnog programa, teško je ili nemoguće debagerom analizirati program. Kompilacija sa intenzivnim optimizovanjem je obično i znatno sporija od osnovne kompilacije. Zbog svega nabrojanog, preporučuje se da se optimizovanje programa primeni nakon intenzivnog testiranja i otklanjanja svih otkrivenih grešaka, ali i da se testiranje ponovi sa optimizovanom verzijom izvršnog programa.

Optimizacijama koje pruža kompilator, mogu se dobiti ubrzanja i od nekoliko desetina procenata, ali moguće je i da optimizovani program bude sporiji nego neoptimizovani.

Kompilatori obično imaju mogućnost da se eksplicitno izabere neka tehnika optimizacije ili nivo optimizovanja (što uključuje ili ne uključuje više pojedinačnih tehnika). Optimizovanje obično uvećava vreme kompiliranja, pa se često primenjuje samo u završnim fazama razvoja programa. Na primer, za kompilator `gcc` nivo optimizacije se bira korišćenjem opcije `-O` iza koje može biti navedena jedna od vrednosti:

0 za podrazumevani režim kompilacije, samo sa bazičnim tehnikama optimizacije.

1 kompilator pokušava da smanji i veličinu izvršnog programa i vreme izvršavanja, ali ne primenjuje tehnike optimizacije koje mogu da bitno uvećaju vreme kompiliranja.

2 kompilator primenjuje tehnike optimizacije koje ne zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).

3 kompilator primenjuje i tehnike optimizacije koje zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).

s kompilator primenjuje tehnike optimizovanja koje smanjuju izvršni program, a ne njegovo vreme izvršavanja.

Ne optimizovati nebitne delove programa. Ukoliko je merenjem potvrđeno da neki deo programa neznatno utiče na njegovu efikasnost – ne vredi unapređivati ga – bolje je da on ostane u jednostavnom i lako razumljivom obliku. Uvek se treba usredsrediti na delove programa koji troše najveći udeo vremena.[2]

Čuvene su reči Donalda Knuta: „Programeri troše enormne količine vremena razmišljajući ili brinući o brzini nekritičnih delova svojih programa, i to zapravo stvara jak negativan uticaj u fazama debagovanja i održavanja. Treba da zaboravimo na male efikasnosti, recimo 97% vremena: prerana optimizacija je koren svih zala. Ipak, ne treba da propustimo mogućnosti u preostalih kritičnih 3%."

Izdvojiti izračunavanja koja se ponavljaju. Identična skupa izračunavanja ne bi trebalo da se ponavljaju. U sledećem primeru:

```
x = x0*cos(0.01) - y0*sin(0.01);
y = x0*sin(0.01) + y0*cos(0.01);
```

vrednosti funkcije cos i sin se izračunavaju po dva puta za istu vrednost argumenta. Ove funkcije su vremenski veoma zahtevne i bolje je u pomoćnim promenljivama sačuvati njihove vrednosti pre korišćenja. Dakle, umesto navedenog, daleko je bolji naredni kôd:

```
cs = cos(0.01);
sn = sin(0.01);
x = x0*cs - y0*sn;
y = x0*sn + y0*cs;
```

Izdvajanje koda izvan petlje. Ova preporuka je u istom duhu kao prethodna. Iz petlje je potrebno izdvojiti izračunavanja koja su ista u svakom prolasku kroz petlju. Na primer, umesto:

```
for (i=0; i < strlen(s); i++)
  if ( s[i] == c )
    ...
```

daleko je bolje:

```
len = strlen(s);
for (i=0; i < len; i++)
  if ( s[i] == c )
    ...
```

U prvoj verziji koda, kroz petlju se prolazi n puta, gde je n dužina niske `s`. Međutim, u svakom prolasku kroz petlju se poziva funkcija `strlen` za argument `s` i u svakom tom pozivu se prolazi kroz celu nisku `s` (do završne nule). Zbog toga je složenost prve petlje (barem) $O(n^2)$. S druge strane, u drugoj verziji koda, funkcija `strlen` se poziva samo jednom i složenost tog poziva i petlje koja sledi zajedno može da bude $O(n)$.

Slično, umesto:

```
for (i=0; i < N; i++) {
  x = x0*cos(0.01) - y0*sin(0.01);
  y = x0*sin(0.01) + y0*cos(0.01);
  ...
```

[2] Ilustrujmo ovaj savet jednim Lojdovim (Samuel Loyd, 1841-1911) problemom: „Ribolovac je sakupio 1kg crva. Sakupljeni crvi imali su 1% suve materije i 99% vode. Sutra, nakon sušenja, crvi su imali 95% vode u sebi. Kolika je tada bila ukupna masa crva?" Na početku, suva materija činila je 1% od 1kg, tj. 10gr. Sutradan, vode je bilo 19 puta više od suve materije, tj. 190gr, pa je ukupna masa crva bila 200 gr.
Ako bi program činile funkcije f i g i ako bi f pokrivala 99% a funkcija g 1% vremena izvršavanja, glavni kandidat za optimizovanje bila bi naravno funkcija f. Ukoliko bi njen udeo u konačnom vremenu pao sa 99% na 95%, onda bi ukupno vreme izvršavanja programa bilo svedeno na 20% početnog.

```
  x0 = x;
  y0 = y;
}
```

daleko je bolji kôd:

```
cs = cos(0.01);
sn = sin(0.01);
for (i=0; i < N; i++) {
  x = x0*cs - y0*sn;
  y = x0*sn + y0*cs;
  ...
  x0 = x;
  y0 = y;
}
```

Zameniti skupe operacije jeftinim. Uslov $\sqrt{x_1x_1 + y_1y_1} > \sqrt{x_2x_2 + y_2y_2}$ je ekvivalentan uslovu $x_1x_1+y_1y_1 > x_2x_2+y_2y_2$, ali je u programu daleko bolje umesto uslova `sqrt(x_1*x_1+y_1*y_1) > sqrt(x_2*x_2+y` koristiti `x_1*x_1+y_1*y_1 > x_2*x_2+y_2*y_2` jer se njime izbegava pozivanje veoma skupe funkcije `sqrt`.

Slično, ukoliko je moguće dobro je izbegavati skupe trigonometrijske funkcije, umesto „većih“ tipova dobro je koristiti manje, poželjno celobrojne, itd.

Ne ostavljati za fazu izvršavanja izračunavanja koja se mogu obaviti ranije. Ukoliko se tokom izvršavanja programa više puta koriste vrednosti iz malog skupa, uštedu može da donese njihovo izračunavanje unapred i uključivanje rezultata u izvorni kôd programa. Na primer, ako se u nekom programu koriste vrednosti kvadratnog korena od 1 do 100, te vrednosti se mogu izračunati unapred i njima se može inicijalizovati konstantni niz. Ovaj pristup prihvatljiv je ako je kritični resurs vreme a ne prostor.

Napisati kritične delove koda na asembleru. Savremeni kompilatori generišu veoma kvalitetan kôd. Ukoliko se koriste i rasplożive optimizacije, kôd takvog kvaliteta može da napiše retko koji programer. Ipak, u nekim situacijama, za neke vremenski kritične delove programa, opcija je pisanje tih delova programa na asembleru.

3.8 Popravljanje prostorne složenosti

Za razliku od nekadašnjih računara, na savremenim računarima memorija obično nije kritični resurs. Optimizacije se obično usredsređuju na štednju vremena, a ne prostora. Ipak, postoje situacije u kojima je potrebno štedeti memoriju, na primer, onda kada program barata ogromnim količinama podataka, kada je sâm program veliki i zauzima značajan deo radne memorije ili kada se program izvršava na nekom specifičnom uređaju koji ima malo memorije. Često ušteda memorije zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva:

Koristiti najmanje moguće tipove. Za celobrojne podatke, umesto tipa `int` često je dovoljan tip `short` ili čak `char`. Za reprezentovanje realnih brojeva, ukoliko preciznost nije kritična, može se, umesto tipa `double` koristiti tip `float`. Za reprezentovanje logičkih vrednosti dovoljan je jedan bit a više takvih vrednosti može da se čuva u jednom bajtu (i da im se pristupa koristeći bitovske operatore).

Ne čuvati ono što može da se lako izračuna. U prethodnom delu je, u slučaju da je kritična brzina, a ne prostor, dobro da se vrednosti koje se često koriste u programu izračunaju unapred i uključe u izvorni kod programa. Ukoliko je kritična memorija, treba uraditi upravo suprotno i ne čuvati nikakve vrednosti koje se mogu izračunati u fazi izvršavanja.

Pitanja i zadaci za vežbu

Pitanje 3.1. *Svaka instrukcija na računaru se izvršava za* $1 \cdot 10^{-9} s$. *Algoritam* A_1 *zahteva* n^2 *instrukcija, a* A_2 *zahteva* n^3 *instrukcija.*

Sa obradom ulaza koje dimenzije algoritam A_1 *može da završi za jedan minut?*

Sa obradom ulaza koje dimenzije algoritam A_2 *može da završi za jedan minut?*

Pitanje 3.2. *Svaka instrukcija na računaru se izvršava za* $2ns$. *Algoritam* A_1 *za obradu ulaza dimenzije* n *zahteva* n^2 *instrukcija, a* A_2 *zahteva* n^3 *instrukcija.*

Sa obradom ulaza koje dimenzije algoritam A_1 *može da završi za jedan minut?*

Sa obradom ulaza koje dimenzije algoritam A_2 *može da završi za jedan minut?*

Pitanje 3.3. *Kada se kaže da važi* $f(n) \in O(g(n))$*?*

Pitanje 3.4. *Dokazati da važi:*

(a) $n^2 + 2n = O(n^2)$.

(b) $2^n + n = O(2^n)$.

(c) $2^n + n^3 = O(2^n)$;

(d) $3n + 2n^2 = O(n^2)$.

(e) $5^n + 2^n = O(5^n)$.

Pitanje 3.5. *Da li funkcija* $7n^2 + 3$ *pripada klasi:*
a) $O(3n^2 + 1)$ *b)* $O(n)$ *c)* $O(n^2)$ *d)* $O(n^3 \log(n))$ *e)* $O(2 \cdot \log(n))$

Pitanje 3.6. *Da li funkcija* $3n \log n + 5n$ *pripada klasi:*
$O(n)$ $O(n \log n)$ $O(n^2 \log n)$ $O(n \log^2 n)$ $O(\log n)$ $O(n + \log n)$ $O(15n)$

Pitanje 3.7. *Da li funkcija* $7n^2 + 3n \log(n)$ *pripada klasi:*
a) $O(3n^2 + 1)$ *b)* $O(n)$ *c)* $O(n^2)$ *d)* $O(n^3 \log(n))$ *e)* $O(2 \cdot \log(n))$

Pitanje 3.8. *Da li funkcija* $3n \log n + 5n + 100$ *pripada klasi:*
$O(n)$ $O(n \log n)$ $O(n^2 \log n)$ $O(n \log^2 n)$ $O(\log n)$ $O(n + \log n)$ $O(15n)$ $O(108)$

Pitanje 3.9. *Da li funkcija* $n^6 + 2^n + 10^{10}$ *pripada klasi:*
$O(n)$ $O(2^n)$ $O(n^6)$ $O(n^{10})$ $O(10^{10})$ $O(n^6 + 2^n)$ $O(2^n + 10^{10})$ $O(2^n 10^{10})$

Pitanje 3.10. *Ako* $a(n)$ *pripada klasi* $O(n \log n)$, *a* $b(n^2)$ *pripada klasi* $O(n^2)$, *onda* $a(n) + b(n)$ *pripada klasi (izabrati sve ispravne odgovore):*
(a) $O(n \log n)$;
(b) $O(n^2)$;
(c) $O(n \log n + n^2)$;
(d) $O(n^2 \log n)$.

Pitanje 3.11. *Ako* $a(n)$ *pripada klasi* $O(n^2)$, *a* $b(n)$ *pripada klasi* $O(n^3)$, *onda* $a(n) + b(n)$ *pripada klasi (izabrati sve ispravne odgovore):*
(a) $O(n^2)$;
(b) $O(n^3)$;
(c) $O(n^5)$;
(d) $O(n^6)$.

Pitanje 3.12. *Kada kažemo da je složenost algoritma* A *jednaka* $O(f)$*?*

Pitanje 3.13. *Ako je složenost algoritma* A *za ulaznu vrednost* n $O(n^3)$, *a složenost algoritma* B *za ulaznu vrednost* n $O(n^4)$, *kolika je složenost algoritma* C *koji se izvršava tako što se izvršava prvo algoritam* A, *pa algoritam* B*:*
(a) $O(n^3)$; *(b)* $O(n^4)$; *(c)* $O(n^7)$; *(d)* $O(n^{12})$.

Pitanje 3.14. *Odrediti* n*-ti član niza* $T(n) = x \cdot T(n-1)$, $T(0) = y$.

Pitanje 3.15. *Kojoj klasi pripada rešenje rekurentne relacije* $T(n) = 2T(n/2) + n$*?*

Pitanje 3.16. *Ako nizovi* T_1 *i* T_2 *zadovoljavaju rekurentnu jednačinu* $T(n) = aT(n-1) + bT(n-2)$*, dokazati da ovu jednačinu zadovoljava i niz* $T_1 + T_2$*.*

Pitanje 3.17. *Ako važi* $T(n) = 4T(n-1) - 4T(n-2)$*, koju formu ima opšti član niza* T*?*

Pitanje 3.18. *Naći opšte rešenje jednačine* $T(n) = 3T(n-1) - 2T(n-2)$*.*

Pitanje 3.19. *Ako je* $T(1) = 1$*,* $T(2) = 7$ *i* $T(n+2) = -T(n+1) + 2T(n)$*, koliko je* $T(20)$*?*

Pitanje 3.20. *Odrediti n-ti član niza* $T(n)$ *za koji važi* $T(1) = 0$*,* $T(2) = 5$*,* $T(n+2) = 5T(n+1) - 6T(n)$*.*

Pitanje 3.21. *Odrediti n-ti član niza:*
(a) $T(n) = 6T(n-1) - 8T(n-2)$*,* $T(0) = 1$
(b) $T(n) = 8T(n-1) - 15T(n-2)$*,* $T(0) = 1$
(c) $T(n) = 6T(n-1) - 9T(n-2)$*,* $T(0) = 1$
(d) $T(n) = 10T(n-1) - 25T(n-2)$*,* $T(0) = 1$*,* $T(1) = 1$*,*
Kojoj klasi složenosti pripada $T(n)$*?*

Pitanje 3.22. *Rešiti rekurentnu relaciju* $T(n) = T(n-1) + 3$*,* $T(0) = 1$*?*

Pitanje 3.23. *Ako je* $T(n+1) = 2 \cdot T(n) + 1$*, kojoj klasi pripada* T*?*

Pitanje 3.24. *Kojoj klasi složenosti pripada* $T(n)$ *ako važi:*
(a) $T(n) = 5T(n-1) - 4T(n-2)$
(b) $T(n) = 5T(n-1) - 6T(n-2)$
(c) $T(n) = 8T(n-1) - 15T(n-2)$
(d) $T(n) = 4T(n-1) - 4T(n-2)$
(e) $T(n) = 6T(n-1) - 9T(n-2)$
(f) $T(n) = 8T(n-1) - 16T(n-2)$*.*

Pitanje 3.25. *Odrediti opšti član niza* $T(n)$ *za koji važi* $T(1) = 0$*,* $T(2) = 5$*,* $T(3) = 14$*,* $T(n+3) = 3T(n+2) - 3T(n+1) + T(n)$*.*

Pitanje 3.26. *Ako je* $T(n) = aT(n/b) + cn^k$ *i* $a = b^k$*, kog je reda* $T(n)$*?*

Pitanje 3.27. *Šta je rešenje rekurentne relacije* $T(n) = aT(n/b) + cn^k$*, gde su* a *i* b *celobrojne konstante* $(a \geq 1, b \geq 1)$ *i* c *i* k *pozitivne konstante?*

Pitanje 3.28. *Kojoj klasi složenosti pripada* $T(n)$ *ako važi:*
(a) $T(n) = T(n-1) + x$*,* $T(0) = y$*?*
(b) $T(n) = 2T(n/2) + O(n)$
(c) $T(n) = 2T(n/2) + cn$
(d) $T(n) = 8T(n/2) + cn^2$
(e) $T(n) = 4T(n/2) + cn^2$

Pitanje 3.29. *Kolika je složenost algoritma za pronalaženje minimuma niza? Kolika je složenost algoritma za pronalaženje drugog po veličini elementa niza?*

Pitanje 3.30. *Ako za vreme izvršavanja* $T(n)$ *algoritma A (gde n određuje ulaznu vrednost za algoritam) važi* $T(n+1) = T(n) + 3$ *(za* $n \geq 1$*) i* $T(1) = 2$*, odrediti složenost algoritma A.*

Pitanje 3.31. *Ako za vreme izvršavanja* $T(n)$ *algoritma A (gde n određuje ulaznu vrednost za algoritam) važi* $T(n+2) = 4T(n+1) - 3T(n)$ *(za* $n \geq 1$*) i* $T(1) = 1, T(2) = 2$*, onda je složenost algoritma A:*
(a) $O(n)$*; (b)* $O(3^n)$*; (c)* $O(4^n)$*; (d)* $O(n^4)$*;*

Pitanje 3.32. *Ako za vreme izvršavanja* $T(n)$ *algoritma A (gde n određuje ulaznu vrednost za algoritam) važi* $T(n+2) = 3T(N+1) + 2T(n)$ *(za* $n \geq 1$*) i* $T(1) = 1$*,* $T(2) = 2$*, odrediti složenost algoritma A.*

Pitanje 3.33. *Algoritam A za ulaznu vrednost n poziva sebe samog za ulaznu vrednost* $n-1$ *i koristi još n dodatnih operacija. Izračunati složenost algoritma A.*

Pitanje 3.34. *Koju vrednost za dato n izračunava naredna funkcija i koja je njena memorijska složenost?*

```
int f(int n) {
  return n == 0 ? 1 : n*f(n-1);
}
```

Pitanje 3.35. *Odrediti složenost izvršavanja sledeće funkcije:*

```
void f(int n)
{
    if (n<1)
        printf("*");
    else
    {
        f(n-1);
        printf("----\n");
        f(n-1);
    }
}
```

Pitanje 3.36. *Odrediti složenost sledeće funkcije:*

```
void f(int n)
{
    if (n<2)
        printf("* ");
    else
    {
        f(n-2);
        f(n-1);
        f(n-2);
    }
}
```

Pitanje 3.37. *Algoritam A izvršava se za vrednost n ($n > 1$) pozivanjem algoritma B za vrednost $n - 1$, pri čemu se za svodjenje problema troši jedna vremenska jedinica. Algoritam B izvršava se za vrednost n ($n > 1$) pozivanjem algoritma A za vrednost $n - 1$, pri čemu se za svodjenje problema troše dve vremenske jedinice. Za ulaznu vrednost $n = 1$, algoritam A troši jednu, a algoritam B dve vremenske jedinice. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.*

Pitanje 3.38. *Za koji problem kažemo da pripada klasi P? Za koji problem kažemo da pripada klasi NP? Za koji problem kažemo da je NP-kompletan?*

Pitanje 3.39. *Kako se dokazuje da neki problem pripada klasi NP? Kakvu bi posledicu imao dokaz da neki NP-kompletan problem pripada klasi P a kakvu dokaz da neki NP-kompletan problem ne pripada klasi P?*

Pitanje 3.40. *Ako je svaki NP problem svodljiv u polinomijalnom vremenu na problem A, kakav je problem A?*

Pitanje 3.41. *Ako je neki NP kompletan problem svodljiv u polinomijalnom vremenu na problem A, šta onda važi za problem A?*

Pitanje 3.42. *Ako neki NP-težak problem A nije NP-kompletan, šta onda to znači?*

Pitanje 3.43. *Ako se zna da je algoritam A NP-kompletan i da algoritam B pripada klasi NP, kako se može dokazati da je algoritam B NP-kompletan?*

Pitanje 3.44. *Dokazano je da važi (izabrati sve ispravne odgovore):*
(a) $P \subset NP$; (b) $NP \subset P$; (c) $P = NP$; (d) $P \neq NP$;

Pitanje 3.45. *Navesti primer problema koji pripada klasi P. Navesti primer problema koji pripada klasi NP. Navesti primer problema koji je NP-kompletan.*

Pitanje 3.46. *SAT problem je:*
(a) $P = NP$; (b) $P \neq= NP$; (c) problem iskazne zadovoljivosti; (d) problem ispitivanja složenosti programa.

Pitanje 3.47. *Šta je SAT? Da li je SAT NP-težak problem? Da li je SAT NP-kompletan problem?*

Pitanje 3.48. *Da li SAT pripada klasi P? Da li SAT pripada klasi NP? Šta je posledica tvrđenja $SAT \in P$, a šta tvrđenja $SAT \notin P$?*

Zadatak 3.8.1. *Ako za vreme izvršavanja $T(n)$ algoritma A (gde n odredjuje ulaznu vrednost za algoritam) važi $T(n) = T(n-1) + n/2$ i $T(1) = 1$, odrediti složenost algoritma A.*

Rešenje:

$$T(n) = T(n-1) + \frac{n}{2} = T(n-2) + \frac{n-1}{2} + \frac{n}{2} = T(1) + \frac{2}{2} + \ldots + \frac{n-1}{2} + \frac{n}{2} =$$

$$= T(1) + \frac{1}{2}(2 + \ldots + n) = 1 + \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) = \frac{n^2 + n + 2}{4}\,,$$

pa je algoritam A kvadratne složenosti.

Glava 4

Ispravnost programa

Jedno od centralnih pitanja u razvoju programa je pitanje njegove ispravnosti (korektnosti). Softver je u današnjem svetu prisutan na svakom koraku: softver kontroliše mnogo toga — od bankovnih računa i komponenti televizora i automobila, do nuklearnih elektrana, aviona i svemirskih letelica. U svom tom softveru neminovno su prisutne i greške. Greška u funkcionisanju daljinskog upravljača za televizor može biti tek uznemirujuća, ali greška u funkcionisanju nuklearne elektrane može imati razorne posledice. Najopasnije greške su one koje mogu da dovedu do velikih troškova, ili još gore, do gubitka ljudskih života. Neke od katastrofe koje su opštepoznate su eksplozija rakete *Ariane (fr. Ariane 5)* 1996. uzrokovana konverzijom broja iz šezdesetčetvorobitnog realnog u šesnaestobitni celobrojni zapis koja je dovela do prekoračenja, zatim greška u numeričkom koprocesoru procesora Pentium 1994. uzrokovana pogrešnim indeksima u `for` petlji u okviru softvera koji je radio dizajn čipa, kao i pad orbitera poslatog na Mars 1999. uzrokovan činjenicom da je deo softvera koristio metričke, a deo softvera engleske jedinice. Međutim, fatalne softverske greške i dalje se neprestano javljaju i one koštaju svetsku ekonomiju milijarde dolara. Evo nekih od najzanimljivijih:

- Ne naročito opasan, ali veoma zanimljiv primer greške je greška u programu *Microsoft Excel 2007* koji, zbog greške u algoritmu formatiranja brojeva pre prikazivanja, rezultat izračunavanja izraza $77.1 * 850$ prikazuje kao $100,000$ (iako je interno korektno sačuvan).

- 14. Septembra 2004. godine, više od četiristo aviona u blizini aerodroma u Los Anđelesu je istovremeno izgubilo vezu sa kontrolom leta. Na sreću, zahvaljujući rezervnoj opremi unutar samih aviona, do nesreće ipak nije došlo. Uzrok gubitka veze bila je greška prekoračenja u brojaču milisekundi u okviru sistema za komunikaciju sa avionima. Da ironija bude veća, ova greška je bila ranije otkrivena, ali pošto je do otkrića došlo kada je već sistem bio isporučen i instaliran na nekoliko aerodroma, njegova jednostavna popravka i zamena nije bila moguća. Umesto toga, preporučeno je da se sistem resetuje svakih 30 dana kako do prekoračenja ne bi došlo. Procedura nije ispoštovana i greška se javila posle tačno 2^{32} milisekundi, odnosno 49.7 dana od uključivanja sistema.

- Pad satelita *Kriosat (eng. Cryosat)* 2005. koštao je Evropsku Uniju oko 135 miliona evra. Pad je uzrokovan greškom u softveru zbog koje nije na vreme došlo do razdvajanja satelita i rakete koja ga je nosila.

- Više od pet procenata penzionera i primalaca socijalne pomoći u Nemačkoj je privremeno ostalo bez svog novca kada je 2005. godine uveden novi računarski sistem. Greška je nastala zbog toga što je sistem, koji je zahtevao desetocifreni zapis svih brojeva računa, kod starijih računa koji su imali osam ili devet cifara brojeve dopunjavao nulama, ali sa desne umesto sa leve strane kako je trebalo.

- Kompanije Dell i Apple su tokom 2006. godine morali da korisnicima zamene više od pet miliona laptop računara zbog greške u dizajnu baterije kompanije Sony koja je uzrokovala da se nekoliko računara zapali.

4.1 Osnovni pristupi ispitivanju ispravnosti programa

Postupak pokazivanja da je program ispravan naziva se *verifikovanje programa*. U razvijanju tehnika verifikacije programa, potrebno je najpre precizno formulisati pojam ispravnosti programa. Ispravnost programa počiva na pojmu *specifikacije*. Specifikacija je, neformalno, opis željenog ponašanja programa koji treba napisati. Specifikacija se obično zadaje u terminima *preduslova* tj. uslova koje ulazni parametri programa moraju da zadovolje, kao i *postuslova* tj. uslova koje rezultati izračunavanja moraju da zadovolje. Kada je poznata specifikacija, potrebno je verifikovati program, tj. dokazati da on zadovoljava specifikaciju. Dva osnovna pristupa verifikaciji su:

dinamička verifikacija koja podrazumeva proveru ispravnost u fazi izvršavanja programa, najčešće putem testiranja;

statička verifikacija koja podrazumeva analizu izvornog kôda programa, često korišćenjem formalnih metoda i matematičkog aparata.

U okviru verifikacije programa, veoma važno pitanje je pitanje zaustavljanja programa. *Parcijalna korektnost* podrazumeva da neki program, ukoliko se zaustavi, daje korektan rezultat (tj. rezultat koji zadovoljava specifikaciju). *Totalna korektnost* podrazumeva da se program za sve (specifikacijom dopuštene) ulaze zaustavlja, kao i da su dobijeni rezultati parcijalno korektni.

4.2 Dinamičko verifikovanje programa

Dinamičko verifikovanje programa podrazumeva proveravanje ispravnosti u fazi izvršavanja programa. Najčešći vid dinamičkog verifikovanja programa je testiranje.

4.2.1 Testiranje

Najznačajnija vrsta dinamičkog ispitivanja ispravnosti programa je testiranje. Testiranje može da obezbedi visok stepen pouzdanosti programa. Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrđenje „program ima prosečno vreme izvršavanja 0.5 sekundi“ je (u principu) proverivo testovima, pa čak i tvrđenje „prosečno vreme između dva pada programa je najmanje 8 sati sa verovatnocom 95%“. Međutim, tvrđenje „prosečno vreme izvršavanja programa je dobro“ suviše je neodređeno da bi moglo da bude testirano. Primetimo da je, na primer, tvrđenje „prosečno vreme između dva pada programa je najmanje 8 godina sa verovatnoćom 95%“ u principu proverivo testovima ali nije praktično izvodivo.

U idealnom slučaju, treba sprovesti iscrpno testiranje rada programa za sve moguće ulazne vrednosti i proveriti da li izlazne vrednosti zadovoljavaju specifikaciju. Međutim, ovakav iscrpan pristup testiranju skoro nikada nije praktično primenljiv. Na primer, iscrpno testiranje korektnosti programa koji sabira dva 32-bitna broja, zahtevalo bi ukupno $2^{32} \cdot 2^{32} = 2^{64}$ različitih testova. Pod pretpostavkom da svaki test traje jednu nanosekundu, iscrpno testiranje bi zahtevalo približno $1.8 \cdot 10^{10}$ sekundi što je oko 570 godina. Dakle, testiranjem nije praktično moguće dokazati ispravnost netrivijalnih programa. S druge strane, testiranjem je moguće dokazati da program nije ispravan tj. pronaći greške u programima.

S obzirom na to da iscrpno testiranje nije praktično primenljivo, obično se koristi tehnika testiranja tipičnih ulaza programa kao i specijalnih, karakterističnih ulaznih vrednosti za koje postoji veća verovatnoća da dovedu do neke greške. U slučaju pomenutog programa za sabiranje, tipični slučaj bi se odnosio na testiranje korektnosti sabiranja nekoliko slučajno odabranih parova brojeva, dok bi za specijalne slučajeve mogli biti proglašeni slučajevi kada je neki od sabiraka 0, 1, -1, najmanji negativan broj, najveći pozitivan broj i slično.

Postoje različite metode testiranja, a neke od njih su:

Testiranje zasebnih jedinica (eng. unit testing) U ovom metodu testiranja, nezavisno se testovima proverava ispravnost zasebnih jedinica koda. „Jedinica“ je obično najmanji deo programa koji se može testirati. U proceduralnom jeziku kao što je C, „jedinica“ je obično jedna funkcija. Svaki *jediničnI test* treba da bude nezavisan od ostalih, ali puno jediničnih testova može da bude grupisano u baterije testova, u jednoj ili više funkcija sa ovom namenom. Jedinični testovi treba da proveravaju ponašanje funkcije, za tipične, granične i specijalne slučajeve. Ovaj metod je veoma

važan u obezbeđivanju veće pouzdanosti kada se mnoge funkcije u programu često menjaju i zavise jedna od drugih. Kad god se promeni željeno ponašanje neke funkcije, potrebno je ažurirati odgovarajuće jedinične testove. Ovaj metod veoma je koristan i zbog toga što često otkriva trivijalne greške, a i zbog toga što jedinični testovi predstavljaju svojevrsnu specifikaciju itd.

Postoje specijalizovani softverski alati i biblioteke koji omogućavaju jednostavno kreiranje i održavanje ovakvih testova. *Jedinične testove* obično pišu i koriste, u toku razvoja softvera, sami autori programa ili testeri koji imaju pristup kodu.

Regresiono testiranje (eng. regression testing) U ovom pristupu, proveravaju se izmene programa kako bi se utvrdilo da se nova verzija ponaša isto kao stara (na primer, generiše se isti izlaz). Za svaki deo programa implementiraju se testovi koji proveravaju njegovo ponašanje. Pre nego što se napravi nova verzija programa, ona mora da uspešno prođe sve stare testove kako bi se osiguralo da ono što je ranije radilo radi i dalje, tj. da nije narušena ranija funkcionalnost programa.

Regresiono testiranje primenjuje se u okviru samog implementiranja softvera i obično ga sprovode testeri.

Integraciono testiranje (engl. integration testing) Ovaj vid testiranja primenjuje se kada se više programskih modula objedinjuje u jednu celinu i kada je potrebno proveriti kako funkciniše ta celina i komunikacija između njenih modula. Integraciono testiranje obično se sprovodi nakon što su pojedinačni moduli prošli kroz druge vidove testiranja. Kada nova programska celina, sastavljena od više modula uspešno prođe kroz integraciono testiranje, onda ona može da bude jedna od komponenti celine na višem nivou koja takođe treba da prođe integraciono testiranje.

Testiranje valjanosti (engl. validation testing) Testiranje valjanosti treba da utvrdi da sistem ispunjava zadate zahteve i izvršava funkcije za koje je namenjen. Testiranje valjanosti vrši se na kraju razvojnog procesa, nakon što su uspešno završene druge procedure testiranja i utvrđivanja ispravnosti. Testovi valjanosti koji se sprovode su testovi visokog nivoa koji treba da pokažu da se u obradama koriste odgovarajući podaci i u skladu sa odgovarajućim procedurama, opisanim u specifikaciji programa.

4.2.2 Debagovanje

Pojednostavljeno rečeno, testiranje je proces proveravanja ispravnosti programa, sistematičan pokušaj da se u programu (za koji se pretpostavlja da je ispravan) pronađe greška. S druge strane, debagovanje se primenjuje kada se zna da program ima grešku. Debager je alat za praćenje izvršavanja programa radi otkrivanja konkretne greške (baga, eng. bug). To je program napravljen da olakša detektovanje, lociranje i ispravljanje grešaka u drugom programu. On omogućava programeru da ide korak po korak kroz izvršavanje programa, prati vrednosti promenljivih, stanje programskog steka, sadržaj memorije i druge elemente programa.

Slika 4.1 ilustruje rad debagera `kdbg`. Uvidom u prikazane podatke, programer može da uoči traženu grešku u programu. Da bi se program debagovao, potrebno je da bude preveden za *debag* režim izvršavanja. Za to se, u kompilatoru `gcc` koristi opcija `-g`. Ako je izvršni program `mojprogram` dobijen na taj način, može se debagovati navođenjem naredbe:

```
kdbg mojprogram
```

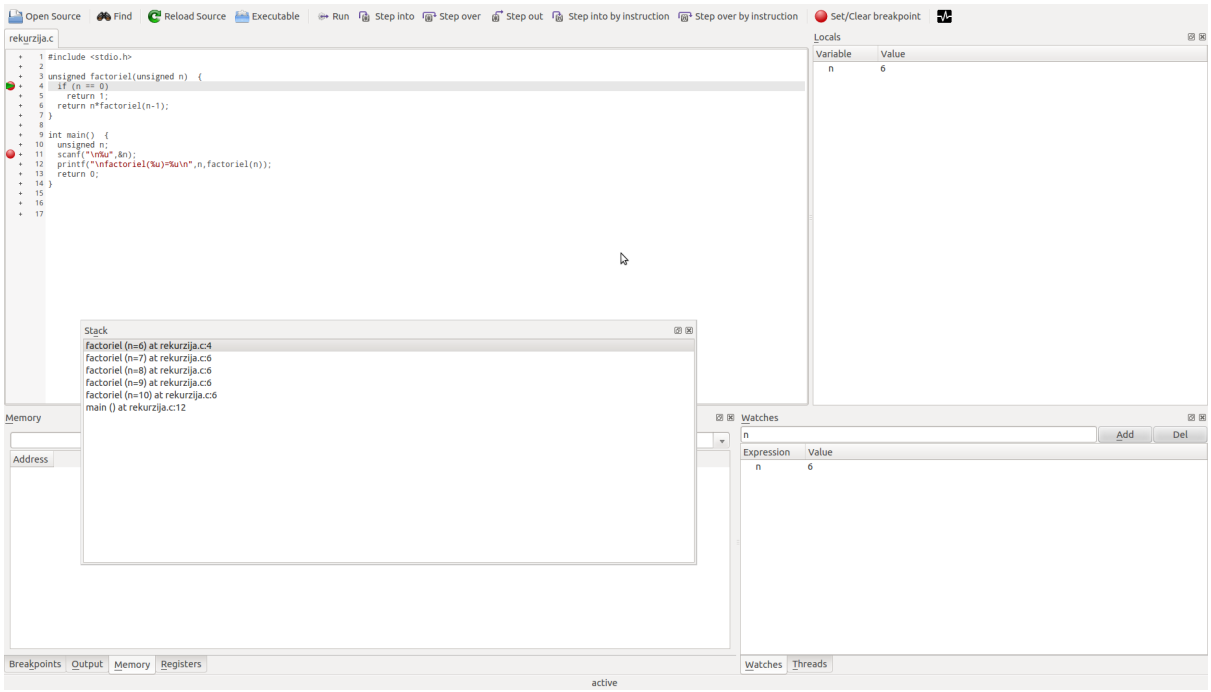
4.2.3 Otkrivanje curenja memorije

Curenje memorije je problem koji je često teško primetiti (sve dok ima memorije na raspolaganju) i locirati u izvornom kôdu. Postoji više programa koji mogu pomoći u proveravanju da li u programu postoji curenje memorije i u lociranju mesta u programu koje je odgovorno za to. Jedan od takvih programa je `valgrind` (videti i poglavlje 3.1.3) koji ima alat `memcheck` sa ovom svrhom.

Razmotrimo sledeći jednostavan program. Očigledno je da prva dinamička alokacija dovodi do curenja memorije.

Program 4.1.

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



Slika 4.1: Ilustracija rada debagera `kdbg`

```
#include<stdlib.h>

int main()  {
  int* p;
  p = (int*)malloc(1000);
  if (p == NULL)
    return -1;
  p = (int*)malloc(1000);
  if (p == NULL)
    return -1;
  else
    free(p);
  return 0;
}
```

Ukoliko je navedeni program kompiliran u *debag* modu i ukoliko je izvršna verzija nazvana `mojprogram`, alat `valgrind` može se, za detektovanje curenja memorije, pozvati na sledeći način:

```
valgrind --tool=memcheck --leak-check=yes ./mojprogram
```

Curenje memorije će biti uspešno otkriveno i `valgrind` daje sledeći izlaz:

```
==9697== Memcheck, a memory error detector
==9697== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==9697== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==9697== Command: ./mojprogram
==9697==
==9697==
==9697== HEAP SUMMARY:
==9697==     in use at exit: 1,000 bytes in 1 blocks
==9697==   total heap usage: 2 allocs, 1 frees, 2,000 bytes allocated
==9697==
==9697== 1,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9697==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==9697==    by 0x8048428: main (curenje_memorije.c:6)
==9697==
==9697== LEAK SUMMARY:
==9697==    definitely lost: 1,000 bytes in 1 blocks
==9697==    indirectly lost: 0 bytes in 0 blocks
```

```
==9697==      possibly lost: 0 bytes in 0 blocks
==9697==    still reachable: 0 bytes in 0 blocks
==9697==         suppressed: 0 bytes in 0 blocks
==9697==
==9697== For counts of detected and suppressed errors, rerun with: -v
==9697== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Alati za otkrivanje curenja memorije su programerima jako korisni, ali oni ipak nisu svemoćni u otkrivanju curenja memorije. Naime, u procesu traženja curenja memorije prati se samo jedno konkretno izvršavanje programa i na neke naredbe koje su odgovorne za curenje memorije u tom konkretnom izvršavanju možda se uopste neće naići.

4.3 Statičko ispitivanje ispravnosti programa

4.3.1 Proveravanje kritičnih mesta u programima

Proveravanje graničnih vrednosti. Najveći broj bagova javlja se na granicama opsega petlje, graničnim indeksima niza, graničnim vrednostima argumenata aritmetičkih operacija i slično. Zbog toga je testiranje na graničnim vrednostima izuzetno važno i program koji prolazi takve ulazne veličine često je ispravan i za druge. U nastavku će biti razmotren primer iz knjige Kernigana i Pajka – kôd koji čita i upisuje u niz karaktere sa standardnog ulaza sve dok ne dođe do kraja reda ili dok ne popuni niz:

```
int i;
char s[MAX];
for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i);
s[--i] = '\0';
```

Jedna od prvih stvari koje treba proveriti je da li kôd radi u graničnom slučaju, i to najjednostavnijem – kada je na ulazu prazan red (tj. red koji sadrži samo karakter `'\n'`. Petlja se zaustavlja u početnoj iteraciji i vrednost `i` neće biti inkrementirana, tj. ostaće jednaka `0`, te će u poslednjoj naredbi biti promenjen element `s[-1]`, van granica niza, što je greška.

Ukoliko se navedeni kôd napiše korišćenjem idiomske forme `for` petlje, on postaje:

```
for (i=0; i < MAX-1; i++)
  if ((s[i] = getchar()) == '\n')
    break;
s[i] = '\0';
```

U ovom slučaju, lako se proverava da kôd ispravno radi za početni test. Drugom granicom mogu se smatrati ulazni redovi koji su veoma dugi ili pre čijeg kraja se nalazi kraj toka podataka. Navedeni kôd radi neispravno u ovom drugom slučaju, te je potrebna nova verzija:

```
for (i=0; i < MAX-1; i++)
  if ((s[i] = getchar()) == '\n' || s[i]==EOF)
    break;
s[i] = '\0';
```

Naredni testovi koje treba napraviti za navedeni kôd odnose se na redove koji su dugi koliko i niz `s` ili kraći za jedan karakter, duži za jedan karakter i slično.

Proveravanje pre-uslova. Da bi neko izračunavanje imalo smisla često je potrebno da važe neki pre-uslovi. Na primer, ako je u kôdu potrebno izračunati prosečan broj poena `n` studenata, pitanje je šta raditi ukoliko je `n` jednako `0` i ponašanje programa treba da bude testirano u takvim situacijama. Jedno moguće ponašanje programa je, na primer, da se ako je `n` jednako `0`, vrati `0` kao rezultat, a drugo da se ne dozvoli pozivanje modula za izračunavanje proseka ako je `n` jednako `0`. U ovom drugom slučaju, pre kôda za izračunavanje proseka može da se navede naredba `assert(n > 0);` koja će da doprinese daljem testiranju. Generalno, program može da obrađuje i situacije koje

logički ne bi smele da se dogode. Time se delovi programa štite od neispravnih ulaza, a ovaj pristup programiranju se naziva *odbrambeno* programiranje.

Proveravanje povratnih vrednosti. Čest izvor grešaka tokom izvršavanja programa je neproveravanje povratnih vrednosti funkcija kao što su funkcije za alokaciju memorije, za rad sa datotekama itd. Povratne vrednosti ovih funkcija ukazuju na potencijalni problem i ukoliko se ignorišu – problem će samo postati veći. Opšti savet je da se uvek proverava povratna vrednost ovakvih funkcija.

4.3.2 Formalno ispitivanje ispravnosti programa

Proces verifikacije može biti neformalan i formalan (kada se koristi precizno definisana semantika programskog jezika i kada je precizno definisan logički okvir u kome se dokazi korektnosti izvode). U praksi se formalni dokaz ispravnosti programa retko izvodi i to često dovodi do neispravnog softvera i mnogih problema.

Veliki izazov za verifikaciju predstavlja činjenica da se semantika uobičajenih tipova podataka i operacija u programima razlikuje od uobičajene semantike matematičkih operacija nad celim i realnim brojevima (iako velike sličnosti postoje). Na primer, iako tip `int` podseća na skup celih brojeva, a operacija sabiranja dva podatka tipa `int` na sabiranje dva cela broja, razlike su evidentne — domen tipa `int` je konačan, a operacija se vrši „po modulu“ tj. u nekim slučajevima dolazi do prekoračenja. Mnoga pravila koja važe za cele brojeve ne važe za podatke tipa `int`. Na primer, $x \geq 0 \wedge y \geq 0 \Rightarrow x + y \geq 0$ važi ako su x i y celi brojevi, ali ne važi ako su podaci tipa `int`. U nekim slučajevima, prilikom verifikacije ovakve razlike se apstrahuju i zanemaruju. Time se, naravno, gubi potpuno precizna karakterizacija ponašanja programa i „dokazi“ korektnosti prestaju da budu dokazi korektnosti u opštem slučaju. Ipak, ovim se značajno olakšava proces verifikacije i u većini slučajeva ovakve aproksimacije ipak mogu značajno da podignu stepen pouzdanosti programa. U nastavku teksta, ovakve aproksimacije će biti stalno vršene.

Funkcionalni programi

U principu, najjednostavnije je dokazivanje korektnosti rekurzivno definisanih funkcija koje ne koriste elemente imperativnog programiranja (dodele vrednosti pomoćnim promenljivama, petlje i slično). Glavni razlog za ovo je činjenica da se ovakve funkcije mogu jednostavno modelovati matematičkim funkcijama koje za iste argumente uvek daju iste vrednosti. Zbog pretpostavke da nema eksplicitne dodele vrednosti promenljivama, funkcije za iste ulazne argumente uvek vraćaju istu vrednost, bez obzira na kontekst poziva i nema potrebe za razmatranjem tekućeg stanja programa (okarakterisanog tekućim vrednostima promenljivih). Dokazivanje korektnosti ovakvih programa teče nekim oblikom matematičke indukcije.

Primer 4.1. *Razmotrimo primer funkcije koja vrši množenje svođenjem na sabiranje. Dokažimo korektnost ove funkcije pod pretpostavkom da je $x \geq 0$ i da nema prekoračenja tokom izračunavanja.*

```
int mnozi(int x, int y) {
   if (x == 0)
        return 0;
   else
        return mnozi(x - 1, y) + y;

}
```

Indukcijom pokazujemo da važi $mnozi(x, y) = x \cdot y$.

Baza indukcije: *U slučaju da je $x = 0$, važi da je*

$$mnozi(x, y) = mnozi(0, y) = 0 = 0 \cdot y\ .$$

Induktivni korak: *Pretpostavimo da je x sledbenik nekog broja, tj. da je $x > 0$ i da tvrđenje važi za broj $x - 1$, tj. $mnozi(x - 1, y) = (x - 1) \cdot y$. Tada važi*

$$mnozi(x, y) = mnozi(x - 1, y) + y = (x - 1) \cdot y + y = x \cdot y\ .$$

Primer 4.2. *Jednostavno se može dokazati da naredna funkcija zaista izračunava faktorijel svog argumenta (pod pretpostavkom da nema prekoračenja):*

```
unsigned faktorijel(unsigned n) {
    if (n == 0)
        return 1;
    else
        return n*faktorijel(n-1);
}
```

Indukcijom pokazujemo da važi $faktorijel(n) = n!$.

Baza indukcije: *U slučaju da je* $n = 0$*, važi da je*

$$faktorijel(n) = faktorijel(0) = 1 = 0! \, .$$

Induktivni korak: *Pretpostavimo da je* n *sledbenik nekog broja, tj. da je* $n > 0$ *i da tvrđenje važi za broj* $n - 1$*, tj.* $faktorijel(n-1) = (n-1)!$*. Tada važi*

$$faktorijel(n) = n \cdot faktorijel(n-1) = n \cdot (n-1)! = n!.$$

Verifikacija imperativnih programa i invarijante petlji

U slučaju imperativnih programa (programa koji sadrže naredbu dodele i petlje), aparat koji se koristi za dokazivanje korektnosti mora biti znatno složeniji. Semantiku imperativnih konstrukata znatno je teže opisati u odnosu na (jednostavnu jednakosnu) semantiku čisto funkcionalnih programa. Sve vreme dokazivanja mora se imati u vidu tekući kontekst tj. stanje programa koje obuhvata tekuće vrednosti svih promenljivih koje se javljaju u programu. Program implicitno predstavlja relaciju prelaska izmedju stanja i dokazivanje korektnosti zahteva dokazivanje da će program na kraju stići u neko stanje u kome su zadovoljeni uslovi zadati specifikacijom. Dodatnu otežavajuću okolnost čine bočni efekti i činjenica da pozivi funkcija mogu da vrate različite vrednosti za iste prosleđene ulazne parametre, u zavisnosti od globalnog konteksta u kojima se poziv izvršio, te je dokaz korektnosti složenog programa teže razložiti na elementarne dokaze korektnosti pojedinih funkcija.

Kao najkompleksniji programski konstrukt, petlje predstavljaju jedan od najvećih izazova u verifikaciji. Umesto pojedinačnog razmatranja svakog stanja kroz koje se prolazi prilikom izvršavanja petlje, obično se formulišu uslovi (*invarijante petlji*) koji precizno karakterišu taj skup stanja. *Invarijanta petlje* je logička formula koja uključuje vrednosti promenljivih koje se javljaju u nekoj petlji i koja važi pri svakom ispitivanju uslova petlje (tj. neposredno pre, za vreme i neposredno nakon izvršavanja petlje).

Da bi se pokazalo da je neka formula invarijanta petlje, dovoljno je pokazati da (i) tvrđenje važi pre prvog ulaska u petlju i (ii) da tvrđenje ostaje na snazi nakon svakog izvršavanja tela petlje. Iz ova dva uslova, induktivnim rezonovanjem po broju izvršavanja tela petlje, moguće je pokazati da će tada tvrđenje važiti i nakon izvršavanja petlje. Slučaj prvog ulaska u petlju odgovara bazi indukcije, dok slučaj izvršavanja tela petlje odgovara induktivnom koraku.

Svaka petlja ima više invarijanti, pri čemu su neki uslovi „preslabi“ a neki „prejaki“ tj. ne objašnjavaju ponašanje programa. Na primer, bilo koja valjana formula (npr. $x \cdot 0 = 0$ ili $(x \geq y) \vee (y \geq x)$) je uvek invarijanta petlje. Međutim, da bi se na osnovu invarijante petlje moglo rezonovati o korektnosti programa, potrebno je da invarijanta bude takva da se jednostavno može pokazati iz svojstava programa pre ulaska u petlju, kao i da obezbeđuje željena svojstva programa nakon izlaska iz petlje.

Primer 4.3. *Razmotrimo program koji vrši množenje dva broja svođenjem na sabiranje.*

```
z = 0; n = 0;
while(n < x) {
  z = z + y;
  n = n + 1;
}
```

Nakon n izvršavanja tela petlje promenljiva z je n puta uvećana za vrednost y, pri čemu promenljiva n sadrži upravo broj izvršavanja tela petlje. Dakle, važi da je $z = n \cdot y$. Ako je x nenegativan ceo broj (što je u ovom slučaju bitno za korektnost algoritma), važi da je sve vreme $n \leq x$. Dokažimo da je formula

$$(n \leq x) \wedge (z = n \cdot y) \ .$$

zaista invarijanta petlje.

1. *Pokažimo da invarijanta važi pre ulaska u petlju. Pre ulaska u petlju je $x \geq 0$ (na osnovu preduslova), $z = 0$ i $n = 0$ te invarijanta, trivijalno, važi.*

2. *Pokažimo da invarijanta ostaje održana nakon svakog izvršavanja tela petlje. Pretpostavimo da invarijanta važi za promenljive z, n, tj. da važi $(n \leq x) \wedge (z = n \cdot y)$. Nakon izvršavanja tela petlje, promenljive imaju vrednosti $z' = z + y$ i $n' = n + 1$. Potrebno je pokazati da ove nove vrednosti zadovoljavaju invarijantu, tj. da važi $n' \leq x \wedge z' = n' \cdot y$. Zaista, pošto je telo petlje izvršavano, važi da je $n < x$. Dakle, $n' = n + 1 \leq x$. Takođe, pošto je $z' = z + y$, a invarijanta je važila za z i n, važi da je $z' = n \cdot y + y$. Dalje, $z' = (n+1) \cdot y = n' \cdot y$, te je invarijanta zadovoljena i nakon izvršenja tela petlje.*

Na kraju, pokažimo da dokazana invarijanta obezbeđuje korektnost. Kada se izađe iz petlje, uslov nije bio ispunjen tako da važi $n \geq x$. Kombinovanjem sa invarijantom, dobija se da je $n = x$, tj. da je $z = x \cdot y$.

Primer 4.4. *Razmotrimo program koji vrši deljenje uzastopnim oduzimanjem.*

```
r = x; q = 0;
while (y <= r) {
  r = r - y;
  q = q + 1;
}
```

Moguće je pokazati da u svakom koraku važi da je $x = q \cdot y + r$ te je ovo jedna invarijanta petlje. S obzirom na to da je na eventualnom kraju izvršavanja programa $r < y$, na osnovu ove invarijante direktno sledi korektnost (tj. q sadrži količnik, a r ostatak pri deljenju broja x brojem y).

Primer 4.5. *Razmotrimo program koji vrši pronalaženje minimuma niza brojeva.*

```
m = a[0];
for (j = 1; j < n; j++)
    if (a[j] < m)
        m = a[j];
```

Ovaj program ima smisla samo za neprazne nizove, pri čemu se još podrazumeva da je n dužina niza a. Invarijanta petlje je da promenljiva m sadrži minimum dela niza između pozicija 0 i j (tj. minimum elemenata od $a[0]$ do $a[j-1]$). Nakon završetka petlje, važi da je $j = n$, i pošto je n dužina niza iz invarijante sledi korektnost programa.

Formalni dokazi i Horova logika

Sva dosadašnja razmatranja o korektnosti programa vršena su neformalno, tj. nije postojao precizno opisan formalni sistem u kome bi se vršilo dokazivanje korektnosti imperativnih programa. Jedan od najznačajnijih formalnih sistema ovog tipa opisao je Toni Hor (Tony Hoare) u svom radu „An Axiomatic Basis for Computer Programming“. Istraživači koji su dali doprinos na ovom polju pre i nakon Hora su, između ostalih, Robert Flojd (Robert Floyd), Džon Fon Nojman (John Von Neumann), Herman Goldštajn (Herman Goldstine), Alan Tjuring (Alan Turing) i Džon Mekarti (John McCarthy).

Formalni dokazi (u jasno preciziranom logičkom okviru) su interesantni jer mogu da se generišu automatski uz pomoć računara ili barem interaktivno u saradnji čoveka sa računarom. U oba slučaja, formalni dokaz može da se proveri automatski, dok automatska provera neformalnog dokaza nije moguća. Takođe, kada se program i dokaz njegove korektnosti istovremeno razvijaju, programer bolje razume sam

program i proces programiranja uopšte. Informatičari su zainteresovani za formalne dokaze jer metodologija dokaza utiče na razmatranje preciznosti, konzistentnosti i kompletnosti specifikacije, na jasnoću implementacije i konzistentnost implementacije i specifikacije. Zahvaljujuci tome dobija se pouzdaniji softver, čak i onda kada se formalni dokaz ne izvede eksplicitno.

Semantika određenog programskog koda može se zapisati trojkom oblika

$$\{\varphi\}\mathtt{P}\{\psi\}$$

gde je P programski kôd, a $\{\varphi\}$ i $\{\psi\}$ su logičke formule koje opisuju veze između promenljivih koje se javljaju u programu. Trojku $(\varphi, \mathtt{P}, \psi)$ nazivamo *Horova trojka*. Interpretacija trojke je sledeća: „Ako izvršenje programa P počinje sa vrednostima ulaznih promenljivih (kažemo i ‚u stanju') koje zadovoljavaju uslov $\{\varphi\}$ i ako se program P završi u konačnom broju koraka, tada vrednosti programskih promenljivih (‚stanje') zadovoljavaju uslov $\{\psi\}$". Uslov $\{\varphi\}$ naziva se *preduslov* za algoritam (program, iskaz) P, a uslov $\{\psi\}$ naziva se *postuslov (posleuslov)* za algoritam (program, naredbu) P.

Na primer, trojka $\{x = 1\}\mathtt{y:=x}\{y = 1\}$[1], opisuje dejstvo naredbe dodele i kaže da, ako je vrednost promenljive x bila jednaka 1 pre izvršavanja naredbe dodele, i ako se naredba dodele izvrši, tada će vrednost promenljive y biti jednaka 1. Ova trojka je tačna. Sa druge strane, trojka $\{x = 1\}\mathtt{y:=x}\{y = 2\}$ govori da će nakon dodele vrednost promenljive y biti jednaka 2 i ona nije tačna.

Formalna specifikacija programa može se zadati u obliku Horove trojke. U tom slučaju preduslov opisuje uslove koji važe za ulazne promenljive, dok postuslov opisuje uslove koje bi trebalo da zadovolje rezultati izračunavanja. Na primer, program P za množenje brojeva x i y, koji rezultat smešta u promenljivu z bi trebalo da zadovolji trojku $\{\top\}\mathtt{P}\{z = x \cdot y\}$ ($\top$ označava iskaznu konstantu *tačno*, i ovom primeru znači da nema preduslova koje vrednosti x i y treba da zadovoljavaju). Ako se zadovoljimo time da program može da množi samo nenegativne brojeve, specifikacija se može oslabiti u $\{x \geq 0 \wedge y \geq 0\}\mathtt{P}\{z = x \cdot y\}$.

Jedno od ključnih pitanja za verifikaciju je pitanje da li je neka Horova trojka tačna (tj. da li program zadovoljava datu specifikaciju). Hor je dao formalni sistem (aksiome i pravila izvođenja) koji omogućava da se tačnost Horovih trojki dokaže na formalan način (slika 4.2). Za svaku sintaksnu konstrukciju programskog jezika koji se razmatra formiraju se aksiome i/ili pravila izvođenja koji aksiomatski daju opis semantike odgovarajućeg konstrukta programskog jezika.[2]

Pravilo posledice (Cons)

$$\frac{\varphi' \Rightarrow \varphi \quad \{\varphi\}\mathtt{P}\{\psi\} \quad \psi \Rightarrow \psi'}{\{\varphi'\}\mathtt{P}\{\psi'\}}$$

Pravilo kompozicije (Comp)

$$\frac{\{\varphi\}\mathtt{P}_1\{\mu\} \quad \{\mu\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2\{\psi\}}$$

Aksioma dodele (assAx)

$$\frac{}{\{\varphi[x \to E]\}\mathtt{x} := E\{\varphi\}}$$

Pravilo grananja (ifAx)

$$\frac{\{\varphi \wedge c\}\mathtt{P}_1\{\psi\} \quad \{\varphi \wedge \neg c\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{if}\ c\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2\{\psi\}}$$

Pravilo petlje (whileAx)

$$\frac{\{\varphi \wedge c\}\mathtt{P}\{\varphi\}}{\{\varphi\}\mathtt{while}\ c\ \mathtt{do}\ P\{\neg c \wedge \varphi\}}$$

Slika 4.2: Horov formalni sistem

Pravilo posledice. Pravilo posledice govori da je moguće ojačati predulsov, kao i oslabiti postuslov svake trojke. Na primer, od tačne trojke $\{x = 1\}\mathtt{y:=x}\{y = 1\}$, moguće je dobiti tačnu trojku $\{x =$

[1] U ovom poglavlju, umesto C-ovske, biće korišćena sintaksa slična sintaksi korišćenoj u originalnom Horovom radu.

[2] U svom originalnom radu, Hor je razmatrao veoma jednostavan programski jezik (koji ima samo naredbu dodele, naredbu grananja, jednu petlju i sekvencijalnu kompoziciju naredbi), ali u nizu radova drugih autora Horov originalni sistem je proširen pravilima za složenije konstrukte programskih jezika (funkcije, pokazivače, itd.).

1}`y:=x`{$y > 0$}. Slično, na primer, ako program `P` zadovoljava {$x \geq 0$}`P`{$z = x \cdot y$}, tada će zadovoljavati i trojku {$x \geq 0 \wedge y \geq 0$}`P`{$z = x \cdot y$}.

Pravilo kompozicije. Pravilom kompozicije opisuje se semantika sekvencijalnog izvršavanja dve naredbe. Kako bi trojka {φ}$\texttt{P}_1; P_2${ψ} bila tačna, dovoljno je da postoji formula μ koja je postuslov programa $\texttt{P}_1$ za preduslov φ i preduslov programa $\texttt{P}_2$ za postuslov ψ. Na primer, iz trojki {$x = 1$}`y := x + 1`{$y = 2$} i {$y = 2$}`z := y + 2`{$z = 4$}, može da se zaključi {$x = 1$}`y := x + 1; z := y + 2`{$z = 4$}.

Aksioma dodele. Shema aksioma dodele definiše semantiku naredbe dodele. Izraz $\varphi[x \to E]$ označava logičku formulu koja se dobije kada se u formuli φ sva slobodna pojavljivanja promenljive x zamene izrazom E. Na primer, jedna od instanci ove sheme je i $\{x+1 = 2\}y := x+1\{y = 2\}$. Zaista, preduslov $x + 1 = 2$ se može dobiti tako što se sva pojavljivanja promenljive kojoj se dodeljuje (u ovom slučaju y) u izrazu postuslova $y = 2$ zamene izrazom koji se dodeljuje (u ovom slučaju $x + 1$). Nakon primene pravila posledice, moguće je izvesti trojku $\{x = 1\}y := x + 1\{y = 2\}$. Potrebno je naglasiti da se podrazumeva da izvršavanje dodele i sračunavanje izraza na desnoj strani ne proizvodi nikakve bočne efekte do samog efekta dodele (tj. izmene vrednosti promenljive sa leve strane dodele) koji je implicitno i opisan navedenom aksiomom.

Pravilo grananja. Pravilom grananja definiše se semantika `if-then-else` naredbe. Korektnost ove naredbe se svodi na ispitivanje korektnosti njene `then` grane (uz mogućnost korišćenja uslova grananja u okviru preduslova) i korektnosti njene `else` grane (uz mogućnost korišćenja negiranog uslova grananja u okviru preduslova). Opravdanje za ovo, naravno, dolazi iz činjenice da ukoliko se izvršava `then` grana uslov grananja je ispunjen, dok, ukoliko se izvršava `else` grana, uslov grananja nije ispunjen.

Pravilo petlje. Pravilom petlje definiše se semantika `while` naredbe. Uslov φ u okviru pravila predstavlja invarijantu petlje. Kako bi se dokazalo da je invarijanta zadovoljena nakon izvršavanja petlje (pod pretpostavkom da se petlja zaustavlja), dovoljno je pokazati da telo petlje održava invarijantu (uz mogućnost korišćenja uslova ulaska u petlju u okviru preduslova). Opravdanje za ovo je, naravno, činjenica da se telo petlje izvršava samo ako je uslov ulaska u petlju ispunjen.

Primer 4.6. *Dokažimo u Horovom sistemu da je klasičan algoritam razmene (`swap`) vrednosti dve promenljive korektan.*

(1) {$x = X \wedge y = Y$} `t := x` {$t = X \wedge y = Y$}
assAx

(2) {$t = X \wedge y = Y$} `x := y` {$t = X \wedge x = Y$}
assAx

(3) {$x = X \wedge y = Y$} `t := x; x := y` {$t = X \wedge x = Y$}
Comp (1) i (2)

(4) {$t = X \wedge x = Y$} `y := t` {$y = X \wedge x = Y$}
assAx

(5) {$x = X \wedge y = Y$} `t := x; x := y; y := t` {$y = X \wedge x = Y$}
Comp (3) i (4)

Primer 4.7. *Dokažimo u Horovom sistemu da kôd*

```
z = 0; n = 0;
while(n < x) {
  z = z + y;
  n = n + 1;
}
```

vrši množenje brojeva x i y, pod uslovom da su x i y celi brojevi i da je x nenegativan. Izvođenje je dato u tabeli 4.1.

(1) $\{x \geq 0 \wedge 0 = 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0\}$
assAx

(2) $\{x \geq 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0\}$,
Cons (1) jer $x \geq 0 \Rightarrow x \geq 0 \wedge 0 = 0$

(3) $\{x \geq 0\}$ `z = 0;` $\{x \geq 0 \wedge z = 0 \wedge 0 = 0\}$
Cons (2) jer $x \geq 0 \wedge z = 0 \Rightarrow x \geq 0 \wedge z = 0 \wedge 0 = 0$

(4) $\{x \geq 0 \wedge z = 0 \wedge 0 = 0\}$ `n = 0;` $\{x \geq 0 \wedge z = 0 \wedge n = 0\}$
assAx

(5) $\{x \geq 0\}$ `z = 0; n = 0;` $\{x \geq 0 \wedge z = 0 \wedge n = 0\}$
Comp (3) i (4)

(6) $\{x \geq 0\}$ `z = 0; n = 0;` $\{n \leq x \wedge z = n * y\}$
Cons (5) jer $x \geq 0 \wedge z = 0 \wedge n = 0 \Rightarrow n \leq x \wedge z = n \cdot y$

(7) $\{n \leq x \wedge z + y = (n+1) \cdot y \wedge n < x\}$ `z = z+y;` $\{n \leq x \wedge z = (n+1) \cdot y \wedge n < x\}$
assAx

(8) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y;` $\{n \leq x \wedge z = (n+1) \cdot y \wedge n < x\}$
Cons(7) jer $n \leq x \wedge z = n \cdot y \wedge n < x \Rightarrow n \leq x \wedge z + y = (n+1) \cdot y \wedge n < x$

(9) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y;` $\{n + 1 \leq x \wedge z = (n+1) \cdot y\}$
Cons(8) jer $n \leq x \wedge z = (n+1) \cdot y \wedge n < x \Rightarrow n + 1 \leq x \wedge z = (n+1) \cdot y$

(10) $\{n + 1 \leq x \wedge z = (n+1) \cdot y\}$ `n = n+1;` $\{n \leq x \wedge z = n \cdot y\}$
assAx

(11) $\{n \leq x \wedge z = n \cdot y \wedge n < x\}$ `z = z+y; n = n+1;` $\{n \leq x \wedge z = n \cdot y\}$
Comp (9) i (10)

(12) $\{n \leq x \wedge z = n \cdot y\}$ `while(n < x) { z = z + y; n = n + 1; }` $\{n \geq x \wedge n \leq x \wedge z = n * y\}$
(whileAx)

(13) $\{n \leq x \wedge z = n \cdot y\}$ `while(n < x) { z = z + y; n = n + 1; }` $\{z = x \cdot y\}$
Cons(12) jer $n \geq x \wedge n \leq x \wedge z = n * y \Rightarrow z = x * y$

(14) $\{x \geq 0\}$ `z = 0; n = 0; while(n < x) {z = z + y; n = n + 1;}` $\{z = x \cdot y\}$
Comp (6) i (13)

Tabela 4.1: Primer dokaza u Horovoj logici

4.3.3 Ispitivanje zaustavljanja programa

S obzirom na to da je halting problem neodlučiv, zaustavljanje svakog programa mora se pokazivati na zaseban način (tj. ne postoji opšti postupak koji bi se primenio na sve programe). U programima kakvi su do sada razmatrani jedine naredbe koje mogu da dovedu do nezaustavljanja su petlje, tako da je potrebno dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[3] takva da su susedna stanja kroz koje se prolazi tokom izvršavanja petlje međusobno u relaciji. Kod elementarnih algoritama ovo se obično radi tako što se izvrši neko preslikavanje stanja u skup prirodnih brojeva i pokaže se da se svako susedno stanje preslikava u manji prirodan broj. Pošto je relacija $>$ na skupu prirodnih brojeva dobro zasnovana, i ovako definisana relacija na skupu stanja će biti dobro zasnovana.

Primer 4.8. *Algoritam koji vrši množenje uzastopnim sabiranjem se zaustavlja. Zaista, u svakom koraku petlje vrednost $x - n$ je prirodan broj (jer invarijanta kaže da je $n \leq x$). Ova vrednost opada kroz svaki korak petlje (jer se x ne menja, a n raste), tako da u jednom trenutku mora da dostigne vrednost* 0.

Primer 4.9. *Ukoliko se ne zna širina podatka tipa* `unsigned int`*, nije poznato da li se naredna funkcija zaustavlja za proizvoljnu ulaznu vrednost* `n`*:*

```
int f(unsigned int n) {
  while(n>1) {
    if (n%2)
      n = 3*n+1;
    else
      n = n/2
  }
}
```

[3] Za relaciju $\succ$ se kaže da je *dobro zasnovana* (eng. well founded) ako ne postoji beskonačan opadajući lanac elemenata $a_1 \succ a_2 \succ \ldots$.

Opšte uverenje je da se funkcija zaustavlja za svako `n` *(to tvrdi još uvek nepotvrđena* Kolacova (Collatz) hipoteza *iz 1937). Navedeni primer pokazuje kako pitanje zaustavljanja čak i za neke veoma jednostavne programe može da bude ekstremno komplikovano.*

(Ukoliko je poznata širina podatka `unsigned int`*, i ukoliko se testiranjem za sve moguće ulazne vrednosti pokaže da se* `f` *zaustavlja, to bi dalo odgovor na pitanje u specijalnom slučaju.)*

Pitanja i zadaci za vežbu

Pitanje 4.1. *Šta je cilj verifikacije programa?*

Pitanje 4.2. Testiranjem *programa se može:*
(a) *dokazati da je program korektan za sve ulaze,*
(b) *opovrgnuti da je program korektan za sve ulaze,*
(c) *dokazati da se program zaustavlja za sve ulaze,*
(d) *dokazati da se program ne zaustavlja za neke ulaze.*
Zaokružiti sve tačne odgovore.

Pitanje 4.3. *Kako se zove provera korektnosti u fazi izvršavanja programa?*

Pitanje 4.4. *Koji je najčešći vid dinamičke verifikacije programa?*

Pitanje 4.5. *Koliko bi, metodom iscrpnog testiranja, bilo potrebno izvršiti testova programa za sabiranje dva neoznačena 32-bitna cela broja?*

Pitanje 4.6. *Šta se može dokazati testiranjem programa?*

Pitanje 4.7. *U čemu je razlika izmedju parcijalne i totalne korektnosti programa?*

Pitanje 4.8. *Da li uz pomoć debagera može da se:*
(a) efikasnije kompilira program?
(b) lakše otkrije greška u programu?
(c) izračuna složenost izvršavanja programa?
(d) registruje „curenje memorije“ u programu?
(e) umanji efekat fragmentisanja memorije?

Pitanje 4.9. *Proveriti ispravnost narednog kôda koji bi trebalo da štampa karaktere niske* `s` *po jedan u redu:*

```
i=0;
do {
  putchar(s[i++]);
  putchar('\n');
} while (s[i] != '\0') ;
```

Pitanje 4.10. *Proveriti ispravnost narednog kôda koji bi trebalo da poredi dva cela broja* `i` *i* `j`*:*

```
if (i > j)
  printf("%d je vece od %d.\n", i, j);
else
  printf("%d je manje od %d.\n", i , j);
```

Pitanje 4.11. *Kako se dokazuje korektnost rekurzivnih funkcija?*

Pitanje 4.12. *Dokazati da naredna funkcija vraća vrednost* `x+y` *(pod pretpostavkom da nema prekoračenja):*

```
unsigned f(unsigned x, unsigned y) {
  if (x == 0) return y;
    else return f(x-1, y) + 1;
}
```

Pitanje 4.13. *Dokazati da naredna funkcija vraća vrednost* `x*y` *(pod pretpostavkom da nema prekoračenja):*

```
int mnozi(int x, int y) {
  if (x == 0) return 0;
    else return mnozi(x - 1, y) + y;
}
```

Pitanje 4.14. *Kako se zove formula koja uključuje vrednosti promenljivih koje se javljaju u nekoj petlji i koja važi pri svakom ispitivanju uslova petlje?*

Pitanje 4.15. *Kako se zovu relacije koje se koriste u dokazivanju ispravnosti programa koji sadrže petlje?*

Pitanje 4.16.

Ako je $x \geq 0$ *i celobrojno, navesti invarijantu koja obezbeđuje korektnost sledećeg algoritma sabiranja:*
`z = y; n = 0; while(n < x) { z = z + 1; n = n + 1; }`

Pitanje 4.17. *Ako je* $x \geq 0$ *i celobrojno, navesti invarijantu koja obezbeđ uje korektnost sledećeg algoritma stepenovanja:*
`z = 1; n = 0; while(n < x) { z = z * y; n = n + 1; }`

Pitanje 4.18. *Kako se interpretira trojka* $\{\varphi\}P\{\psi\}$*?*

Pitanje 4.19. *Da li je zadovoljena sledeća Horova trojka:*
$\{x > 1\}n := x\{n > 1 \wedge x > 0\}$

Pitanje 4.20. *Navesti barem jedno pravilo Horovog formalnog sistema.*

Pitanje 4.21. *U* Horovoj trojci $\{\varphi\}P\{\psi\}$, φ *je* program/formula, P *je* program/formula *i* ψ *je* program/formula.
(zaokružiti ispravno).

Pitanje 4.22. *Dopuniti sledeću Horovu trojku tako da ona bude zadovoljena:*

1. {.....} `if (a>b) then z:=a; else z:=b;` $\{z < 9\}$
2. {.....} `if (a<b) then z:=a; else z:=b;` $\{z > 0\}$
3. {.....} `if (a<b) then z:=a; else z:=b;` $\{z > 3\}$
4. {.....} `a:=b; c:=a;` $\{c > 3\}$
5. {.....} `n:=x;` $\{n > 3\}$

Pitanje 4.23. *Navesti primer petlje za koju se ne zna da li se zaustavlja.*

Deo II

Osnovi algoritmike

Glava 5

Rekurzija

U matematici i računarstvu, rekurzija je pristup u kojem se neki pojam, objekat ili funkcija definiše na osnovu jednog ili više baznih slučajeva i na osnovu pravila koja složene slučajeve svode na jednostavnije. Na primer, pojam *predak* može se definisati na sledeći način:

- Roditelj osobe je predak te osobe (bazni slučaj)
- Roditelj bilo kog pretka neke osobe je takođe predak te osobe (rekurzivni korak)

Kaže se da je prethodna definicija rekurzivna (ili induktivna).

Izostanak bilo baznog koraka (koji obezbeđuje „zaustavljanje" primene definicije) bilo rekurzivnog koraka čini definiciju nepotpunom[1].

Matematička indukcija i rekurzija. Rekurzija je tesno povezana sa matematičkom indukcijom. Dokaze zasnovane na matematičkoj indukciji čine (obično trivijalni) dokazi baznog slučaja (na primer, za $n = 0$) i dokazi induktivnog koraka: pod pretpostavkom da tvrđenje važi za n dokazuje se da tvrđenje važi za $n + 1$. Rekurzivne definicije imaju sličan oblik.

- Bazni slučaj rekurzivne definicije je slučaj koji može biti rešen bez rekurzivnog poziva;
- U rekurzivnom koraku, za vrednost n, pretpostavljamo da je definicija raspoloživa za vrednost $n - 1$.

Princip dokazivanja matematičkom indukcijom je u vezi sa induktivnom definicijom skupa prirodnih brojeva — skup prirodnih brojeva je najmanji skup koji sadrži nulu i zatvoren je za operaciju sledbenika. Iz ovoga proističe da je algoritme za rad sa prirodnim brojevima ponekad moguće definisati tako da u slučaju izlaska iz rekurzije razrešavaju slučaj nule, dok u slučaju nekog broja većeg od nule rekurzivno svode problem na slučaj njegovog prethodnika. Ovakav tip rekurzije, koji direktno odgovara induktivnoj definiciji tipa podataka, naziva se *primitivna rekurzija*.

Na sličan način, moguće je induktivno predstaviti i druge tipove (skupove) podataka, što ih onda čini pogodnim za primenu primitivne rekurzije.

Na primer, niska čini zapis prirodnog broja ako ili (i) niska je dekadna cifra (bazni slučaj) ili (ii) niska se dobija nadovezivanjem dekadne cifre sa desne strane zapisa prirodnog broja (induktivni korak). U tom slučaju primitivnom rekurzijom je moguće definisati funkcije tako da izlaz iz rekurzije predstavljaju jednocifreni brojevi, dok se u slučaju višecifrenih brojeva `n` rekurzivno razrešava slučaj broja sa odsečenom poslednjom cifrom (`n/10`), najčešće kombinujući dobijeni rezultat sa poslednjom cifrom (`n%10`). Dualni pristup razlaganja broja na prvu cifru i ostale nije pogodan, zbog toga što ovakvo razlaganje nije moguće izračunati jednostavnim matematičkim operacijama (kakvo je deljenje sa ostatkom sa 10).

Dalje, na primer, niz je moguće definisati tako što (i) prazan niz predstavlja niz (bazni slučaj) i (ii) niz dobijen dodavanjem elementa na kraj nekog niza predstavlja niz. Pri ovom razmatranju, primitivnom

[1] Često se citira šala kako se u rečniku rekurzija može najilustrativnije objasniti tako što se pod stavkom *rekurzija* napiše: *vidi rekurzija*.

rekurzijom je moguće definisati funkcije tako da pri izlasku iz rekurzije obrađuju slučaj praznog niza, dok slučaj nepraznog niza dužine `n` rešavaju tako što rekurzivno razreše njegov prefiks dužine `n-1` i onda rezultat iskombinuju sa poslednjim elementom niza. Dualno razlaganje na prvi element i sufiks niza je takođe moguće, pri čemu je onda prilikom svakog rekurzivnog poziva potrebno, pored indeksa, prosleđivati i poziciju početka sufiksa.

U nekim slučajevima, potrebno je koristiti i naprednije oblike indukcije, kakva je, na primer, *totalna indukcija*. U tom slučaju, nakon dokazivanja induktivne baze, u okviru induktivnog koraka moguće je pretpostaviti tvrđenje za sve brojeve manje od n i iz te pretpostavke dokazati tvrđenje za broj n. Slično, umesto primitivne rekurzije, dozvoljeno je pisati funkcije koje su *totalno (generalno) rekurzivne*. Tako je npr. prilikom razmatranja nekog broja `n`, dozvoljeno izvršiti rekurzivni poziv za bilo koji broj manji od njega (pa čak i više rekurzivnih poziva za različite prirodne brojeve manje od njega). Slično, prilikom implementacije algoritma koji radi sa nekim nizom, dozvoljeno je vršiti rekurzivne pozive za sve podnizove polaznog niza kraće od njega. Kao što će kasnije biti pokazano, razmatranje niza kao unije dva njegova dvostruko kraća podniza često vodi boljoj efikasnosti nego primitivno-rekurzivno razmatranje niza kao unije jednog njegovog elementa i podniza bez tog elementa.

5.1 Primeri rekurzivnih funkcija

U programiranju, rekurzija je tehnika u kojoj funkcija poziva samu sebe, direktno ili indirektno. Rekurzivne funkcije su pogodne za širok spektar informatičkih problema, ali pored svojih dobrih strana imaju i loše.

5.1.1 Faktorijel

Funkcija faktorijel (za prirodni broj n, vrednost $n!$ jednaka je proizvodu svih prirodnih brojeva od 1 do n) može se definisati na (primitivno) rekurzivan način:

- $0! = 1$ (bazni slučaj)
- za $n > 0$ važi: $n! = n \cdot (n-1)!$ (rekurzivni korak)

Vrednost faktorijela se može izračunati korišćenjem petlje, ali i korišćenjem rekurzije, kao što je to prikazano u poglavlju 3.5.1:

```
unsigned faktorijel(unsigned n) {
    if (n == 0)
        return 1;
    else
        return n*faktorijel(n-1);
}
```

Ukoliko je argument funkcije, na primer, vrednost 5, onda se funkcija `f` poziva najpre za tu vrednost, a onda, rekurzivno, za vrednosti `4`, `3`, `2`, `1`, `0`. Prilikom svakog poziva funkcije u stek segmentu memorije stvara se novi stek okvir — stek okvir za novu instancu funkcije `f`. U ovim stek okvirima lokalna promenljiva `n` imaće redom vrednosti `5`, `4`, `3`, `2`, `1`, `0`. Sve instance funkcije `f` koriste isti primerak koda funkcije `f` (koji se nalazi u kôd segmentu). Stek okvir svake instance funkcije `f` „pamti“ dokle je ta instanca funkcije stigla sa izvršavanjem koda (kako bi izvršavanje moglo da bude nastavljeno od te tačke kada ta instanca funkcije ponovo postane aktivna).

Kao što je pokazano u poglavlju 3.5.1, funkcija `faktorijel` ima složenost $O(n)$. Pod pretpostavkom da nema prekoračenja, funkcija `faktorijel` je ispravna, kao što je pokazano u poglavlju 4.3.2.

5.1.2 Sumiranje niza

Sumacija niza brojeva može biti izražena (primitivno) rekurzivno.

- $\sum_{i=1}^{0} a_i = 0$ (bazni slučaj)
- za $n > 0$ važi: $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n-1} a_i + a_n$.

Rekurzivna funkcija koja sumira elemente niza može se definisati na sledeći način:

```
float sum(float a[], unsigned n) {
  if (n == 0)
     return 0.0f;
  else
     return sum(a, n-1) + a[n-1];
}
```

Lako se dokazuje da navedena funkcija `sum` ima složenost $O(n)$.

5.1.3 Stepenovanje

Stepenovanje broja može biti izraženo (primitivno) rekurzivno.

- $x^0 = 1$
- za $k > 0$ važi: $x^k = x \cdot x^{k-1}$.

Vrednost x^k za nenegativne celobrojne izložioce može se jednostavno izračunati sledećom funkcijom:

```
float stepen_sporo(float x, unsigned k) {
    if (k == 0)
        return 1.0f;
    else
        return x * stepen_sporo(x, k - 1);
}
```

Iterativna varijanta prethodne funkcije je:

```
float stepen(float x, unsigned k) {
    unsigned i; float m = 1.0f;
    for(i=0; i<k; i++)
        m *= x;
    return m;
}
```

Sledeće rekurzivno rešenje je znatno brže (zahteva mnogo manje množenja, što će kasnije biti dokazano):

```
float stepen_brzo(float x, unsigned k) {
    if (k == 0)
        return 1.0f;
    else if (k % 2 == 0)
        return stepen_brzo(x * x, k / 2);
    else
        return x * stepen_brzo(x, k - 1);
}
```

Iterativnu verziju prethodne funkcije nije trivijalno napisati.

Izračunajmo vremensku složenost naredne rekurzivne funkcije za stepenovanje u zavisnosti od vrednosti `k` (jer je ona ključna u ovom primeru). Neka $T(k)$ označava broj instrukcija koje zahteva poziv funkcije `stepen_brzo` za ulaznu vrednost `k` (za `k`≥ 0). Za $k = 0$ važi $T(k) = 2$ (jedno poređenje i jedna naredba `return`). Za $k > 1$, ako je n neparan važi $T(k) = T(k-1) + 6$, a ako je paran, važi $T(k) = T(k/2) + 6$. Ukoliko se uvek koristi druga grana, onda na osnovu teoreme 3.1 važi $T(k) = O(\log k)$. Međutim, ovo je za algoritam najbolji slučaj. Najgori je ako se uvek koristi prva grana pa iz veze $T(k) = T(k-1) + 6$ sledi da je $T(k) = O(k)$. No, analizom kôda može se zaključiti da je scenario da se uvek koristi prva grana nemoguć (jer ako je k neparan broj, onda je $k-1$ paran). Dublja analiza pokazuje da je navedeni algoritam složenosti $O(\log k)$.

Dokažimo korektnost navedenog algoritma za brzo stepenovanje. S obzirom da na to da je u ovom slučaju funkcija definisana opštom rekurzijom, dokaz će biti zasnovan na totalnoj indukciji i pratiće sledeću shemu: „Da bi se pokazalo da vrednost $stepen\_brzo(x, k)$ zadovoljava neko svojstvo, može se pretpostaviti da za $n \neq 0$ i k parno vrednost $stepen\_brzo(x \cdot x, k/2)$ zadovoljava to svojstvo, kao i da za $k \neq 0$ i k neparno vrednost $stepen\_brzo(x, k-1)$ zadovoljava to svojstvo, i onda tvrđenje treba dokazati pod tim pretpostavkama“. Slične sheme indukcije se mogu dokazati i za druge funkcije definisane opštom rekurzijom i u principu, one dozvoljavaju da se prilikom dokazivanja korektnosti dodatno pretpostavi da svaki rekurzivni poziv vraća korektan rezultat.

Pređimo na sâm dokaz činjenice da je $stepen\_brzo(x, k) = x^k$ (za nenegativnu vrednost k).

Slučaj: $k = 0$. Tada je $stepen\_brzo(x, k) = stepen\_brzo(x, 0) = 1 = x^0$.

Slučaj: $k \neq 0$. Tada je k ili paran ili neparan.

- **Slučaj:** k **je paran.** Tada je $stepen\_brzo(x, k) = stepen\_brzo(x \cdot x, k/2)$. Na osnovu prve induktivne pretpostavke, važi da je $stepen\_brzo(x \cdot x, k/2) = (x \cdot x)^{k/2}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je $stepen\_brzo(x, k) = (x \cdot x)^{k/2} = (x^2)^{k/2} = x^k$.
- **Slučaj:** k **je neparan.** Tada je $stepen\_brzo(x, k) = x \cdot stepen\_brzo(x, k-1)$. Na osnovu druge induktivne pretpostavke važi da je $stepen\_brzo(x, k-1) = x^{k-1}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je $stepen\_brzo(x, k) = x \cdot x^{k-1} = x^k$.

5.1.4 Fibonačijev niz

Fibonačijev niz (0,1,1,2,3,5,8,13,...) može se definisati u vidu (generalne) rekurzivne funkcije F:

- $F(0) = 0$ i $F(1) = 1$ (bazni slučaj)
- za $n > 1$ važi: $F(n) = F(n-1) + F(n-2)$ (rekurzivni korak)

Funkcija za izračunavanje n-tog elementa Fibonačijevog niza može se definisati na sledeći način:

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

5.1.5 NZD

Euklidov algoritam za izračunavanje najvećeg zajedničkog delioca dva broja se može izračunati narednom rekurzivnom funkcijom.

```
unsigned nzd(unsigned a, unsigned b) {
  if (b == 0)
     return a;
  else
     return nzd(b, a % b);
}
```

5.1.6 Kule Hanoja

Problem kula Hanoja[2] glasi ovako: data su tri tornja i na prvom od njih n diskova opadajuće veličine; zadatak je prebaciti sve diskove sa prvog na treći toranj (koristeći i drugi) ali tako da nikada nijedan disk ne stoji iznad manjeg.

[2]Ovaj problem je u formi autentičnog mita opisao francuski matematičar De Parvil u jednom časopisu 1884.

Iterativno rešenje ovog problema je veoma kompleksno, a rekurzivno prilično jednostavno: ukoliko je $n = 0$, nema diskova koji treba da se prebacuju; inače: prebaci (rekurzivno) $n - 1$ diskova sa polaznog na pomoćni toranj (korišćenjem dolaznog tornja kao pomoćnog), prebaci najveći disk sa polaznog na dolazni toranj i, konačno, prebaci (rekurzivno) $n - 1$ diskova sa pomoćnog na dolazni disk (korišćenjem polaznog tornja kao pomoćnog). U nastavku je implementacija ovog rešenja:

```
void tower(unsigned n, char start, char finish, char spare) {
  if (n > 0) {
     tower(n-1,start,spare,finish);
     printf("Prebaci disk sa kule %c na kulu %c\n",start,finish);
     tower(n-1,spare,finish,start);
  }
}
```

Poziv navedene funkcije `tower(3,'A','C','B')` daje sledeći izlaz:

```
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule A na kulu B
Prebaci disk sa kule C na kulu B
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule B na kulu A
Prebaci disk sa kule B na kulu C
Prebaci disk sa kule A na kulu C
```

Izračunajmo broj prebacivanja diskova koje opisuje navedena funkcija. Važi $T(0) = 0$ i $T(n) = 2T(n-1)+1$ (i $T(1) = 2T(0)+1 = 1$). Iz $T(n) - 2T(n-1) = 1 = T(n-1) - 2T(n-2)$ (za $n > 1$) sledi $T(n) = 3T(n-1) - 2T(n-2)$. Karakteristična jednačina ove veze je $t^2 = 3t - 2$ i njeni koreni su 2 i 1. Iz sistema

$$\alpha \cdot 1 + \beta \cdot 1 = 0$$

$$\alpha \cdot 2 + \beta \cdot 1 = 1$$

sledi $\alpha = 1$ i $\beta = -1$, pa je

$$T(n) = 1 \cdot 2^n + (-1) \cdot 1^n = 2^n - 1 \, .$$

5.2 Uzajamna rekurzija

U dosadašnjim primerima, rekurzivne funkcije su pozivale same sebe direktno. Postoji i mogućnost da se funkcije međusobno pozivaju i tako stvaraju *uzajamnu rekurziju*.

Naredni primer ilustruje dve funkcije koje se koriste u ispitvanju da li je parametar n (n je prirodan broj) paran broj. Funkcija `paran` rešava problem za vrednost n kada je $n > 0$ primenom funkcije *neparan* za vrednost $n-1$, pri čemu se za svodjenje problema troše tri vremenske jedinice. Funkcija B za vrednost parametra n kada je $n > 0$ poziva funkciju `paran` za vrednost $n - 1$, pri čemu se za svodjenje problema troše tri vremenske jedinice. Za $n = 0$, i funkcija `paran` i funkcija `neparan` troše dve vremenske jedinice. Zadatak je izračunati vreme izvršavanja funkcije `paran`.

```
int paran(int n) {
  if (n==0)
    return 1;
  else
    return neparan(n-1);
```

```
}

int neparan(int n) {
  if (n==0)
    return 0;
  else
    return paran(n-1);
}
```

Označimo sa $P(n)$ vreme izvršavanja funkcije `paran`, a sa $N(n)$ vreme izvršavanja funkcije `neparan` za ulaznu vrednost n. Važi:

$P(0) = 2$

$N(0) = 2$

$P(n) = N(n-1) + 3 \quad (n > 0)$

$N(n) = P(n-1) + 3 \quad (n > 0)$

Kako je $N(n-1) = P(n-2) + 3$, važi:

$P(n) = P(n-2) + 3 + 3$

Dalje se jednostavno pokazuje da važi $P(n) = 3n + 2$ i $N(n) = 3n + 2$.

Naredni primer ilustruje dve funkcije koje se koriste u rešavanju nekog problema P koji ima parametar n (n je ceo broj). Problem P rešava se primenom algoritma A. Algoritam A rešava problem za vrednost n ($n > 0$) primenom algoritma B za vrednost $n-1$, pri čemu se za svodjenje problema troši n vremenskih jedinica. Algoritam B rešava problem za vrednost n ($n > 0$) primenom algoritma A za vrednost $n - 1$, pri čemu se za svodjenje problema troši n vremenskih jedinica. Problem za $n = 0$, algoritam A rešava
R: uskladiti trivijalno za jednu vremensku jedinicu, a algoritam B za dve vremenske jedinice. Izračunati vreme izvršavanja algoritma A za ulaznu vrednost n.

```
void A(int n) {
  if (n <= 0)
    return;
  n--;
  B(n-1);
}

void B(int n) {
  if (n <= 0)
    return;
  n--;
  A(n-1);
}
```

Označimo sa $A(n)$ vreme izvršavanja algoritma A, a sa $B(n)$ vreme izvršavanja algoritma B za ulaznu vrednost n. Na osnovu uslova zadatka je:

$A(1) = 1$

$B(1) = 2$

$A(n) = B(n-1) + n \quad (n > 1)$

$B(n) = A(n-1) + n \quad (n > 1)$

Kako je $B(n-1) = A(n-2) + n - 1$, zaključujemo da je

$A(n) = A(n-2) + n - 1 + n$

Primenom matematičke indukcije dokažimo da za neparne vrednosti n važi $A(n) = \frac{n(n+1)}{2}$.

Tvrdjenje važi za $n = 1$ jer je $A(1) = 1 = \frac{1 \cdot 2}{2}$.

Pretpostavimo da je tvrdjenje tačno za neki neparan broj n i dokažimo da važi i za sledeći neparan broj - $n + 2$:

$A(n+2) = B(n+1) + n + 2 = A(n) + n + 1 + n + 2$

$A(n+2) = \frac{n(n+1)}{2} + n + 1 + n + 2$

$A(n+2) = \frac{n(n+1)+2(n+1)+2(n+2)}{2}$

$A(n+2) = \frac{(n+1)(n+2)+2(n+2)}{2}$

$A(n+2) = \frac{(n+2)(n+1+2)}{2}$
$A(n+2) = \frac{(n+2)(n+3)}{2}$
Dakle, na osnovu principa matematičke indukcije tvrdjenje važi za sve neparne brojeve.
Dokažimo da za parne vrednosti n važi $A(n) = \frac{n(n+1)}{2} + 1$.
Za $n = 2$ tvrdjenje je tačno: $A(2) = B(1) + 2 = 2 + 2 = \frac{2 \cdot 3}{2} + 1$.
Pretpostavimo da je tvrdjenje tačno za neki paran broj n i dokažimo da je tačno i za $n + 2$, tj. za sledeći paran broj:
$A(n+2) = B(n+1) + n + 2 = A(n) + n + 1 + n + 2$
$A(n+2) = \frac{n(n+1)}{2} + 1 + n + 1 + n + 2$
$A(n+2) = \frac{n(n+1)+2(n+1)+2(n+2)}{2} + 1$
$A(n+2) = \frac{(n+1)(n+2)+2(n+2)}{2} + 1$
$A(n+2) = \frac{(n+2)(n+1+2)}{2} + 1$
$A(n+2) = \frac{(n+2)(n+3)}{2} + 1$
Dakle, tvrdjenje je tačno za sve parne brojeve.
Konačno, važi:

$$A(n) = \begin{cases} \frac{n(n+1)}{2} + 1 & \text{za } n \text{ parno} \\ \frac{n(n+1)}{2} & \text{za } n \text{ neparno} \end{cases}$$

5.3 Dobre i loše strane rekurzije

Dobre strane rekurzije su (obično) čitljiv i kratak kod, jednostavan za razumevanje, analizu, dokazivanje korektnosti i održavanje. Ipak, rekurzivna rešenja imaju i mana.

Cena poziva. Prilikom svakog rekurzivnog poziva kreira se novi stek okvir i kopiraju se argumenti funkcije. Kada rekurzivnih poziva ima mnogo, ovo može biti veoma memorijski i vremenski zahtevno, te je poželjno rekurzivno rešenje zameniti iterativnim.

Suvišna izračunavanja. U nekim slučajevima prilikom razlaganja problema na manje potprobleme dolazi do preklapanja potproblema i do višestrukih rekurzivnih poziva za iste potprobleme.

Razmotrimo, na primer, izvršavanje navedene funkcije koja izračunava elemente Fibonačijevog niza za $n = 5$. U okviru tog izvršavanja, funkcija f je 3 puta pozvana za $n = 0$, 5 puta za $n = 1$, itd. Naravno, na primer, poziv $f(1)$ je svaki put izvršavan iznova i nije korišćena vrednost koju je vratio prethodni takav poziv. Zbog ovoga, izvršava se mnogo suvišnih izračunavanja i količina takvih izračunavanja u ovom primeru raste. Kako bi se izbegla suvišna izračunavanja moguće je koristiti tehniku *memoizacije*, koja podrazumeva da se u posebnoj strukturi podataka čuvaju svi rezultati već završenih rekurzivnih poziva. Pri svakom ulasku u funkciju konsultuje se ova struktura i, ako je rezultat već poznat, vraća se prethodno izračunat rezultat.

```
unsigned memo[MAX_FIB];
unsigned fib(unsigned n) {
  if (memo[n]) return memo[n];
  if(n == 0 || n == 1)
        return memo[n] = n;
    else
        return memo[n] = fib(n-1) + fib(n-2);
}
```

Drugi pristup rešavanja problema suvišnih izračunavanja naziva se *dinamičko programiranje*. Ključna ideja je da se, umesto da se analizirajući problem on svodi na niže ka jednostavnijim potproblemima, od jednostavnih potproblema naviše konstruiše rešenje glavnog problema. Rešenja dinamičkim programiranjem obično ne uključuju rekurziju.

Na primer, gore navedena funkcija `fib` može se zameniti iterativnom funkcijom koja od početnih elemenata niza postepeno sintetiše sve dalje i dalje elemente niza. Primetimo da za izračunavanje n-tog elementa niza nije neophodno pamtiti sve elemente niza do indeksa `n` već samo dva prethodna:

```
int fib(int n) {
    int i, fpp, fp;

    if (n == 0 || n == 1)
        return n;

    fpp = 0;
    fp = 1;

    for(i = 2; i <= n; i++) {
        int f = fpp + fp;
        fpp = fp;
        fp = f;
    }

    return fp;
}
```

5.4 Eliminisanje rekurzije

Svaku rekurzivnu funkciju je moguće implementirati na drugi način tako da ne koristi rekurziju. Ne postoji jednostavan opšti postupak za generisanje takvih alternativnih implementacija[3].

Međutim, takav postupak postoji za neke specijalne slučajeve. Naročito je zanimljiv slučaj *repne rekurzije* jer se u tom slučaju rekurzija može jednostavno eliminisati[4]. Rekurzivni poziv je *repno rekurzivni* ukoliko je vrednost rekurzivnog poziva upravo i konačan rezultat funkcije, tj. nakon rekurzivnog poziva funkcije ne izvršava se nikakva dodatna naredba. U tom slučaju, nakon rekurzivnog poziva nema potrebe vraćati se u kôd pozivne funkcije, te nema potrebe na stek smeštati trenutni kontekst (vrednosti lokalnih promenljivih). Na primer, u funkciji

```
float stepen_brzo(float x, unsigned k) {
    if (k == 0)
        return 1.0f;
    else if (k % 2 == 0)
        return stepen_brzo(x * x, k / 2);
    else
        return x * stepen_brzo(x, k - 1);
}
```

prvi rekurzivni poziv je repno rekurzivan, dok drugi nije (zato što je po izlasku iz rekurzije neophodno još pomnožiti rezultat sa `x`).

Pokažimo na narednom kôdu kako je moguće eliminisati repnu rekurziju.

```
void r(int x) {
   if (p(x))
      a(x);
   else {
      b(x);
      r(f(x));
   }
}
```

gde su `p`, `a`, `b` i `f` proizvoljne funkcije.

Ključni korak je da se pre rekurzivnog koraka vrednost parametara funkcije (u ovom slučaju promenljive `x`) postavi na vrednost parametra rekurzivnog poziva, a zatim da se kontrola toka izvršavanja

[3] Opšti postupak bi uključivao implementaciju strukture podataka u koju bi eksplicitno bili smeštani podaci koji se inače smeštaju na programski stek.

[4] Neki kompilatori (pre svega za funkcionalne programske jezike) automatski detektuju repno rekurzivne pozive i eliminišu ih.

nekako prebaci na početak funkcije. Ovo je najjednostavnije (ali ne previše elegantno) uraditi korišćenjem bezuslovnog skoka.

```
void r(int x) {
pocetak:
   if (p(x))
      a(x);
   else {
      b(x);
      x = f(x); goto pocetak;
   }
}
```

Daljom analizom moguće je ukloniti bezuslovni skok i dobiti sledeći iterativni kôd.

```
void r(int x) {
   while(!p(x)) {
      b(x);
      x = f(x);
   }
   a(x);
}
```

Demonstrirajmo tehniku ulanjanja repne rekurzije i na primeru Euklidovog algoritma.

```
unsigned nzd(unsigned a, unsigned b) {
  if (b == 0)
     return a;
  else
     return nzd(b, a % b);
}
```

Pošto je poziv repno rekurzivan, potrebno je pripremiti nove vrednosti promenljivih `a` i `b` i preneti kontrolu izvršavanja na početak funkcije.

```
unsigned nzd(unsigned a, unsigned b) {
pocetak:
  if (b == 0)
     return a;
  else {
     unsigned tmp = a % b; a = b; b = tmp;
     goto pocetak;
  }
}
```

Daljom analizom, jednostavno se uklanja `goto` naredba.

```
unsigned nzd(unsigned a, unsigned b) {
  while (b != 0)
     unsigned tmp = a % b; a = b; b = tmp;
  }
  return a;
}
```

Pitanja i zadaci za vežbu

Pitanje 5.1. *Za šta se koristi matematička indukcija, a za šta rekurzija?*

Pitanje 5.2. *Šta je potrebno da bi rekurzivna veza bila ispravno definisana?*

Pitanje 5.3. *Za svaki rekurzivni poziv, na programskom steku se stvara novi stek okvir za tu funkciju (tačno/netačno).*

Pitanje 5.4. *Za svaki rekurzivni poziv, u kôd segmentu se stvara nova kopija kôda te funkcije (tačno/netačno).*

Pitanje 5.5. *Za svaki rekurzivni poziv, na hipu se rezerviše novi prostor za tu funkciju. (tačno/netačno)*

Pitanje 5.6. *Za svaki rekurzivni poziv, u segmentu podataka se rezerviše novi prostor za njene promenljive. (tačno/netačno)*

Pitanje 5.7. *Napisati rekurzivnu funkciju za izračunavanje faktorijela prirodnog broja. Koja je složenosti napisane funkcije?*

Pitanje 5.8. *Napisati rekurzivnu funkciju koja rešava problem "kule Hanoja".*

Pitanje 5.9. *Funkcija* `f` *je definisana sa:*

```
unsigned f(unsigned x) {
   if (x == 0) return 1; else return 2*x + f(x - 1);
}
```

Šta je rezultat poziva `f(4)`*?*

Pitanje 5.10. *Funkcija* `f` *je definisana sa:*

```
unsigned f(unsigned x) {
   if (x == 0) return 1; else return x + 2*f(x - 1);
}
```

Šta je rezultat poziva `f(4)`*?*

Pitanje 5.11. *Funkcija* `f` *je definisana sa:*

```
void f(int i) {
   if (i > 0) {
      f(i / 2);
      printf("%d", i % 2);
   }
}
```

Šta je rezultat poziva `f(14)`*?*

Pitanje 5.12. *Funkcija:*

```
int f(int x) {
  if (x==0)
    return 1;
  else
    return (x%10)*f(x/10);
}
```

Šta je rezultat poziva `f(1234)`*?*

Pitanje 5.13. *Funkcija* `f` *je definisana sa:*

```
void f(int i) {
   if (i > 0)
      return f(i/10)+i%10;
   else
      return 0;
}
```

Šta je rezultat poziva `f(2010)`*?*

Pitanje 5.14. *Funkcija* `f` *je definisana sa:*

```
int f(int n) {

  if (n == 0) return 1;
  if (n == 1) return 2;
  if (n % 2 == 0) return f(n / 2);
  else return f(n / 2) + 1;
}
```

Šta je rezultat poziva `f(472)`*?*

Pitanje 5.15. *Funkcija* `f` *definisana je sa:*

```
void f(unsigned n) {
   if (n > 0) { f(n-1); putchar('.'); f(n-1); }
}
```

Koliko tačkica ispisuje poziv `f(n)`*?*

Pitanje 5.16. *Navesti rekurzivnu verziju Euklidovog algoritma.*

Pitanje 5.17. *Dopuniti funkciju tako da računa najveći zajednički delilac:*

```
unsigned nzd(unsigned a, unsigned b) {
   if (b == 0) return          ;
   return               ;
}
```

Pitanje 5.18. *Dopuniti narednu funkciju koja implementira Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja:*

```
unsigned nzd(unsigned a, unsigned b) {
   if (b == 0)
     return a;

}
```

Pitanje 5.19. *Dopuniti narednu funkciju koja iterativno računa elemente Fibonačijevog niza:*

```
int fib(int n) {
  int f, fpp, fp;

  if (n == 0 || n == 1) return n;
  fpp = 0; fp = 1;
  for(i = 2; i <= n; i++) {
     ..........................
  }
return fp;
}
```

Pitanje 5.20. *Fibonačijev niz je definisan na sledeći način:* $F(1) = 1$, $F(2) = 2$, $F(n+2) = F(n) + F(n+1)$, *za* $n \geq 0$. *Napisati rekurzivnu funkciju koja računa vrednost* n*-tog Fibonačijevog broja. Šta su nedostaci ove funkcije?*

Pitanje 5.21. *Šta je to repna rekurzija?*

Pitanje 5.22. *Kako se zove rekurzivni poziv nakon kojeg nema daljih akcija?*

Pitanje 5.23. *Jedan od obrazaca funkcija kod kojih se može eliminisati rekurzija je:*
(a) glavna rekurzija; (b) repna rekurzija; (c) master rekurzija; (d) linearna rekurzija.

Pitanje 5.24. *Funkcija* `f` *je definisana sa:*

```
unsigned f(unsigned x) {
   if (x == 0) return 1;
   else if (x==1) return f(x - 1);
   else return 3*x + f(x - 1);
```

Zaokružiti sve repno rekurzivne pozive.

Pitanje 5.25.

Funkcija `f` *je definisana sa:*

```
void f(unsigned x) {
   if (x == 0) return;
   else if (x == 1) { f(x-1); printf("1"); }
   else if (x == 2) { printf("2"); f(x-2); }
   else f(x - 3);
}
```

Zaokružiti sve repno rekurzivne pozive.

Pitanje 5.26. *Funkcija* `f` *je definisana sa:*

```
void f(unsigned x) {
   if (x == 0) return;
   else if (x == 1) f(x-1);
   else if (x == 2) { printf("2"); f(x-2); }
   else { f(x - 3); printf("3"); }
}
```

Zaokružiti sve repno rekurzivne pozive.

Pitanje 5.27. *Da li za bilo koju rekurzivnu funkciju postoji funkcija koja je njoj ekvivalentna a ne koristi rekurziju?*

Da li se rekurzija može eliminisati iz bilo koje repno rekurzivne funkcije?

Pitanje 5.28. *Napisati nerekurzivnu implementaciju funkcije:*

```
int f(int x) {
  if (p(x))
     return a(x)
  b(x);
  return f(c(x));
}
```

Pitanje 5.29. *Napisati nerekurzivnu implementaciju funkcije:*

```
void r(unsigned n) {
   if (n == 0) f(); else { p(x); r(n - 1); }
}
```

Pitanje 5.30. *Napisati nerekurzivnu implementaciju funkcije:*

```
void f(int x) {
  if (x <= 0)
     a(x);
  else {
     b(x); f(x - 1);
   }
}
```

Pitanje 5.31. *Napisati nerekurzivnu implementaciju funkcije:*

```
void a(unsigned k) {
   if (k < 2) return; else { b(k); a(k - 2); }
}
```

Pitanje 5.32. *Napisati nerekurzivnu implementaciju funkcije:*

```
void f(unsigned x) {
   if (x == 0) a(); else { g(x + 1); h(x); f(x - 1); }
}
```

Pitanje 5.33.

Napisati nerekurzivnu implementaciju funkcije:

```
void f(unsigned x) {
   if (x < 2) a(); else { g(x + 1); h(x+2); f(x - 2); }
}
```

Pitanje 5.34. *Napisati nerekurzivnu implementaciju funkcije:*

```
int f(int x, int a) {
   if (x == 0)
     return a;
   return f(x-1, x*a);
}
```

Pitanje 5.35. *Napisati nerekurzivnu implementaciju funkcije:*

```
int XYZ(int m, int n) {
  if (n == 0) return m;
  return XYZ(n, m % n);
}
```

Pitanje 5.36. *Napisati iterativnu verziju funkcije:*

```
void nalazenje_slozenog_broja(int n)
{
    if (!slozen(n))
    {
        printf("Broj %d je prost",n);
        nalazenje_slozenog_broja(n+1);
    }
    else
        printf("Broj %d je slozen",n);
}
```

Pitanje 5.37. *Napisati iterativnu verziju funkcije:*

```
void nalazenje_prostog_broja(int n)
{
    if (!prost(n))
    {
        printf("Broj %d je slozen",n);
        nalazenje_prostog_broja(n+1);
    }
    else
        printf("Broj %d je prost",n);
}
```

Pitanje 5.38. *Napisati iterativnu (nerekurzivnu) verziju sledeće funkcije:*

```
void test(int n,int k)
{
    if (k>1)
    {
        printf("\n %d",n+k);
        test(n,k-1)
    }
    else
        printf("\n %d",n*n);
}
```

Pitanje 5.39. *Napisati iterativnu (nerekurzivnu) verziju sledeće funkcije:*

```
void proba(int n,int k) {
    if (k==1)
        printf("\n %d",n*n);
    else
    {
        printf("\n %d",n+k);
        proba(n,k-1)
    }
}
```

Pitanje 5.40. *Šta su dobre a šta loše strane rekurzije?*

Pitanje 5.41. *Izabrati jedan od dva ponuđena odgovora u zagradi (iterativno ili rekurzivno):*

1. *Ukoliko se razmatra brzina izvršavanja obično su povoljnije (*iterativne/rekurzivne*) implementacije algoritma.*
2. *Ukoliko se razmatra razumljivost obično su povoljnije (*iterativne/rekurzivne*) implementacije algoritma.*
3. *Ukoliko se razmatra zauzeće memorije obično su povoljnije (*iterativne/rekurzivne*) implementacije algoritma.*

5.4.1 Zadaci

- Neka je, za zadati ceo broj n, n_1 proizvod cifara broja n, n_2 proizvod cifara broja n_1, ..., n_k proizvod cifara broja n_{k-1}, pri čemu je k najmanji prirodan broj za koji je n_k jednocifren. Na primer:
  - za $n = 10$, $n_1 = 1 * 0 = 0$, znači $k = 1$
  - za $n = 25$, $n_1 = 2 * 5 = 10$, $n_2 = 1 * 0 = 0$, znači $k = 2$
  - za $n = 39$, $n_1 = 3 * 9 = 27$, $n_2 = 2 * 7 = 14$, $n_3 = 1 * 4 = 4$, znači $k = 3$

  Napisati: (a) rekurzivnu; (b) iterativnu funkciju koja za dato n računa k. Zadatak rešiti bez korišćenja nizova.
- Neka je, za zadati ceo broj n, n_1 suma cifara broja n, n_2 suma cifara broja n_1, ..., n_k suma cifara broja n_{k-1}, pri čemu je k najmanji prirodan broj za koji je n_k jednocifren. Na primer:
  - za $n = 10$, $n_1 = 1 + 0 = 1$, znači $k = 1$
  - za $n = 39$, $n_1 = 3 + 9 = 12$, $n_2 = 1 + 2 = 3$, znači $k = 2$
  - za $n = 595$, $n_1 = 5 + 9 + 5 = 19$, $n_2 = 1 + 9 = 10$, $n_3 = 1 + 0 = 1$, znači $k = 3$

Napisati: (a) rekurzivnu; (b) iterativnu funkciju koja za dato n računa k. Zadatak rešiti bez korišćenja nizova.

- Napisati rekurzivnu i iterativnu funkciju koja za uneto n sa standardnog ulaza računa $f(n)$ ako je $f(n)$ definisan na sledeći način:

  $f(1) = 1$, $f(2) = 2$, $f(3) = 3$, $f(n+3) = f(n+2) + f(n+1) + f(n)$, za $n > 0$. Napisati i program koji poziva ove dve funkcije.

- Napisati rekurzivnu funkciju koja iz datog broja n izbacuje pojavljivanja svih parnih cifara koje su se nalaze na parnim mestima broja n, i svih neparnih cifara koje se nalaze na neparnim mestima broja n. Mesto cifre u broju čitati sdesna na levo, počev od indeksa 1. Odrediti vremensku složenost algoritma.

- Napisati rekurzivnu funkciju koja za zadato k i n crta "stepenice". Svaka stepenica je širine k, a ima n stepenika. Na primer k = 4, n = 3:

  ```
  ****
      ****
          ****
  ```

  Izračunati vremensku složenost algoritma.

- Dat je broj n i neka su $a_1, \ldots, a_k$ sleva na desno cifre broja n. Napisati rekurzivnu funkciju koja izračunava: $a_1 + 2 * a_2 + \ldots + k * a_k$. Izračunati vremensku složenost.

- Napisati rekurzivnu funkciju koja svaku parnu cifru c u broju n zamenjuje sa $c/2$. Napisati glavni program koji kao argument komandne linije dobija broj n, a na standardni izlaz ispisuje novi broj.

- Napisati rekurzivnu funkciju koja svaku parnu cifru c u broju n zamenjuje sa $c/2$. Napisati glavni program koji kao argument komandne linije dobija broj n, a na standardni izlaz ispisuje novi broj.

- Za zadati ceo broj n veći od 9 možemo primetiti da je suma cifara tog broja uvek manja od broja n. Neka je n_1 suma cifara broja n, neka je n_2 suma cifara broja n_1, n_3 suma cifara broja n_2, ..., n_{i+1} suma cifara broja n_i. Važi $n > n_1 > n_2 > \ldots > n_i$ dok god su brojevi n_i veći od 9, pa se u jednom trenutku sigurno dolazi do jednocifrenog broja n_k. Taj indeks k zavisi od početnog broja n.

  Na primer:

  - za $n = 10$, $n_1 = 1 + 0 = 1$, važi $k = 1$
  - za $n = 39$, $n_1 = 3 + 9 = 12$, $n_2 = 1 + 2 = 3$, važi $k = 2$
  - za $n = 595$, $n_1 = 5 + 9 + 5 = 19$, $n_2 = 1 + 9 = 10$, $n_3 = 1 + 0 = 1$, važi $k = 3$

  Napisati program koji za uneto n sa standardnog ulaza računa njemu odgovarajući broj k. Zadatak rešiti bez korišćenja nizova.

Zadatak 5.4.1. *1. Napisati rekurzivnu funkciju koja za dato n iscrtava trougao dimenzije n:*

```
(a)          (b)          (c)          (d)
+             +            +           +
++           +++          +++          ++
+++         +++++        +++++         +
```

2. Napisati rekurzivnu funkciju koja prikazuje sve cifre datog celog broja i to: (a) s leva na desno; (b) s desna na levo.

3. Napisati rekurzivnu funkciju koja određuje binarni (oktalni, heksadekadni) zapis datog celog broja.

4. Napisati rekurzivnu funkciju koja izračunava zbir cifara datog celog broja.

5. *Napisati rekurzivnu funkciju koja izračunava broj cifara datog celog broja.*
6. *Napisati rekurzivnu funkciju koja izračunava broj parnih cifara datog celog broja.*
7. *Napisati rekurzivnu funkciju koja izračunava najveću cifru datog broja.*
8. *Napisati rekurzivnu funkciju koja uklanja sva pojavljivanja date cifre iz datog broja.*
9. *Napisati rekurzivnu funkciju koja kreira niz cifara datog celog broja.*
10. *Napisati rekurzivnu funkciju koja obrće cifre datog celog broja.*
11. *Napisati rekurzivnu funkciju koja obrće niz brojeva.*
12. *Napisati rekurzivnu funkciju koja ispituje da li su elementi nekog niza brojeva poređani palindromski (isto od napred i od pozadi).*
13. *Napisati rekurzivnu funkciju koja obrće nisku.*
14. *Napisati rekurzivnu funkciju koja izbacuje sve parne cifre datog celog broja.*
15. *Napisati rekurzivnu funkciju koja iz datog broja izbacuje svaku drugu cifru.*
16. *Napisati rekurzivnu funkciju koja posle svake neparne cifre datog broja dodaje 0.*
17. *a) Napisati rekurzivnu funkciju koja štampa brojeve između* 0 *i* n*.*
    b) Napisati rekurzivnu funkciju koja štampa brojeve između n *i* 0*.*
18. *Napisati rekurzivnu funkciju koja izračunava sumu prvih* n *prirodnih brojeva.*
19. *Korišćenjem identiteta* $\sqrt{4 \cdot x} = 2 \cdot \sqrt{x}$ *napisati rekurzivnu funkciju koja izračunava ceo deo korena datog broja.*
20. *Napisati rekurzivnu funkciju koja izračunava vrednost binomnog koeficienta* $\binom{n}{k}$*.*
21. *Napisati rekurzivnu funkciju koja određuje maksimum niza celih brojeva.*
22. *Napisati rekurzivnu funkciju koja izračunava skalarni proizvod dva data vektora (predstavljena nizovima dužine* n*).*
23. *Napisati repno-rekurzivnu funkciju koja izračunava* $n!$*.*

Glava 6

Fundamentalni algoritmi

U ovoj glavi biće predstavljeni neki od osnovnih algoritama za pretraživanje, sortiranje i izračunavanje.

6.1 Pretraživanje

Pod pretraživanjem za dati niz elemenata podrazumevamo određivanje indeksa elementa niza koji je jednak datoj vrednosti ili ispunjava neko drugo zadato svojstvo (npr. najveći je element niza).

6.1.1 Linearno pretraživanje

Linearno (ili *sekvencijalno*) pretraživanje niza je pretraživanje zasnovano na ispitivanju redom svih elemenata niza ili ispitivanju redom elemenata niza sve dok se ne naiđe na traženi element (zadatu vrednost ili element koji ima neko specifično svojstvo, na primer — maksimum niza). Linearno pretraživanje je vremenske linearne složenosti po dužini niza koji se pretražuje. Ukoliko se u nizu od n elemenata traži element koji je jednak zadatoj vrednosti, u prosečnom slučaju (ako su elementi niza slučajno raspoređeni), ispituje se $n/2$, u najboljem 1, a u najgorem n elemenata niza.

Primer 6.1. *Naredna funkcija vraća indeks prvog pojavljivanja zadatog celog broja* `x` *u zadatom nizu* `a` *dužine* `n` *ili vrednost -1, ako se taj ne pojavljuje u* `a`*:*

```
int linearna_pretraga(int a[], int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

Složenost ove funkcije je $O(\mathtt{n})$ *a njena korektnost se dokazuje jednostavno (pod pretpostavkom da nenegativan broj* `n` *predstavlja broj elemenata niza* `a`*).*

Česta greška prilikom implementacije linearne pretrage je prerano vraćanje negativne vrednosti:

```
int linearna_pretraga(int a[], int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == x)
      return i;
    else
      return -1;
}
```

Navedena naredba `return` *prouzrokuje prekid rada funkcije tako da implementacija može da prekine petlju već nakon prve iteracije.*

Primer 6.2. *Linearna pretraga može biti realizovana i rekurzivno. U narednom kodu se poziv* `linearna_pretraga(a, i,` *nalazi indeks prvog pojavljivanja elementa* `x` *u nizu* `a[i, n-1]`*. Kako bi se izvršila pretraga celog niza, potrebno je izvršiti poziv* `linearna_pretraga(a, 0, n, x)`*, pri čemu je* `n` *dužina niza* `a`*.*

```
int linearna_pretraga(int a[], int i, int n, int x) {
  if (i == n)
     return -1;
  if (a[i] == x)
     return i;
  return linearna_pretraga(a, i+1, n, x);
}
```

Ukoliko se zadovoljimo pronalaženjem poslednjeg pojavljivanja elementa `x`*, kôd se može dodatno uprostiti.*

```
int linearna_pretraga(int a[], int n, int x) {
  if (n == 0)
     return -1;
  else if (a[n - 1] == x)
     return n-1;
  else return linearna_pretraga(a, n-1, x);
}
```

U oba slučaja rekurzivni pozivi su repno rekurzivni. Eliminacijom repne rekurzije u prvom slučaju se upravo dobija prikazana iterativna implementacija. Eliminacijom repne rekurzije u drugom slučaju se dobija:

```
int linearna_pretraga(int a[], int n, int x) {
  int i;
  for (i = n; i > 0; i--)
     if (a[i - 1] == x)
         return i - 1;
  return -1;
}
```

odnosno

```
...
  for (i = n-1; i >= 0; i--)
     if (a[i] == x)
         return i;
...
}
```

Primer 6.3. *Naredna funkcija vraća indeks poslednjeg pojavljivanja zadatog karaktera* `c` *u zadatoj niski* `s` *ili vrednost -1, ako se* `c` *ne pojavljuje u* `s`*:*

```
int string_last_char(char s[], char c) {
  int i;
  for (i = strlen(s)-1; i >= 0; i--)
    if (s[i] == c)
      return i;
  return -1;
}
```

Složenost ove funkcije je linearna po dužini niske `s` *a njena korektnost se dokazuje jednostavno.*

Primer 6.4. *Naredna funkcija vraća indeks najvećeg elementa među prvih* `n` *elemenata niza* `a` *(pri čemu je* `n` *veće ili jednako* 1*):*

```
int max(int a[], int n) {
   int i, index_max;
   index_max = 0;
   for(i = 1; i < n; i++)
     if (a[i] > a[index_max])
       index_max = i;
   return index_max;
}
```

Složenost ove funkcije je $O(\mathtt{n})$ a njena korektnost se dokazuje jednostavno.

6.1.2 Binarno pretraživanje

Binarno pretraživanje (ne nužno niza) je pronalaženje zadate vrednosti u zadatom skupu objekata koje pretpostavlja da su objekti zadatog skupa sortirani i u svakom koraku, sve dok se ne pronađe tražena vrednost, taj skup se deli na dva dela i pretraga se nastavlja samo u jednom delu — odbacuje se deo koji sigurno ne sadrži traženu vrednost. U diskretnom slučaju, ako skup koji se pretražuje ima n (konačno mnogo) elemenata, binarno pretraživanje je vremenski logaritamske složenosti — $O(\log n)$. Pretraživanje se u tom, diskretnom slučaju, vrši na sledeći način: pronalazi se srednji (po postojećem uređenju) element skupa (obično niza), proverava se da li je on jednak zadatoj vrednosti i ako jeste vraća se njegov indeks, a ako nije pretraživanje se nastavlja nad skupom svih manjih (ako je srednji element veći od zadate vrednosti) ili svih većih elemenata (ako je srednji element manji od zadate vrednosti). Binarno pretraživanje je primer *podeli-i-vladaj* (*divide and conquer*) algoritama.

Primer 6.5. *Binarno pretraživanje se može koristiti u igri pogađanja zamišljenog prirodnog broja iz zadatog intervala. Jedan igrač treba da zamisli jedan broj iz tog intervala, a drugi igrač treba da pogodi taj broj, na osnovu što manjeg broja pitanja na koje prvi igrač odgovara samo sa* da *ili* ne*. Ako pretpostavimo da interval čine brojevi od* 1 *do* 16 *i ako je prvi igrač zamislio broj* 11*, onda igra može da se odvija na sledeći način:*

Da li je zamišljeni broj veći od 8? da
Da li je zamišljeni broj veći od 12? ne
Da li je zamišljeni broj veći od 10? da
Da li je zamišljeni broj veći od 11? ne

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj 11*. Broj pitanja potrebnih za određivanje intervala pretrage je $O(\log k)$, gde je k širina polaznog intervala.*

Primer 6.6. *Ukoliko u prethodnoj igri nije zadata gornja granica intervala, najpre treba odrediti jedan broj koji je veći od zamišljenog broja i onda primeniti binarno pretraživanje. Ako pretpostavimo da je prvi igrač zamislio broj* 11*, onda igra može da se odvija na sledeći način:*

Da li je zamišljeni broj veći od 1? da
Da li je zamišljeni broj veći od 2? da
Da li je zamišljeni broj veći od 4? da
Da li je zamišljeni broj veći od 8? da
Da li je zamišljeni broj veći od 16? ne

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj u intervalu od 1 *do* 16 *i da primeni binarno pretraživanje na taj interval. Broj pitanja potrebnih za određivanje intervala pretrage je $O(\log k)$, gde je k zamišljeni broj i ukupna složenost pogađanja je ponovo $O(\log k)$.*

Binarno pretraživanje je daleko efikasnije nego linearno, ali zahteva da su podaci koji se pretražuju uređeni. Na primer, ovo je jedan od glavnih razloga zašto se reči u rečnicima, enciklopedijama, telefonskim imenicima i slično sortiraju. Ove knjige se obično pretražuju postupkom koji odgovara varijantama linearne pretrage[1]. Odnos složenosti, postaje još očigledniji ukoliko se zamisli koliko bi komplikovano bilo sekvencijalno pretraživanje reči u nesortiranom rečniku.

Binarno pretraživanje se može implementirati iterativno ili rekurzivno.

[1] Postupak se naziva *interpolaciona pretraga* i podrazumeva da se knjiga ne otvara uvek na sredini, već se tačka otvaranja određuje otprilike na osnovu položaja slova u abecedi (npr. ako se traži reč na slovo B knjiga se otvara mnogo bliže početku, a ako se traži reč na slovo U knjiga se otvara mnogo bliže kraju).

Primer 6.7. *Naredna funkcija daje rekuriznu implementaciju binarnog pretraživanja. Poziv* `binarna_pretraga(a, l, d` *vraća indeks elementa niza* `a` *između* `l` *i* `d` *(uključujući i njih) koji je jednak zadatoj vrednosti* `x` *ako takav postoji i* `-1` *inače. Dakle, ukoliko se želi pretraga celog niza, funkciju treba pozvati sa* `binarna_pretraga(a, 0, n-1, v)`.

```
int binarna_pretraga(int a[], int l, int d, int x) {
    int s;

    if (l > d)
        return -1;

    s = l + (d - l)/2;
    if (x > a[s])
       return binarna_pretraga(a, s+1, d, x);
    else if (x < a[s])
       return binarna_pretraga(a, l, s-1, x);
    else /* if (x == a[s]) */
       return s;
}
```

Primetimo da se za nalaženje središta, umesto izraza `l + (d - l)/2` *može koristiti i kraći izraz* `(l + d)/2`. *Ipak, upotreba prvog izraza je preporučena kako bi se smanjila mogućnost nastanka prekoračenja. Ovo jedan u nizu primera gde izrazi koji su matematički ekvivalentni, pokazuju različita svojstva u aritmetici fiksirane širine.*

Složenost ove funkcije je $O(\log \mathtt{n})$ *a njena korektnost se dokazuje jednostavno, pri čemu se pretpostavlja da je niz* `a` *sortiran.*

Primer 6.8. *Oba rekurzivna poziva u prethodnoj funkciji su bila repno-rekurzivna, tako da se mogu jednostavno eliminisati. Time se dobija iterativna funkcija koja vraća indeks elementa niza* `a` *koji je jednak zadatoj vrednosti* `x` *ako takva postoji i* `-1` *inače.*

```
int binarna_pretraga(int a[], int n, int x) {
   int l, d, s;

   l = 0; d = n-1;
   while(l <= d)  {
     s = l + (d - l)/2;
     if (x > a[s])
         l = s + 1;
     else if (x < a[s])
         d = s - 1;
     else /* if (x == a[s]) */
         return s;
   }
   return -1;
}
```

Sistemska implementacija binarnog pretraživanja

U standardnog biblioteci jezika C, postoji podrška za binarno pretraživanje u vidu funkcije `bsearch`, deklarisane u okviru zaglavlja `stdlib.h`. Implementacija ove funkcije je *generička*, jer se može koristiti za pretraživanje niza bilo kog tipa i bilo koje dužine i koristeći bilo koji uslov poređenja elemenata. Prototip funkcije `bsearch` je:

```
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *, const void *));
```

Argumenti funkcije imaju sledeću ulogu:

key je pokazivač na vrednost koja se traži; on je tipa `void*` kako bi mogao da prihvati pokazivač na bilo koji konkretan tip;

base je pokazivač na početak niza koji se pretražuje; on je tipa `void*` kako bi mogao da prihti pokazivač na bilo koji konkretan tip;

num je broj elemenata niza koji će biti ispitani;

size je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da prelazi sa jednog na sledeći element niza;

compare je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa; da bi funkcija imala isti prototip za sve tipove, njeni argumenti su tipa `void *` (zapravo — `const void *` jer vrednost na koju ukazuje ovaj pokazivač ne treba da bude menjana u okviru funkcije `compare`). Funkcija vraća jednu od sledećih vrednosti:

- < 0 ako je vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument;
- 0 ako su vrednosti na koju ukazuju prvi i drugi argument jednake;
- > 0 ako je vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument;

Funkcija `bsearch` vraća pokazivač na element koji je pronađen ili `NULL`, ako on nije pronađen. Ukoliko u nizu ima više elemenata koji su jednaki traženoj vrednosti, biće vraćen pokazivač na jednog od njih (standard ne propisuje na koji).

Opštost funkcije `bsearch` ima i svoju cenu: zbog neophodnog kastovanja pokazivača nema provere tipova, a za svaku proveru odnosa dve vrednosti poziva se druga funkcija (`compare`) što može da troši mnogo vremena i memorijskog prostora.

Primer 6.9. *Naredni program, koristeći sistemsku implementaciju binarne pretrage, traži zadati element u datom nizu celih brojeva.*

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

int poredi_dva_cela_broja(const void *px, const void *py) {
  return ( *(int*)px - *(int*)py );
}

int main () {
  int a[] = { 10, 20, 25, 40, 90, 100 };
  int *p;
  int trazena_vrednost = 40;
  p = (int*) bsearch (&trazena_vrednost, a, sizeof(a)/sizeof(int),
                      sizeof(int), poredi_dva_cela_broja);
  if (p!=NULL)
    printf("Trazena vrednost se nalazi u nizu.\n");
  else
    printf("Trazena vrednost se ne nalazi u nizu.\n");
  return 0;
```

Funkcija poređenja brojeva je mogla da bude implementirana i kao

```
int poredi_dva_cela_broja(const void *px, const void *py) {
  int x = *(int*)px, y = *(int*)py;
  if (x < y) return -1;
  else if (x > y) return 1;
  else /* if (x == y) */ return 0;
}
```

Na ovaj način, uklanja se mogućnost nastanka prekoračenja prilikom poređenja.

Određivanje nula funkcije

Jedan od numeričkih metoda za određivanje nula neprekidne realne funkcije zasniva se na metodu polovljenja intervala koji odgovara binarnoj pretrazi nad neprikidnim domenom. Pretpostavka metoda je da je funkcija neprekidna i da su poznate vrednosti a i b u kojima funkcija ima različit znak. U svakom koraku se tekući interval polovi i postupak se nastavlja nad jednom ili drugom polovinom u zavisnosti od toga da li je u središtu intervala funkcija pozitivna ili negativna. Metod uvek konvergira (kada su ispunjeni uslovi primene), tj. za bilo koju zadatu tačnost pronalazi odgovarajuće rešenje. U svakom koraku postupka se dužina intervala smanjuje dvostruko i ako je dužina početnog intervala jednaka d, onda nakon n interacija dužina tekućeg intervala jednaka $d/2^n$. Zato, ukoliko je nula funkcije tražena sa greškom manjom od ε, potrebno je napraviti $\log_2(d/\varepsilon)$ iteracija.

Primer 6.10. *Naredna funkcija pronalazi, za zadatu tačnost, nulu zadate funkcije* `f`*, na intervalu* [`l`, `d`]*, pretpostavljajući da su vrednosti funkcije* `f` *različitog znaka u* `l` *i* `d`*.*

```
#include <math.h>

float polovljenje(float (*f)(float), float l, float d, float epsilon) {
  float fl=(*f)(l), fd=(*f)(d);
  for (;;) {
    float s = (l+d)/2;
    float fs = (*f)(s);
    if (fs == 0.0 || d-l < epsilon)
      return s;
    if (fl*fs <= 0.0) {
      d=s; fd=fs;
    }
    else {
      l=s; fl=fs;
    }
  }
}
```

Pitanja i zadaci za vežbu

Pitanja

Pitanje 6.1. *Da bi se primenilo linearno pretraživanje niza, elementi niza (*izabrati sve ispravne odgovore*):*

(a) ...neophodno je da budu sortirani
(b) ...poželjno je da budu sortirani
(c) ...ne smeju da budu sortirani
(d) ...nepotrebno je da budu sortirani

Pitanje 6.2. *Da bi linearna pretraga bila primenljiva treba da važi (*izabrati sve ispravne odgovore*): (i) niz mora da bude uređen; (ii) niz mora da bude niz brojeva; (iii) nema preduslova.*

Pitanje 6.3. *Koja je, po analizi najgoreg slučaja, složenost algoritma za linearno pretraživanje niza?*

Pitanje 6.4. *Koja je složenost linearne pretrage niza od k celih brojeva čije vrednosti su između m i n?*

Pitanje 6.5. *Napisati funkciju za binarno pretraživanje niza celih brojeva.*

Pitanje 6.6. *Binarno pretraživanje niza radi*
(a) ...najbrže...
(b) ...najsporije...
(c) ...ispravno...
samo ako su elementi niza sortirani (zaokružiti jednu opciju).

Pitanje 6.7. *Da bi se primenilo binarno pretraživanje niza, elementi niza (*izabrati sve ispravne odgovore*):*
(a) ...neophodno je da budu sortirani
(b) ...poželjno je da budu sortirani
(c) ...ne smeju da budu sortirani
(d) ...nepotrebno je da budu sortirani

Pitanje 6.8. *Da bi binarna pretraga bila primenljiva treba da važi (*izabrati sve ispravne odgovore*): (i) niz mora da bude uređen; (ii) niz mora da bude niz brojeva; (iii) nema preduslova.*

Pitanje 6.9. *Da li je binarno pretraživanje moguće primeniti ako su elementi niza sortirani opadajuće?*

Pitanje 6.10. *Koja je, po analizi najgoreg slučaja, složenost algoritma za binarno pretraživanje niza?*

Pitanje 6.11. *Koja je složenost binarne pretrage niza od k celih brojeva čije vrednosti su između m i n?*

Pitanje 6.12. *Koja komanda treba da sledi iza komande* `if (data[mid] == value)` *u funkciji koja binarnom pretragom traži vrednost* `value` *u nizu* `data`*?*

Pitanje 6.13. *Ako je niz od 1024 elementa sortiran, onda binarno pretraživanje može da proveri da li se neka vrednost nalazi u nizu u (izabrati ispravan odgovor):*
(a) 10 koraka; (b) 128 koraka; (c) 512 koraka; (d) 1024 koraka;

Pitanje 6.14. *Nesortirani niz ima n elemenata. Potrebno je proveravati da li neka zadata vrednost postoji u nizu:*
(a) $O(1)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?
(b) $O(n)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?
(c) $O(n^2)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?

Pitanje 6.15. *Nesortirani niz ima n^2 elemenata. Potrebno je proveravati da li neka zadata vrednost postoji u nizu:*
(a) $O(1)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?
(b) $O(n)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?
(c) $O(n^2)$ puta: da li se tada više isplati primenjivati linearnu ili binarnu pretragu?

Pitanje 6.16. *Navesti prototip funkcije* `bsearch` *iz* `stdlib.h`*.*

Pitanje 6.17. *Dopuniti funkciju tako da može da se koristi kao poslednji element funkcije* `bsearch` *za pretragu niza brojeva tipa* `int`*:*

```
int poredi(const void* px, const void* py) {

}
```

Pitanje 6.18. *Koji uslov je dovoljan da bi metodom polovljenja intervala mogla da se aproksimira nula funkcije f, ako funkcija f ima različit znak na krajevima intervala?*

Pitanje 6.19. *Kako se jednostavno proverava da funkcija f ima različit znak na krajevima intervala $[a, b]$?*

Pitanje 6.20. *Metod polovljenja intervala koji traži nulu neprekidne funkcije f na zatvorenom intervalu $[a, b]$, ima smisla primenjivati ako je funkcija f neprekidna i važi*
(a) $f(a)f(b) < 0$; (b) $f(a)f(b) > 0$; (c) $f(a) + f(b) > 0$; (d) $f(a) + f(b) < 0$;

Pitanje 6.21. *Kada se zaustavlja numeričko određivanje nule funkcije metodom polovljenja intervala?*

Pitanje 6.22. *Napisati prototip funkcije koja odredjuje nulu zadate funkcije na intervalu* `[a,b]` *(jedan od argumenata treba da bude pokazivač na funkciju).*

Zadaci

Zadatak 6.1.1. *Napisati funkciju* `float f5(float a, float b, float eps)` *koja računa nulu funkcije* $f(x) = 5 * sin(x) * ln(x)$ *na intervalu* `(a, b)` *sa tačnošću* `eps`. *Brojevi* `a`, `b` *i* `eps` *unose se sa standardnog ulaza.*
Napomena1*: koristiti algoritam binarne pretrage*
Napomena2*: u* `math.h` *nalaze se* `float sin(float)` *za računanje sinusa i* `float log(float x)` *za računanje prirodnog logaritma. Testirati funkciju pozivom u main-u.*

Zadatak 6.1.2. *Kao argumenti komandne linije dati su celi brojevi* a, b, c, d, e. *Binarnom pretragom naći ceo broj na intervalu* $[d, e]$ *nulu funkcije* $a \cdot x^2 + b \cdot x + c$ *sa tačnošću* `0.0001`. *U slučaju greške na standardni izlaz napisati* `-1`. *Ako su argumenti komandne linije ispravno dati podrazumevati da funkcija ima tačno jednu nulu na datom intervalu. Na standardni izlaz ispisati nađenu vrednost. Broj ispisati sa 3 decimale.*

Zadatak 6.1.3. *Napisati funkciju za određivanje, metodom polovljenja, nule funkcije* f *na intervalu* $[a, b]$ *takve da su vrednosti* $f(a)$ *i* $f(b)$ *različitog znaka.*

Zadatak 6.1.4. *Napisati funkciju koja u uređenom nizu niski pronalazi indeks zadatke niske (a vraća -1 ako takva niska nije pronađena). Prototip funkcije treba da bude:* `int pronadji_nisku(char **niz, char* niska, int`

Zadatak 6.1.5. *Napisati funkciju sa prototipom*

```
double treci_koren(double x, double a, double b, double eps);
```

koja, metodom polovljenja intervala, računa treći koren zadatog broja x *(*$x \geq 1$*), tj. za dato* x *određuje broj* k *za koji važi* $k^3 = x$*, sa tačnošću* `eps` *i sa početnom pretpostavkom da se broj* k *nalazi u datom intervalu* $[a, b]$. *Napisati pratећi program koji omogućava korisniku da unese broj* x *i traženu tačnost. Korisnik ne unosi početnu procenu intervala, a za početni interval može se uzeti* $[0, x]$. *Ispisati poruku o grešci ako je uneto* x *manje od* 0.

6.2 Sortiranje

Sortiranje je jedan od fundamentalnih zadataka u računarstvu. Sortiranje podrazumeva uređivanje niza u odnosu na neko linearno uređenje (npr. uređenje niza brojeva po veličini — rastuće ili opadajuće, uređivanje niza niski leksikografski ili po dužini, uređivanje niza struktura na osnovu vrednosti nekog polja i slično). Mnogi zadaci nad nizovima se mogu jednostavnije rešiti u slučaju da je niz sortiran (npr. pretraživanje se može vršiti binarnom pretragom).

Postoji više različitih algoritama za sortiranje nizova. Neki algoritmi su jednostavni i intuitivni, dok su neki kompleksniji, ali izuzetno efikasni. Najčešće korišćeni algoritmi za sortiranje su:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort
- Heap sort

Neki od algoritama za sortiranje rade u mestu (eng. in-place), tj. sortiraju zadate elemente bez korišćenja dodatnog niza. Drugi algoritmi zahtevaju korišćenje pomoćnog niza ili nekih drugih struktura podataka.

Prilikom sortiranja nizova koji sadrže podatke netrivijalnih tipova najskuplje operacije su operacija poređenja dva elementa niza i operacija razmene dva elemente niza. Zbog toga se prilikom izračunavanja složenosti algoritama obično u obzir uzimaju samo ove operacije.

Algoritmi za sortiranje obično pripadaju jednoj od dve klase vremenske složenosti. Jednostavniji, sporiji algoritmi sortiranja imaju složenost najgoreg slučaja $O(n^2)$ i u tu grupu spadaju *bubble*, *insertion* i *selection*. *Shell sort* je algoritam koji je negde između pomenute dve klase (u zavisnosti od implementacije složenost mu varira od $O(n^2)$ do $O(n \log^2 n)$). Kompleksniji, brži algoritmi imaju složenost

najgoreg slučaja $O(n \log n)$ i u tu grupu spadaju *heap* i *merge sort*. Algoritam *quick sort* ima složenost najgoreg slučaja $O(n^2)$, ali, pošto je složenost prosečnog slučaja $O(n \log n)$ i pošto u praksi pokazuje dobre rezultate, ovaj se algoritam ubraja u grupu veoma brzih algoritama.

Može se dokazati da algoritmi koji sortiraju niz permutovanjem elemenata niza (većina nabrojanih funkcioniše upravo ovako) ne mogu imati bolju složenost od $O(n \log n)$.

Svi algoritmi sortiranja u nastavku teksta će biti prikazani na standardnom primeru sortiranja niza celih brojeva u neopadajućem poretku. Pre opisa algoritama sortiranja navedimo funkciju koja proverava da li je niz već sortiran.

```
int sortiran(int a[], int n) {
  int i;
  for (i = 0; i < n - 1; i++)
      if (a[i] > a[i + 1])
         return 0;
  return 1;
}
```

U većini algoritama sortiranja, osnovi korak će biti razmena elemenata niza, te će se pretpostavljati da je na raspolaganju sledeća funkcija:

```
void razmeni(int a[], int i, int j) {
  int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

Opšti oblik funkcije za sortiranje niza brojeva je

```
void sort(int a[], int n);
```

Preduslov funkcije je da nenegativna promenljiva `n` sadrži dimenziju niza `a`[2], dok je postuslov da su elementi niza[3] nakon primene funkcije sortirani kao i da se (mulit)skup elemenata niza nije promenio (ova invarijanta je trivijalno ispunjena kod svih algoritama zasnovanih na razmenama elemenata niza).

6.2.1 *Bubble sort*

Bubble sort algoritam u svakom prolazu kroz niz poredi uzastopne elemente, i razmenjuje im mesta ukoliko su u pogrešnom poretku. Prolasci kroz niz se ponavljaju sve dok se ne napravi prolaz u kome nije bilo razmena, što znači da je niz sortiran.

Primer 6.11. *Prikažimo rad algoritma na primeru sortiranja niza* (6 1 4 3 9)*:*

Prvi prolaz:
(**6 1** 4 3 9) → (**1 6** 4 3 9), razmena jer je $6 > 1$
(1 **6 4** 3 9) → (1 **4 6** 3 9), razmena jer je $6 > 4$
(1 4 **6 3** 9) → (1 4 **3 6** 9), razmena jer je $6 > 3$
(1 4 3 **6 9**) → (1 4 3 **6 9**)

Drugi prolaz:
(**1 4** 3 6 9) → (**1 4** 3 6 9)
(1 **4 3** 6 9) → (1 **3 4** 6 9), razmena jer $4 > 3$
(1 3 **4 6** 9) → (1 3 **4 6** 9)
(1 3 4 **6 9**) → (1 3 4 **6 9**)

Treći prolaz:
(**1 3** 4 6 9) → (**1 3** 4 6 9)
(1 **3 4** 6 9) → (1 **3 4** 6 9)
(1 3 **4 6** 9) → (1 3 **4 6** 9)
(1 3 4 **6 9**) → (1 3 4 **6 9**)

Primetimo da je niz bio sortiran već nakon drugog prolaza, međutim, da bi se to utvrdilo, potrebno je bilo napraviti još jedan prolaz.

[2]Preciznije, preduslov je da je `n` manje ili jednako od alociranog broja elemenata niza. Sve funkcije u nastavku ovog poglavlja će deliti ovaj ili neki sličan preduslov koji obezbeđuje da tokom sortiranja ne dolazi do pristupa nedozvoljenoj memoriji.

[3]Preciznije, elementi između pozicija `0` i `n-1` su sortirani, što je zaista ceo niz ukoliko `n` sadrži dimenziju niza.

Naredna funkcija *bubble sort* algoritmom sortira niz `a`, dužine `n`.

```
void bubblesort(int a[], int n) {
  int bilo_razmena, i;
  do {
    bilo_razmena = 0;
    for (i = 0; i < n - 1; i++)
       if (a[i] > a[i + 1]) {
          razmeni(a, i, i+1);
          bilo_razmena = 1;
       }
  } while (bilo_razmena);
}
```

Uslov koji obezbeđuje parcijalnu korektnost je da ako `bilo_razmena` ima vrednost 0 pre provere uslova izlaska iz spoljašnje petlje, onda je niz `a` sortiran (tj. njegov početni deo dužine `n`). Zaista, kada se spoljašnja petlja završi, vrednost promenljive `bilo_razmena` je 0, te prethodna implikacija obezbeđuje korektnost. Invarijanta unutrašnje petlje koja obezbeđuje pomenuti uslov je da ako promenljiva `bilo_razmena` ima vrednost 0, onda je deo niza `a[0, i]` sortiran. Zaista, po završetku unutrašnje petlje `i` ima vrednost `n-1`, te je pomenuti uslov obezbeđen. Pošto je algoritam zasnovan na razmenama, invarijanta algoritma je i da se (multi)skup elemenata niza ne menja tokom njegovog izvršavanja.

Svojstvo algoritma koje obezbeđuje zaustavljanje je da se nakon svake iteracije spoljašnje petlje sledeći najveći elemenat koji nije već bio na svojoj poziciji dolazi na nju. *Bubble sort* je na osnovu ovog svojstva i dobio ime (jer veliki elementi kao mehurići „isplivavaju" ka kraju niza). Ovo sa jedne strane obezbeđuje zaustavljanje algoritma, dok se sa druge strane može iskoristiti za optimizaciju. Ovom optimizacijom se može smanjiti broj poređenja, ali ne i broj razmena.

```
void bubblesort(int a[], int n) {
  do {
    int i;
    for (i = 0; i < n - 1; i++)
       if (a[i] > a[i + 1])
          razmeni(a, i, i+1);
    n--;
  } while (n > 1);
}
```

Isti algoritam, implementiran na stilski malo drugačiji način (koji se može ponekad sresti u literaturi).

```
void bubblesort(int a[], int n) {
  int i, j;
  for (i = n - 1; i > 0; i--)
    for (j = 0; j < i; j++)
       if (a[j] > a[j + 1])
          razmeni(a, j, j+1);
}
```

Optimizacija može otići i korak dalje, ukoliko se dužina ne smanjuje samo za 1, već ukoliko se posmatra koliko je zaista elemenata na kraju niza već postavljeno na svoje mesto (ovo se može odrediti na osnovu mesta gde se dogodila poslednja zamena).

```
int bubblesort(int a[], int n) {
  do {
    int nn = 1, i;
    for (i = 0; i < n - 1; i++)
       if (a[i] > a[i + 1]) {
          razmeni(a, i, i+1);
          nn = i + 1;
       }
```

```
    n = nn;
  } while (n > 1);
}
```

U ovom slučaju, uslov koji važi pre provere uslova spoljašnje petlje je da su elementi od pozicije `n-1` do kraja niza sortirani. Ovo obezbeđuje invarijanta unutrašnje petlje koja obezbeđuje da su svi elementi `a[nn-1, i]` sortirani.

Što se efikasnosti tiče, najgori slučaj nastupa kada su elementi početnog niza sortirani u opadajućem poretku. U tom slučaju vrši se n prolaska kroz niz, a u svakom prolasku se pravi n poređenja i n razmena, što ukupno daje $O(n^2)$ poređenja i $O(n^2)$ razmena. Primetimo da se optimizacijom broj poređenja smanjuje oko dva puta.

Bubble sort algoritam se smatra veoma lošim algoritmom. Neki autori čak zagovaraju tezu da bi ga trebalo potpuno izbaciti iz nastave računarstva. U šali se čak navodi kako bi jedini algoritam gori od njega bio algoritam koji permutuje algoritam na slučajan način sve dok niz slučajno ne postane sortiran.

6.2.2 *Selection sort*

Selection sort je algoritam koji se može u jednoj rečenici opisati sa: „Ako niz ima više od jednog elementa, zameni početni element sa najmanjim elementom niza i zatim rekurzivno sortiraj rep (elemente iza početnog)". U iterativnoj implementaciji, niz se sortira tako što se u svakoj iteraciji na svoju poziciju dovodi sledeći po redu element niza, tj. u i-toj iteraciji se i-ti po redu element dovodi na poziciju i. Ovo se može realizovati tako što se pronađe pozicija m najmanjeg elementa od pozicije i do kraja niza i zatim se razmene element na poziciji i i element na poziciji m. Algoritam se zaustavlja kada se pretposlednji po veličini element dovede na pretposlednju poziciju u nizu.

Primer 6.12. *Prikažimo rad algoritma na primeru sortiranja niza* (5 3 4 2 1).
(.**5** 3 4 2 *1*), $i = 0$, $m = 4$, razmena elemenata 5 i 1
(1 .**3** 4 *2* 5), $i = 1$, $m = 3$, razmena elemenata 3 i 2
(1 2 .**4** *3* 5), $i = 2$, $m = 3$, razmena elemenata 4 i 3
(1 2 3 .***4*** 5), $i = 3$, $m = 3$, razmena elemenata 4 i 4
(1 2 3 4 .5)

Pozicija najmanjeg elementa u nizu `a`, dužine `n`, počevši od pozicije `i` se može naći narednom funkcijom.

```
int poz_min(int a[], int n, int i) {
  int m = i, j;
  for (j = i + 1; j < n; j++)
     if (a[j] < a[m])
        m = j;
  return m;
}
```

U tom slučaju, *selection sort* algoritam izgleda ovako:

```
int selectionsort(int a[], int n) {
  int i;
  for (i = 0; i < n - 1; i++)
     razmeni(a, i, poz_min(a, n, i));
}
```

Invarijanta petlje je da su elementi niza `a[0, i]` sortirani, kao i da su svi oni manji od svih elemenata niza `a[i+1, n-1]` (eventualno je, poslednji, najveći element prvog dela jednak najmanjem elementu drugog dela). Pošto je algoritam zasnovan na razmenama, (multi)skup elemenata polaznog niza se (trivijalno) ne menja. Zaustavljanje je, takođe, trivijalno.

Primetimo da je broj razmena jednak $n - 1$, tj. $O(n)$. Međutim, broj poređenja je $O(n^2)$. Zaista, broj poređenja koja se izvrše u okviru funkcije `poz_min` jednak je `n-i`. Tako da je ukupan broj poređenja $\sum_{i=0}^{n-1} n - i = \sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2}$.

Ukoliko se ne koriste pomoćna funkcije `poz_min`, algoritam se može implementirati na sledeći način.

```
int selectionsort(int a[], int n) {
  int i;
  for (i = 0; i < n - 1; i++) {
     int m = i, j;
     for (j = i + 1; j < n; j++)
        if (a[j] < a[m])
          m = j;
     razmeni(a, i, m);
  }
}
```

Ponekad se sreće i naredna implementacija.

```
int selectionsort(int a[], int n) {
  int i, j;
  for (i = 0; i < n; i++)
     for (j = i + 1; j < n; j++)
         if (a[i] < a[j])
            razmeni(a, i, j);
}
```

Napomenimo da je ova implementacija znatno neefikasnija od prethodne (iako je kôd kraći) jer se u najgorem slučaju osim $O(n^2)$ poređenja vrši i $O(n^2)$ razmena. Zbog toga bi, naravno, ovaj način implementacije algoritma trebalo izbegavati.

Rekruzivna implementacija je donekle jednostavnija u slučaju da se umesto dovođenja najmanjeg elementa na početak vrši dovođenje najvećeg elementa na kraj.

```
int poz_max(int a[], int n) {
  if (n == 1)
     return 0;
  else {
     int m = poz_max(a, n-1);
     return a[m] > a[n-1] ? m : n-1;
  }
}

void selectionsort(int a[], int n) {
  if (n > 1) {
     razmeni(a, n-1, poz_max(a, n));
     selectionsort(a, n-1);
  }
}
```

Naravno, moguća je i originalna varijanta, uz slanje dodatnog indeksa kroz rekurzivne pozive.

```
int poz_min(int a[], int n, int i) {
  if (i == n-1)
     return n-1;
  else {
     int m = poz_min(a, n, i+1);
     return a[m] < a[i] ? m : i;
  }
}

void selectionsort(int a[], int n, int i) {
  if (i < n - 1) {
     razmeni(a, i, poz_min(a, n, i));
     selectionsort(a, n, i+1);
```

```
  }
}
```

Početni poziv je u tom slučaju `selectionsort(a, n, 0)`.

Naglasimo da se korišćenjem naprednijih struktura podataka, ideje *selection sort* algoritma mogu iskoristiti da se dobije algoritam složenosti $O(n \log n)$ (tzv. *heap sort* algoritam koji koristi strukturu podataka poznatu kao hip (eng. heap)).

6.2.3 *Insertion sort*

Insertion sort algoritam sortira niz tako što uzima jedan po jedan element niza i umeće ga na odgovarajuće mesto u sotrirani niz koji čuva krajnji rezultat. Konceptualno gledano, postoje dva niza — polazni niz iz kojeg se uklanjaju elementi i niz koji čuva rezultat i u koji se dodaju elementi. Međutim, obično implementacije koriste memorijski prostor polaznog niza za obe uloge — početni deo niza predstavlja rezultujući niz, dok krajnji deo predstavlja preostali deo polaznog niza.

Primer 6.13. *Prikažimo rad algoritma na primeru sortiranja niza* 5 3 4 1 2.
5. 3 4 1 2
3 5. 4 1 2
3 **4** 5. 1 2
1 3 4 5. 2
1 **2** 3 4 5.
Podebljanim slovima su prikazani elementi umetnuti na svoju poziciju.

Dakle, *insertion sort* se može formulisati na sledeći način: „Ako niz ima više od jednog elementa, sortiraj rekurzivno sve elemente ispred poslednjeg, a zatim umetni poslednji u već sortirani prefiks." Ovim se dobija sledeća (rekurzivna) implementacija.

```
void insertionsort(int a[], int n) {
  if (n > 1) {
    insertionsort(a, n-1);
    umetni(a, n-1);
  }
}
```

Kada se eliminiše rekurzija, dolazi se do sledeće iterativne implementacije.

```
void insertionsort(int a[], int n) {
  int i;
  for (i = 1; i < n; i++)
     umetni(a, i);
}
```

Invarijanta petlje je da je deo niza `a[0, i-1]` sortiran, kao i da se (multi)skup elemenata u nizu `a` ne menja. Kako bi se obezbedila korektnost, preduslov funkcije `umetni` je da je sortiran deo niza `a[0, i-1]`, dok ona treba da obezbedi postuslov da je sortiran deo niza `a[0, i]`.

Funkcija `umetni` može biti implementirana na različite načine.

Jedna od mogućnosti je da se element menja sa svojim prethodnikom sve dok je prethodnik veći od njega.

```
void umetni(int a[], int i) {
  int j;
  for(j = i; j > 0 && a[j] > a[j-1]; j--)
      razmeni(a, j, j-1);
}
```

Prikažimo još i odgovarajuću rekurzivnu implementaciju.

```
void umetni(int a[], int j) {
  if (j > 0 && a[j] < a[j-1]) {
       razmeni(a, j, j-1);
       umetni(a, j-1);
  }
}
```

Funkcija `umetni` se poziva $n-1$ put i to za vrednosti i od 1 do n, dok u najgorem slučaju (obrnuto sortiranog niza) ona izvršava i razmena i i poređenja. Zbog toga je ukupan broj razmena, kao i broj poređenja $O(n^2)$ ($\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}$).

Efikasnija implementacija se može dobiti ukoliko se ne koriste razmene, već se element koji nije na svom mestu zapamti, zatim se pronađe pozicija na koju treba da se umetne, svi elementi od te pozicije se pomere za jedno mesto u desno da bi se na kraju zapamćeni element upisao na svoje mesto.

```
void umetni(int a[], int i) {
  int j, tmp = a[i];
  for (j = i; j > 0 && a[j-1] > tmp; j--)
      a[j] = a[j-1];
  a[j] = tmp;
}
```

S obzirom da se prilikom razmena koriste tri dodele, ovim se broj dodela smanjuje 3 puta (broj poređenja ostaje nepromenjen).

Prikažimo i verziju `insertionsort` funkcije u kojoj je kod pomoćne funkcije integrisan.

```
void insertionsort(int a[], int n) {
  int i;
  for (i = 1; i < n; i++) {
    int j, tmp = a[i];
    for (j = i; j > 0 && a[j-1] > tmp; j--)
        a[j] = a[j-1];
    a[j] = tmp;
  }
}
```

Broj poređenja se može smanjiti ukoliko se za pronalaženje ispravne pozicije elementa u sortiranom prefiksu umesto linearne pretrage koristi binarna. Međutim, broj dodela nije moguće smanjiti te se ukupno vreme neće značajno smanjiti. Zbog toga, nećemo prikazati implementaciju ovakvog rešenja.

Naglasimo da se korišćenjem naprednijih struktura podataka, ideje *insertion sort* algoritma mogu iskoristiti da se dobije algoritam složenosti $O(n \log n)$ (tzv. *tree sort* algoritam koji koristi binarna stabla).

6.2.4 *Shell sort*

Najveći uzrok neefikasnosti kod *insertion sort* algoritma je slučaj malih elemenata koji se nalaze blizu kraja niza. Pošto se nalaze blizu kraja, oni se umeću u relativno dugačak niz, a pošto su mali umeću se na početak te je potrebno izvršiti pomeranje velikog broja elemenata kako bi se oni postavili na svoje mesto. *Shell sort*[4] popravlja ovo. Osnovni cilj je da se „skrati put" ovakvih elemenata. *Shell sort* koristi činjenicu da *insertion sort* funkcioniše odlično kod nizova koji su „skoro sortirani". Algoritam radi tako što se niz deli na veći broj kratkih kolona koje se sortiraju primenom *insertion sort* algoritma, čime se omogućava direktna razmena udaljenih elemenata. Broj kolona se zatim smanjuje, sve dok se na kraju *insertion sort* ne primeni na ceo niz. Međutim, do tada su „pripremni koraci" deljenja na kolone doveli niz u „skoro sortirano" stanje te se završni korak prilično brzo odvija.

Primer 6.14. *Ilustrujmo jednu varijantu* Shell sort *algoritma na primeru sortiranja niza* (9, 10, 16, 8, 5, 11, 1, 12, 4, 6, 13, 7, 14, 3, 15, 2).

U prvoj fazi podelimo niz na 8 kolona.

[4] Nazvan po D.L.Šelu koji ga je prvi opisao 1959. godine.

9, 10, 16, 8, 5, 11, 1, 12,
4, 6, 13, 7, 14, 3, 15, 2

i primenimo insertion sort *na sortiranje svake kolone ponaosob.*

4, 6, 13, 7, 5, 3, 1, 2,
9, 10, 16, 8, 14, 11, 15, 12

Ovim je dobijen niz (4, 6, 13, 7, 5, 3, 1, 2, 9, 10, 16, 8, 14, 11, 15, 12). *Primetimo da je ovim veliki broj malih elemenata (npr. 2) kroz samo jednu operaciju razmene došao u prvu polovinu niza.*

U sledećoj fazi delimo niz na 4 kolone

4, 6, 13, 7,
5, 3, 1, 2,
9, 10, 16, 8,
14, 11, 15, 12

i primenimo insertion sort *na sortiranje svake kolone ponaosob.*

4, 3, 1, 2,
5, 6, 13, 7,
9, 10, 15, 8,
14, 11, 16, 12

Ovim je dobijen niz (4 3 1 2 5 6 13 7 9 10 15 8 14 11 16 12).

U sledećoj fazi delimo niz na dve kolone.

4, 3,
1, 2,
5, 6,
13, 7,
9, 10,
15, 8,
14, 11,
16, 12

i primenimo insertion sort *na sortiranje svake kolone ponaosob.*

1, 2,
4, 3,
5, 6,
9, 7,
13, 8,
14, 10,
15, 11,
16, 12

Ovim se dobija niz (1, 2, 4, 3, 5, 6, 9, 7, 13, 8, 14, 10, 15, 11, 16, 12).

Na kraju se dobijeni sortira primenom insertion sort *algoritma, čime se dobija niz* (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16).

Napomenimo da je podela na kolone samo fiktivna operacija i da se ona u implementaciji izvodi tako što se prilikom umetanja elementa ne razmatraju susedni elementi već elementi na rastojanju `gap` gde `gap` označava tekući broj kolona.

```
void shellsort(int a[], int n) {
  int gap, i, j;
  for (gap = n/2; gap > 0; gap /= 2)
     for (i = gap; i < n; i++)
        for (j=i-gap; j>=0 && a[j]>a[j+gap]; j-=gap)
           razmeni(a, j, j + gap);
}
```

S obzirom da je u poslednjoj iteraciji spoljne petlje `gap` ima vrednost 1, algoritam u poslednjoj iteraciji izvodi običan *insertion sort* algoritam, te se korektnost *Shell sort* algoritma oslanja na već diskutovanu korektnost *insertion sort* algoritma i invarijante da je sve vreme (multi)skup elemenata niza nepromenjen (koja je trivijalno ispunjena jer se algoritam zasniva na razmenama).

Ono što se može proizvoljno određivati je broj kolona na koji se vrši deljenje niza u fazama (broj kolona se obično označava sa `gap` i ovaj niz se u literaturi često označava `gap sequence`). Originalni Šelov

predlog (koji je i korišćen u prethodnom primeru i implementaciji je $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, \lfloor n/2^k \rfloor, \ldots, 1$). Kako bi se garantovala korektnost, na kraju je potrebno primeniti *insertion sort* bez podele na kolone (tj. poslednji član sekvence mora biti 1). U zavisnosti od sekvence varira i složenost najgoreg slučaja. Originalna sekvenca ima složenost $O(n^2)$ dok postoje druge sekvence koje garantuju složenost $O(n^{\frac{3}{2}})$, $O(n^{\frac{4}{3}})$ pa i $O(n \log^2 n)$. Napomenimo da je empirijski utvrđeno da postoje relativno nepravilne sekvence koje u praksi daju bolje rezultate (imaju najbolju složenost prosečnog slučaja).

6.2.5 *Merge sort*

Dva već sortirana niza se mogu objediniti u treći sortirani niz samo jednim prolaskom kroz nizove (tj. u linearno vremenu $O(m+n)$ gde su m i n dimenzije polaznih nizova).

```
void merge(int a[], int m, int b[], int n, int c[]) {
  int i, j, k;
  i = 0, j = 0, k = 0;
  while (i < m && j < n)
     c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while(i < m) c[k++] = a[i++];
  while(j < n) c[k++] = b[j++];
}
```

U prikaznoj implementaciji, paralelno se prolazi kroz nizove `a` dimenzije `m` i `b` dimenzije `n`. Promenljiva `i` čuva tekuću pozicija u nizu `a`, dok promenljiva `j` čuva tekuću poziciju u nizu `b`. Tekući elementi se porede i manji se upisuje u niz `c` (na tekuću poziciju `k`), pri čemu se napreduje samo u nizu iz koga je taj manji element uzet. Prolazak se ne stigne do kraja jednog od nizova. Kada se kraći niz isprazni, eventualni preostali elementi iz dužeg niza se nadovezuju na kraj niza `c`.

Merge sort algoritam deli niz na dve polovine (čija se dužina razlikuje najviše za 1), rekurzivno sortira svaku od njih, i zatim objedinjuje sortirane polovine. Problematično je što je za objedinjavanje neophodno koristiti dodatni niz pomoćni niz. Na kraju se izvršava vraćanje objedinjenog niza iz pomoćnog u polazni. Izlaz iz rekurzije je slučaj jednočlanog niza (slučaj praznog niza ne može da nastupi).

Funkicija `mergesort_` *merge sort* algoritmom sortira deo niza `a[l, d]`, uz korišćenje niza `tmp` kao pomoćnog.

```
void mergesort_(int a[], int l, int d, int tmp[]) {
  if (l < d) {
    int i, j;
    int n = d - l + 1, s = l + n/2;
    int n1 = n/2, n2 = n - n/2;

    mergesort_(a, l, s-1, tmp);
    mergesort_(a, s, d, tmp);
    merge(a + l, n1, a + s, n2, tmp);

    for (i = l, j = 0; i <= d; i++, j++)
      a[i] = tmp[j];
  }
}
```

Promenljiva `n` čuva broj elemenata koji se sortiraju u okviru ovog rekurzivnog poziva, a promenljiva `s` čuva središnji indeks u nizu između `l` i `d`. Rekurzivno se sortira `n1 = n/2` elemenata između pozicija `l` i `s-1` i `n2 = n - n/2` elemenata između pozicija `s` i `d`. Nakon toga, sortirani podnizovi se objedinjuju u pomoćni niz `tmp`. Primetimo na ovom mestu korišćenje pokazivačke aritmetike. Adresa početka prvog sortiranog podniza koji se objedinjava je `a+l`, dok je adresa početka drugog `a + s`.

Pomoćni niz se može pre početka sortiranja dinamički alocirati i koristiti kroz rekurzivne pozive.

```
void mergesort(int a[], int n) {
  int* tmp = (int*)malloc(n*sizeof(int));
  mergesort_(a, 0, n-1, tmp);
```

```
  free(tmp);
}
```

Provera da li je je alokacija nije uspela ovaj put nije vršena.

Rekurentna jednačina koja opisuje složenost je $T(n) = 2 \cdot T(n/2) + O(n)$ te je složenost $O(n \log n)$.

6.2.6 *Quick sort*

Osnovna ideja kod *selection sort* algoritma je da se jedan element postavi na svoje mesto, a zatim da se isti metod rekurzivno primeni na niz koji je za jedan kraći od polaznog. S obzirom da je pripremna akcija zahtevala $O(n)$ operacija, dobijena je jednačina $T(n) = T(n-1) + O(n)$, čije rešenje je $O(n^2)$. Sa druge strane, kod *merge sort* algoritma sortiranje se svodi na sortiranje dva podniza polaznog niza dvostruko manje dimenzije. S obzirom da korak objedinjavanja dva sortirana niza zahteva $O(n)$ operacija, dobija se jednačina $T(n) = 2T(n/2) + O(n)$, čije je rešenje $O(n \log n)$. Dakle, značajno je efikasnije da se problem dimenzije n svodi na dva problema dimenzije `n/2` nego na jedan problem dimenzije `n-1` — ovo je osnovna ideja tzv. *podeli i vladaj (divide-and-conquer)* algoritama u koje spada i *quick sort*.

Quick sort algoritam pokušava da postigne bolju efikasnost, modifikujući osnovnu ideju *selection sort* algoritma tako što umesto minimuma (ili maksimuma), u svakom koraku na svoje mesto dovede neki element (obično nazivan *pivot*) koji je relativno blizu sredine niza. Međutim, da bi nakon toga, problem mogao biti sveden na sortiranje dva dvostruko manja podniza, potrebno je prilikom dovođenja pivota na svoje mesto grupisati sve elemente manje od njega levo od njega, a sve elemente veće od njega desno od njega. Dakle, ključni korak *quick sort* je tzv. korak *particionisanja* koji nakon izbora nekog pivotirajućeg elementa podrazumeva da se niz organizuje da prvo sadrži elemente manje od pivota, zatim pivotirajući element, i na kraju elemente veće od pivota.

Qsort algoritam algoritam se može implementirati na sledeći način. Poziv `qsort_(a, l, d)` sortira deo niza `a[l, d]`.

```
void qsort_(int a[], int l, int d) {
  if (l < d) {
    razmeni(a, l, izbor_pivota(a, l, d));
    int p = particionisanje(a, l, d);
    qsort_(a, l, p - 1);
    qsort_(a, p + 1, d);
  }
}
```

Funkcija `qsort` se onda jednostavno implementira

```
void qsort(int a[], int n) {
  qsort_(a, 0, n-1);
}
```

Funkcija `izbor_pivota` odabire za pivot neki element niza `a[l, d]` i vraća njegov indeks (u nizu `a`). Pozivom funkcije `razmeni` pivot se postavlja na poziciju `l`. Funkcija `particionisanje` vrši particionisanje niza (pretpostavljajući da se pre particionisanja pivot nalazi na poziciji `l`) i vraća poziciju na kojoj se nalazi pivot nakon particionisanja. Funkcija se poziva samo za nizove koji imaju više od jednog elementa te joj je preduslov da je `l` manje ili jednako `d`. Postuslov funkcije particionisanje je da je (multi) skup elemenata niza `a` nepromenjen nakon njenog poziva, međutim njihov redosled je takav da su svi elementi niza `a[l, p-1]` manji ili jednaki elementu `a[p]`, dok su svi elementi niza `a[p+1, d]` veći ili jednaki od elementa `a[p]`.

Kako bi se dobila jednačina $T(n) = 2T(n/2) + O(n)$ i efikasnost $O(n \log n)$, potrebno je da korak particionisanja (tj. funkcija `particionisanje`) bude izvršen u linearnom vremenu $O(n)$. U nastavku će biti prikazano nekoliko algoritama particionisanja koji ovo zadovoljavaju. Dalje, potrebno je da pozicija pivota nakon particionisanja bude blizu sredini niza (kako dužina dva podniza na koje se problem svodi bilo približno jednaka $n/2$). Međutim, određivanje srednjeg člana u nizu brojeva (što predstavlja idealnu strategiju za funkciju `izbor_pivota`) je problem koji nije značajno jednostavniji od samog sortiranja. S obzirom da se očekuje da je implementacija funkcije brza (obično $O(1)$), obično se ne garantuje da će za pivot biti izabiran upravo srednji član, već se koriste heuristike koje za pivot biraju elemente koji nisu

daleko od središnje pozicije u nizu. Napomenimo da se za svaku strategiju izbora pivota (koja ne koristi slučajno izabrane brojeve) može konstruisati niz tako da u svakom koraku izbor pivota bude najgori mogući — onaj koji deli niz na nizove dužine `0` i `n-1`, što dovodi do jednačine $T(n) = T(n-1) + O(n)$ i kvadratne složenosti ($O(n^2)$). Međutim, većina strategija je takva da se u prosečnom slučaju može očekivati relativno ravnomerna raspodela što dovodi do optimalne složenosti ($O(n \log n)$).

Napomenimo da je *quick sort* algoritam koji u praksi daje najbolje rezultate kod sortiranja dugačkih nizova. Međutim, važno je napomenuti da kod sortiranje kraćih nizova naivni algoritmi (npr. *insertion sort*) mogu da se pokažu praktičnijim. Većina realnih implementacija *quick sort* algoritma koristi hibridni pristup — izlaz iz rekurzije se vrši kod nizova koji imaju nekoliko desetina elemenata i na njih se primenjuje *insertion sort*.

Implementacije particionisanja. Jedna od mogućih implementacija koraka particionisanja je sledeća:

```
int particionisanje(int a[], int l, int d) {
  int p = l, j;
  for (j = l+1; j <= d; j++)
    if (a[j] < a[l])
      razmeni(a, ++p, j);

  razmeni(a, l, p);
  return p;
}
```

Invarijanta petlje je da je (multi)skup elemenata u nizu `a` nepromenjen, kao i da se u nizu `a` na poziciji `l` nalazi pivot, da su elementi `a[l+1, p]` manji od pivota, dok su elementi `a[p+1, j-1]` su veći ili jednaki od pivota. Nakon završetka petlje, `j` ima vrednost `d+1`, te su elementi `a[p+1, d]` veći ili jednaki od pivota. Kako bi se ostvario postuslov funkcije `particionisanje` vrši se još razmena pivota i elementa na poziciji `p` — time pivot dolazi na svoje mesto (na poziciju `p`).

Drugi način implementacije particionisanja je zasnovan na Dijkstrinom algoritmu „trobojke" (Dutch National Flag Problem). U ovom slučaju, radi se malo više od onoga što postuslov striktno zahteva — niz se permutuje tako da prvo idu svi elementi striktno manji od pivota, zatim sva pojavljivanja pivota i na kraju svi elementi striktno veći od pivota.

```
int particionisanje(int a[], int l, int d) {
  int pn = l-1, pp = d+1, pivot = a[l], t = l;
  while (t < pp) {
    if (a[t] < pivot)
        razmeni(a, t++, ++pn);
    else if (a[t] > pivot)
        razmeni(a, t, --pp);
    else
        t++;
  }
  return pn+1;
}
```

Invarijanta petlje je da se (multi)skup elemenata niza ne menja, da su svi elementi niza `a[l, pn]` manji od pivota, da su svi elementi niza `a[pn+1, t-1]` jednaki pivotu, i da su svi elementi niza `a[pp, d]` veći od pivota. Kada se petlja završi važi da je `t` jednako `pp` tako da su svi elementi niza `a[l, pn]` manji od pivota, niza `a[pn+1, pp-1]` jednaki pivotu, a niza `a[pp, d]` veći od pivota.

Treći mogući način implementacije koraka particionisanja je da se obilazi paralelno sa dva kraja i kada se na levom kraju nađe element koji je veći od pivota, a na desnoj neki koji je manji od pivota, da se izvrši njihova razmena.

```
int particionisanje(int a[], int l, int d) {
  int l0 = l;

  while (l < d) {
    while (a[l] <= a[l0] && l < d)
       l++;
    while (a[d] >= a[l0] && l < d)
       d--;
    if (l < d)
       razmeni(a, l, d);
  }

  if (a[l] >= a[l0])
    l--;
  razmeni(a, l0, l);
  return l;
}
```

Invarijanta spoljašnje petlje je da je `l` manje ili jednako `d`, da je (multi)skup elemenata niza `a` nepromenjen (što je očigledno jer algoritam zasniva isključivo na razmenama), da se na poziciji `l0` nalazi pivot i da su elementi `a[l0, l-1]` manji od pivota (gde je `l0` početna vrednost promenljive `l`), a elementi `a[d+1, d0]` (gde je `d0` početna vrednost promenljive `d`) veći ili jednaki od pivota. Po završetku petlje je `l` jednako `d`. Element na toj poziciji još nije ispitan i njegovim ispitivanjem se osigurava da `l` bude takav da su elementi `a[l0, l]` manji ili jednak od pivota, a elementi niza `a[l+1, d0]` veći ili jednaki od pivota. Odavde, nakon zamene pozicija `l0` i `l` postiže se postuslov particionisanja.

S obzirom da se u svakom koraku petlji smanjuje pozitivna celobrojna vrednost `d - l`, a sve petlje se zaustavljaju u slučaju kada je `d - l` nula zaustavljanje sledi.

Naredna implementacija optimizuje prethodnu ideju, tako što izbegava korišćenje zamena.

```
int particionisanje(int a[], int l, int d) {
  int pivot = a[l];
  while (l < d) {
    while (a[d] >= pivot && l < d)
       d--;
    if (l != d)
      a[l++] = a[d];

    while (a[l] <= pivot && l < d)
       l++;
    if (l != d)
      a[d--] = a[l];
  }
  a[l] = pivot;
  return l;
}
```

Invarijanta spoljašnje petlje je da je `l` manje ili jednako `d`, da je (multi)skup elemenata niza `a` van pozicije `l` jednak (multi)skupu elemenata polaznog niza bez jednog pojavljivanja pivota, da su elementi `a[l0, l-1]` manji od pivota (gde je `l0` početna vrednost promenljive `l`), a elementi `a[d+1, d0]` (gde je `d0` početna vrednost promenljive `d`) veći ili jednaki od pivota. Na sredini spoljne petlje, invarijanta se donekle naruši — svi uslovi ostaju da važe, osim što je (multi)skup elemenata `a` van pozicije `d` (umesto van pozicije `l`) jednak (multi) skupu elemenata polaznog niza bez jednog pojavljivanja pivota. Kada se petlja završi, `l` je jednako `d` i upisivanjem (sačuvane) vrednosti pivota na ovo mesto se postiže postuslov funkcije particionisanja.

Zaustavljanje je analogno prethodnom slučaju.

Izbor pivota. Kao što je već rečeno, iako je poželjno da se pivot izabere tako da podeli niz na dve potpuno jednake polovine, to bi trajalo previše tako da se obično pribegava heurističkim rešenjima. Ukoliko se može pretpostaviti da su elementi niza slučajno raspoređeni (što se uvek može postići ukoliko se pre primene sortiranja niz permutuje na slučajan način), bilo koji element niza se može uzeti za pivot. Na primer,

```
int izbor_pivota(int a[], int l, int d) {
  return l;
}
```

ili

```
int izbor_pivota(int a[], int l, int d) {
  return slucajan_broj(l, d);
}
```

Bolje performanse se mogu postići ukoliko se npr. za pivot uzme srednji od tri slučajno izabrana elementa niza.

6.2.7 Korišćenje sistemske implementacije quick sort-a

Funkcija `qsort` (deklarisana u zagavlju `<stdlib.h>`) izvršava *quick sort* algoritam. Način korišćenja ove funkcije veoma je blizak načinu koiršćenja funkcije `bsearch`.

```
void qsort(void *base, size_t num, size_t width,
           int (*compare)(const void *elem1, const void *elem2));
```

Argumenti funkcije imaju sledeću ulogu:

base je pokazivač na početak niza koji se sortira; on je tipa `void*` kako bi mogao da prihti pokazivač na bilo koji konkretan tip;

num je broj elemenata niza;

width je veličina jednog elementa u bajtovima; ona je potrebna kako bi funkcija mogla da prelazi sa jednog na sledeći element niza;

compare je pokazivač na funkciju (često definisanu od strane korisnika) koja vrši poređenje dve vrednosti zadatog tipa; da bi funkcija imala isti prototip za sve tipove, njeni argumenti su tipa `void *` (zapravo — `const void *` jer vrednost na koju ukazuje ovaj pokazivač ne treba da bude menjana u okviru funkcije `compare`). Funkcija vraća jednu od sledećih vrednosti:

- < 0 ako je vrednost na koju ukazuje prvi argument manja od vrednosti na koju ukazuje drugi argument;
- 0 ako su vrednosti na koju ukazuju prvi i drugi argument jednake;
- > 0 ako je vrednost na koju ukazuje prvi argument veća od vrednosti na koju ukazuje drugi argument;

Niz se sortira u rastućem poretku određenom funkcijom sortiranja.

Navedimo nekoliko primera. Naredni program gradi slučajan niz celih brojeva i uređuje ga rastuće.

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

int ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     printf("%d ", a[i]);
  printf("\n");
}
```

```
int poredi_brojeve(const void *pa, const void *pb) {
  int a = *(int*)pa, b = *(int*)pb;
  if (a < b) return -1;
  else if (a > b) return 1;
  else return 0;
}

#define MAX 10
int main() {
    int i, niz[MAX];
    for(i = 0; i < MAX; i++)
        niz[i] = rand() % MAX;
    ispisi(niz, MAX);
    qsort(niz, MAX, sizeof(int), poredi_brojeve);
    ispisi(niz, MAX);
    return 0;
}
```

Naredni program leksikografski sortira argumente komandne linije.

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

int compare(const void *pa, const void *pb) {
   return strcmp(*(char**)pa, *(char**)pb);
}

void main( int argc, char **argv ) {
   int i;
   argv++; argc--;   /* argv[0] se ne sortira */
   qsort((void*)argv, (size_t)argc, sizeof(char*), compare);
   for(i = 0; i < argc; i++)
      printf("%s ", argv[i]);
   printf("\n");
}
```

Primer korišćenja:

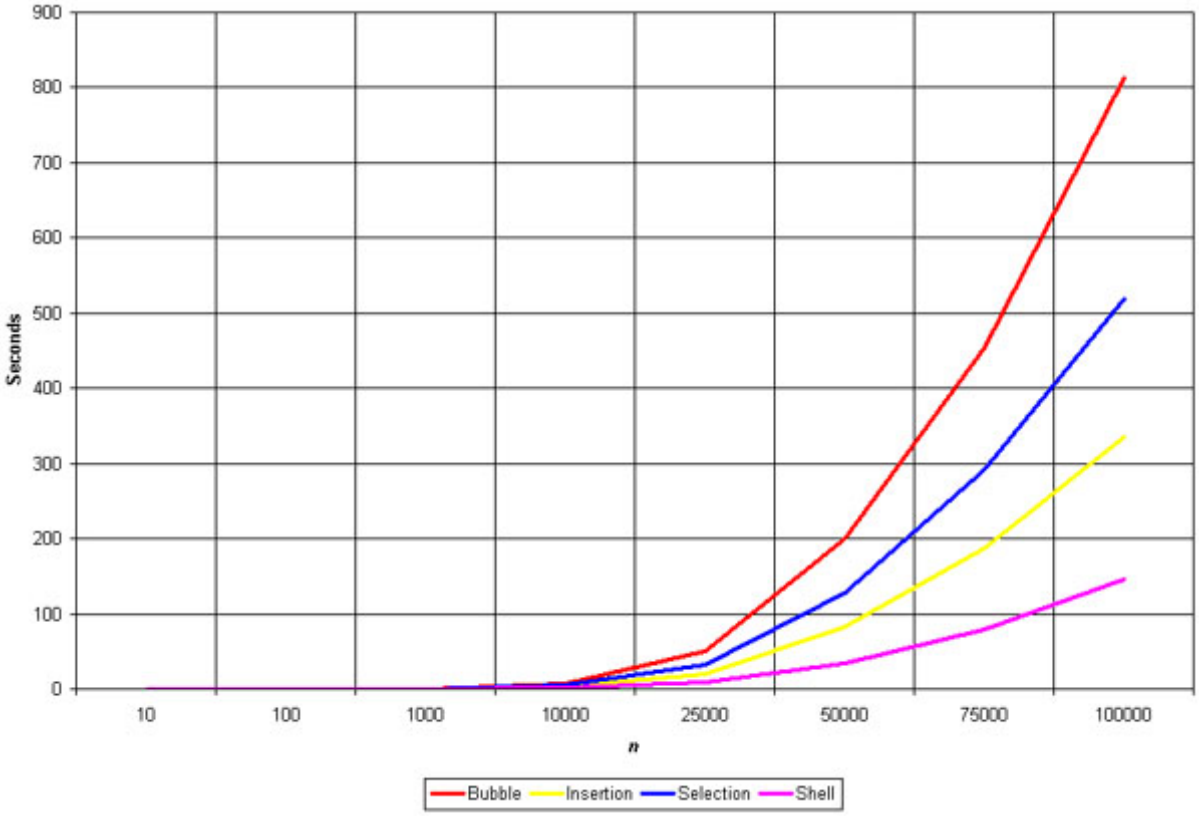
```
# ./qsort every good boy deserves favor
# boy deserves every favor good
```

6.2.8 Empirijsko poređenje različitih algoritama sortiranja

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. The empirical data given is the average of a hundred runs against random data sets on a single-user 250MHz UltraSPARC II. The run times on your system will almost certainly vary from these results, but the relative speeds should be the same - the selection sort runs in roughly half the time of the bubble sort on the UltraSPARC II, and it should run in roughly half the time on whatever system you use as well.

These empirical efficiency graphs (6.1 and 6.1) are kind of like golf - the lowest line is the "best". Keep in mind that "best" depends on your situation - the quick sort may look like the fastest sort, but using it to sort a list of 20 items is kind of like going after a fly with a sledgehammer.

As the graph 6.1 pretty plainly shows, the bubble sort is grossly inefficient, and the shell sort blows it out of the water. Notice that the first horizontal line in the plot area is 100 seconds - these aren't sorts



Slika 6.1: Rezultati za algoritme sortiranja složenosti $O(n^2)$

that you want to use for huge amounts of data in an interactive application. Even using the shell sort, users are going to be twiddling their thumbs if you try to sort much more than 10,000 data items.

On the bright side, all of these algorithms are incredibly simple (with the possible exception of the shell sort). For quick test programs, rapid prototypes, or internal-use software they're not bad choices unless you really think you need split-second efficiency.

Speaking of split-second efficiency, the $O(n \log n)$ sorts (slika 6.2) are where it's at. Notice that the time on this graph is measured in tenths of seconds, instead hundreds of seconds like the $O(n^2)$ graph.

But as with everything else in the real world, there are trade-offs. These algorithms are blazingly fast, but that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays - these algorithms make extensive use of those nasty things.

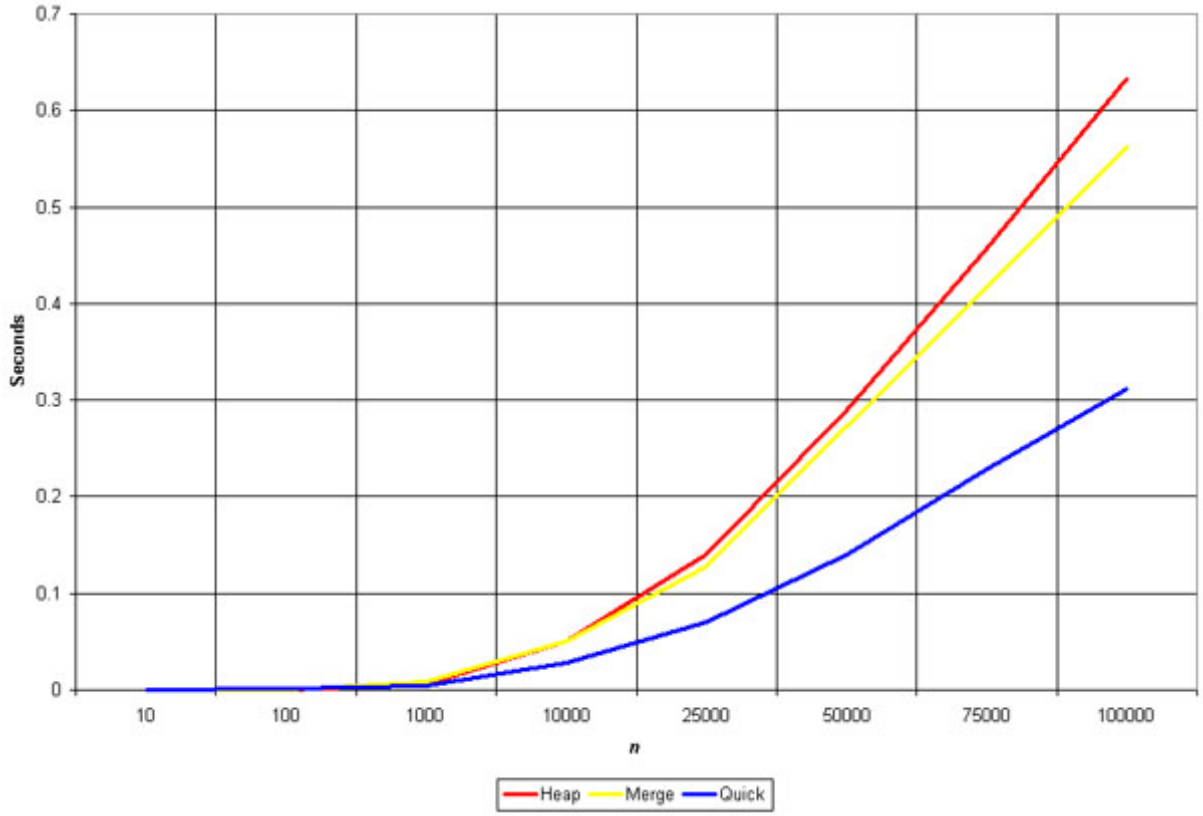
In the end, the important thing is to pick the sorting algorithm that you think is appropriate for the task at hand. You should be able to use the source code on this site as a "black box"if you need to - you can just use it, without understanding how it works. Obviously taking the time to understand how the algorithm you choose works is preferable, but time constraints are a fact of life.

The selection sort is in the group of n^2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

Realistically, there isn't a noticeable performance difference between the various sorts for 100 items



Slika 6.2: Rezultati za algoritme sortiranja složenosti $O(n \log n)$

or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

Pitanja i zadaci za vežbu

Pitanje 6.23. *Koje su dve operacije osnovne u većini algoritama sortiranja?*

Pitanje 6.24. *Navesti imena bar pet algoritama sortiranja i njihovu složenost u najgorem slučaju i prosečnu složenost.*

Pitanje 6.25. *Da li postoji algoritam za sortiranje čija se složenost u prosečnom i u najgorem slučaju razlikuju?*

Pitanje 6.26. *Opisati osnovnu ideju algoritma* selection sort.

Pitanje 6.27. *Prikazati stanje niza (nakon svake razmene) prilikom izvršavanja* selection sort *algoritma za niz* `5 3 4 2 1`.

Pitanje 6.28. *Šta važi nakon i-tog prolaska kroz petlju u algoritmu* selection sort*?*

Pitanje 6.29. *Za niz dužine n, koliko najefikasnije implementacije* selection sort *algoritma vrše poređenja i zamena?*

Pitanje 6.30. *Koji je slučaj najgori za algoritam sortiranja* selection sort*?*

Pitanje 6.31. *Opisati osnovnu ideju algoritma* bubble-sort.

Pitanje 6.32. *Koji ulazni niz predstavlja najgori slučaj za algoritam* bubble sort*?*

Pitanje 6.33. *Opisati osnovnu ideju algoritma* insertion sort.

Pitanje 6.34. *Prikazati stanje niza prilikom izvršavanja* insertion sort *algoritma za niz* `5 3 4 2 1`.

Pitanje 6.35. *Šta važi nakon i-tog prolaska kroz petlju u algoritmu* insertion sort*?*

Pitanje 6.36. *Kolika je složenost algoritma* insertion sort *u najgorem i u prosečnom slučaju?*

Pitanje 6.37. *Dopuniti implementaciju fukcije* `umetni` *koja se koristi u algoritmu za sortiranje* insertion sort *(i koja umeće i-i element na svoje mesto u nizu):*

```
void umetni(int a[], int i) {
  int j;
  for(_________________________________)
   razmeni(a, j, j-1);
}
```

Pitanje 6.38. *Opisati osnovnu ideju algoritma* merge sort*.*

Pitanje 6.39. *Kod algoritma* merge sort *je:*
(a) jednostavno razdvajanje na dva dela niza koji se sortira, ali je komplikovano spajanje;
(b) komplikovano razdvajanje na dva dela niza koji se sortira, ali je jednostavno spajanje;
(c) jednostavno je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;
(d) komplikovano je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;

Pitanje 6.40. *Dopuniti implementaciju algoritma* merge sort*:*

```
void mergesort_(int a[], int l, int d, int tmp[]) {
  if (l < d) {
    int i, j;
    int n = d - l + 1, s = l + n/2;
    int n1 = n/2, n2 = n - n/2;
    ________________________________
    ________________________________
    merge(a + l, n1, a + s, n2, tmp);
    for (i = l, j = 0; i <= d; i++, j++)
      a[i] = tmp[j];
}
```

Pitanje 6.41. *Opisati osnovnu ideju algoritma* quick sort*.*

Pitanje 6.42. *U okviru algoritma* quick sort *, kada se izabere pivot, potrebno je izvršiti:*
(a) permutovanje elemenata niza; (b) particionisanje elemenata niza; (c) invertovanje elemenata niza; (d) brisanje elemanata niza;

Pitanje 6.43. *Kod algoritma* quick sort *je:*
(a) jednostavno razdvajanje na dva dela niza koji se sortira, ali je komplikovano spajanje;
(b) komplikovano razdvajanje na dva dela niza koji se sortira, ali je jednostavno spajanje;
(c) jednostavno je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;
(d) komplikovano je i razdvajanje na dva dela niza koji se sortira i njihovo spajanje;

Pitanje 6.44. *Dopuniti implementaciju algoritma* quick sort*:*

```
void qsort_(int a[], int l, int d) {
  if (l < d) {
    razmeni(a, l, izbor_pivota(a, l, d));
  int p = particionisanje(a, l, d);

  ________________________________

  ________________________________
}
```

Pitanje 6.45. *Koja je složenost koraka particionisanja (za niz od n elemenata) koje se koristi u algoritmu quicksort?*

Pitanje 6.46. *Koji ulazni niz predstavlja najgori slučaj za algoritam* quick sort*?*

Pitanje 6.47. *Koja je složenost u najgorem slučaju algoritma* quick sort*:*
(a) ako se za pivot uzima prvi element niza?
(b) ako se za pivot uzima poslednji element niza?
(c) ako se za pivot uzima srednji (po indeksu) element niza?

Pitanje 6.48. *Da li se može promeniti složenost algoritma* quick sort| *pametnim algoritmom za izbor pivota?*
Koja se složenost u najgorem slučaju može tako dobiti?

Pitanje 6.49. *Navesti barem dva algoritma sortiranja iz grupe „podeli i vladaj“.*

Pitanje 6.50. *Kojem tipu algoritama (ne po klasi složenosti, nego po pristupu) pripadaju algoritmi za sortiranje* quick-sort| *i* merge sort*?*
Šta je teško a šta lako u prvom, a šta u drugom?

Pitanje 6.51. *Da li je u algoritmu* quick sort *razdvajanje na dva podniza lako ili teško (u smislu potrebnog vremena)? Da li je u algoritmu* quick sort *spajanje sortirana dva podniza lako ili teško (u smislu potrebnog vremena)?*

Pitanje 6.52. *Navesti prototip funkcije* `qsort` *iz* `stdlib.h`

Pitanje 6.53. *Navesti primer funkcije* `compare` *koja se može koristiti u okviru funkcije* `qsort`*:*

Pitanje 6.54. *Ako se bibliotečka funkcija* `qsort` *koristi za sortiranje niza struktura tipa* `S` *po članu* `kljuc` *tipa* `int`*, navesti funkciju za poređenje koju treba proslediti funkciji* `qsort`*.*

Pitanje 6.55. *Dopuniti funkciju tako da može da se koristi kao poslednji argument funkcije* `qsort` *za sortiranje niza niski leksikografski:*

```
int poredi(const void* px, const void* py) {
   return                                          ;

}
```

Pitanje 6.56. *Napisati primer funkcije koja koristi funkciju* `qsort` *iz standardne biblioteke.*

Zadaci

Zadatak 6.2.1. *Napisati program koji iz datoteke čije se ime zadaje kao argument komandne linije, čita prvo broj elemenata niza pa zatim i elemente niza (celi brojevi). Ovaj niz sortirati pozivom funkcije* qsort *a zatim za uneti ceo broj sa standradnog ulaza proveriti, pozivom funkcije* bsearch*, da li se nalazi u nizu ili ne i ispisati odgovarajuću poruku.*

Zadatak 6.2.2. *Sa standardnog ulaza učitavamo prvo broj studenata a zatim i njihove podatke. Za svakog studenta dobijamo ime (niska od najviše 30 karaktera) i broj indeksa (ceo broj). Napisati program koji sortira ovaj niz studenata po imenima studenata pozivom standardne funkcije* qsort *a zatim pronalazi broj indeksa studenta čije se ime zadaje sa standardnog ulaza pozivom funkcije* bsearch*.*

Zadatak 6.2.3. *Napisati program koji sa standardnog ulaza učitava podatke o studentima tako što za svakog studenta dobijamo prezime (karakterska niska od najviše* 30 *karaktera) i broj indeksa (ceo broj). Pretpostavka je da studenata nema više od* 100*.*
Sortirati ovaj niz studenata po prezimenima studenata pozivom standardne funkcije `qsort` *i ispisati ih na standardni izlaz.*

Zadatak 6.2.4. *Napisati program koji sa standardnog ulaza unosi prvo broj artikala a zatim i podatke o artiklima (ime artikla - karakterska niska dužine do* 20 *karaktera i cena artikla – ceo broj), sortira ih po ceni (pozivom funkcije* `qsort`*) i nakon toga (pozivom funkcije* `bsearch`*) određuje naziv artikla čiju cenu korisnik zadaje sa standardnog ulaza.*

Zadatak 6.2.5. *Napisati program koji sa standardnog ulaza unosi prvo broj studenata a zatim i podatke o studentima (ime studenata - karakterska niska dužine do* 30 *karaktera i broj indeksa studenta - ceo broj), sortira ih po imenu studenta leksikografski (pozivom funkcije* `qsort`*) i nakon toga (pozivom funkcije* `bsearch`*) određuje broj indeksa studenta čije ime korisnik zadaje sa standardnog ulaza.*

Zadatak 6.2.6. *Sa standardnog ulaza učitavamo prvo broj studenata a zatim i njihove podatke. Za svakog studenta dobijamo ime (niska od najviše 30 karaktera) i broj indeksa (ceo broj). Napisati program koji sortira ovaj niz studenata po imenima studenata pozivom standardne funkcije* qsort *i zatim štampa tako dobijeni niz na standardni izlaz.*

Zadatak 6.2.7. *Napisati program koji sa standardnog ulaza učitava broj artikala (ne više od* 50*) a zatim imena (karakterske niske dužine do* 20 *karaktera) i cene artikala (ceo broj). Ovaj niz artikala sortirati po ceni (pozivom funkcije* qsort*) a zatim za uneti ceo broj c sa standardnog ulaza pronaći (pozivom funkcije* bsearch*) naziv artikla sa tom cenom. Ako takav artikal ne postoji ispisati odgovarajuću poruku.*

Zadatak 6.2.8. *Napisati program koji radi sa tačkama. Tačka je predstavljena svojim x i y koordinatama (celi brojevi). Sa standardnog ulaza se učitava prvo broj tačaka a zatim koordinate tačaka. Dobijeni niz struktura sortirati pozivom funkcije* qsort*. Niz sortirati po x koordinati, a ako neke dve tačke imaju istu x koordinatu onda ih sortirati po y koordinati.*

Ako na ulazu dobijete niz: $(4, 6)$, $(2, 9)$, $(4, 5)$*; sortirani niz će izgledati:* $(2, 9)$, $(4, 5)$, $(4, 6)$.

Zadatak 6.2.9. *Napisati program koji sa standardnog ulaza unosi prvo broj studenata a zatim i podatke o studentima (ime studenata - karakterska niska dužine do* 30 *karaktera i broj indeksa studenta - ceo broj), sortira ih po imenu studenta leksikografski (pozivom funkcije qsort) i ispisuje sortiran niz na standardni ulaz.*

Zadatak 6.2.10. *Napisati program koji omogućava učitavanje artikala iz datoteke koja se zadaje prvim argumentom komandne linije. Jedan artikal je definisan strukturom*

```
typedef struct {
  char ime[50];
  float tezina;
  float cena;
} Artikal;
```

Datoteka je ispravno formatirana i sadrži najviše 500 artikala. Program treba da, nakon učitavanja, ispiše sve artikle sortirane po zadatom kriterijumu. Kriterijum se zadaje kao drugi argument komandne linije i može biti: `i` *- sortirati po imenu;* `t` *- sortirati po težini;* `c` *- sortirati po ceni.*

Podrazumeva se da je sortiranje u rastućem redosledu, a da se za sortiranje po imenu koristi leksikografski poredak (dozvoljeno je korišćenje funkcije `strcmp`*). Koristiti generičku funkciju za sortiranje* `qsort` *iz standardne biblioteke.*

Zadatak 6.2.11. *Ime tekstualne datoteke zadaje se kao argument komadne linije. U svakom redu datoteke nalazi se ime proizvoda (ne više od 20 karaktera) i količina u kojoj se proizvodi (broj redova datoteke nije poznat). Proizvodi su leksikografski poređani. Sa standardnog ulaza učitava se ime proizvoda. Korišćenjem sistemske funkcije* `bsearch` *pronaći u kojoj meri se dati proizvod proizvodi. NAPOMENA: koristiti strukturu*

```
tupedef struct
{
    char ime[20];
    int kolicina;
}proizvod;
```

Zadatak 6.2.12. *Sa standardog ulaza se zadaje ime tekstualne datoteke koja sadrži podatke o artiklima prodavnice. Datoteka je u formatu:*
`<bar kod> <ime artikla> <proizvodjac> <cena>` *i sortirana je prema* `<bar kod>`*. Nije unapred poznat broj artikala u datoteci. Učitati podatke o artiklima u niz (niz alocirati dinamički). Zatim se sa standardnog ulaza unose bar kodovi artikla sve dok se ne unese 0. Izračunati ukupnu cenu unetih proizvoda. (koristiti ugradjenu f-ju* `bsearch` *za traženje artikla sa datim bar kodom).*

Zadatak 6.2.13. *a) Definisati tip podataka za predstavljanje studenata, za svakog studenta poznato je: nalog na Alasu (oblika napr. mr97125, mm09001), ime (maksimalne dužine 20 karaktera) i broj poena.*

b) Podaci o studentima se nalaze u datoteci `studenti. txt` *u obliku: nalog ime br.poena. Kao argument komadne linije korisnik unosi opciju i to može biti* `-p` *ili* `-n`*. Napisati program koji sortira korišćenjem ugrađene funkcije* `qsort` *studente i to: po broju poena ako je prisutna opcija* `-p`*, po nalogu ukoliko je prisutna opcija* `-n`*, ili po imenu ako nije prisutna ni jedna opcija. Studenti se po nalogu sortiraju tako što se sortiraju na osnovu godine, zatim na osnovu smera i na kraju na osnovu broja indeksa.*

Zadatak 6.2.14. *U svakom redu datoteke* `transakcije.txt` *nalazi se identifikacija (niska maksimalne dužine 20) korisnika banke i iznos transakcije koju je korisnik napravio (ceo broj). Jedan korisnik može imati više transakcija, a svaka je predstavljena celim brojem (negativan - isplata sa računa, pozitivan - uplata na račun). Ispisati identifikacioni broj korisnika koji je najviše zadužen.*
NAPOMENA: Kreirati strukturu `klijent`*, učitati sve korisnike u dinamicki alociran niz koji je u svakom trenutku sortiran po identifikaciji (obavezno koristiti ugrađenu funkciju* `qsort` *i za pronalaženje korisnika ugrađenu funkciju* `bsearch`*), a zatim u jednom prolasku kroz niz naći najzaduženijeg korisnika.*

Zadatak 6.2.15. *Data je datoteka* `voce.txt` *koja ima najviše 100 redova. U svakom redu se nalazi ime voćke (ne duže od 20 karaktera) i cena (int). Napisati funkciju* `int f4(int p, char* ime, int** cena)` *koja otvara datoteku i učitava podatke iz nje u niz struktura i sortira taj niz po cenama. U promenljive* `ime` *i* `cena` *upisuju se ime voćke i cena voćke koja se nalazi na poziciji* `p` *u sortiranom nizu. Ukoliko* `p` *ima nekorektnu vrednost funkcija vraća -1 a inače vraća 1.* `p` *je nekorektno ako je manje od 0 ili veće ili jednako od broja elemenata niza.*

Primeri(podaci o voćkama su u datoteci `voce.txt`*):*

```
Primer 1:                Primer 2:                Primer 3:                Primer 4:

jabuka 10                ananas 12                jagode 14                limun 45
kruska 15                kivi 34                  breskve 23               grejpfrut 23
sljiva 23                                         pomorandze 11
malina 7                 p = 5                    banane 3                 p = -1

p = 2                    funkcija vraca -1;       p = 1                    funkcija vraca -1;

ime = kruska;                                     ime = pomorandze
cena = 15;                                        cena = 11
funkcija vraca 1;                                 funkcija vraca 1;
```

Zadatak 6.2.16. *U datoteci* `podaci.txt` *se nalaze reči i pozitivni celi brojevi. Napisati funkciju* `void f4(int m, int n)` *koja sortira leksikografski reči u rastućem poretku i ispisuje reč na poziciji* `m`*. Potom sortirati brojeve u opadajućem poretku i ispisati broj na poziciji* `n`*. Ukoliko datoteka ne postoji ispisati -1. Ukoliko* `m` *i* `n` *nisu u odgovarajućem opsegu ispisati -1. Reči su maksimalne dužine 20 karaktera.*

Zadatak 6.2.17. *Argumenti komandne linije su cu celi brojevi* `a`*,* `n`*,* `m`*,* `p`*. Napisati program koji računa niz*

```
a mod n,
2a mod n,
3a mod n
```

```
...
m*a mod n
```

sortira ga u rastućem poretku i ispisuje u datoteku `rez.txt` *p-ti član niza (brojanje indeksa pocinje od 0)*

Ukoliko je došlo do neke od grešaka (nije tačan broj argumenata komandne linije, p>n,...) u `rez.txt` *upisati* `-1`.

Zadatak 6.2.18. *Data je struktura student:*

```
typedef struct
{
    char ime[20];
    char prezime[20];
}student;
```

Napisati funkciju
`void pronadji(student *studenti, int n, int *max_p, int *max_p_i)`, *koja bibliotečkim qsort–om, sortira niz, prvo prema pema prezimenu, a zatim i prema imenu, a zatim u sortiranom nizu studenata pronalazi maksimalan broj studenata koji imaju isto prezime i maksimalan broj studenata koji imaju isto ime i prezime. Ove podatke vraća kroz prosleđene parametre max_p i max_p_i.*

Zadatak 6.2.19. *Argument programa je putanja tekstualne datoteke koja sadrži isključivo cele brojeve. Napisati program koji pronalazi i ispisuje na standardnom izlazu dva broja koja se najmanje razlikuju (ako ima više parova koji se isto razlikuju, ispisati bilo koji par). Uputstvo: koristiti funkciju* `qsort`.

Zadatak 6.2.20. *Napisati funkciju koja sortira slova unutar niske karaktera i ispisuje ih na standardni izlaz. Napisati program koji proverava da li su dve niske karaktera koje se zadaju kao argumenti komandne linije anagrami. Dve niske su anagrami ako se sastoje od istog broja istih karaktera. Program treba na standardni izlaz da ispiše true ili false.*

```
Primer 1:    ./anagram miroljub ljubomir          bijlmoru bijlmoru true
Primer 2:    ./anagram vatra trava                aartv aartv true
Primer 3:    ./anagram racunar racun              aacnrru acnru false
Primer 4:    ./anagram program gram               agmoprr agmr false
```

Zadatak 6.2.21. *Sa standardnog ulaza unosi se broj reči* `n`, *za kojim sledi* `n` *reči (u svakom redu po jedna). Nakon unosa reči unosi se i broj* `q`. *Reči nisu duže od 20 karaktera. Učitati reči u niz, a zatim ih sortirati. Ispisati reč na poziciji* `q`. *Kriterijum sortiranja je suma* `ascii` *kodova slova svake reči, poredak je rastući.*

Zadatak 6.2.22. *Napisati funkciju* `unsigned swap_pair(unsigned x, int i, int j)` *koja razmenjuje vrednosti bitova na pozicijama* `i` *i* `j`, *i vraća rezultat kao povratnu vrednost. Bit sa najmanjom težinom nalazi se na poziciji 0, bit do njega na poziciji 1, itd... Napisati i program koji testira ovu funkciju. Sa standardnog ulaza se unose redom broj kome se invertuju bitovi, i pozicije bitova, a rezultat funkcije se ispisuje na standardni izlaz.*

Zadatak 6.2.23. *Sa standardnog ulaza se ucitava broj n a zatim i niz od n niski(svaka niska je maksimalne duzine 20 karaktera, duzina niza n nije unapred poznata niti ogranicena). Sortirati taj niz u opadajucem poretku pri cemu je kriterijum poredenja broj samoglasnika u reci. Ispisati sortirani niz na standardni izlaz. U slucaju greske ispisati -1"na standardni izlaz.*

6.3 Jednostavni algebarsko-numerički algoritmi

U ovom poglavlju biće prikazani neki algebarsko-numerički algoritmi u kojima je direktno izračunavanje ("grubom silom") zamenjeno efikasnijim.

6.3.1 Izračunavanje vrednosti polinoma

Vrednost polinoma oblika

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

za konkretnu vrednost x može se izračunati ("grubom silom") korišćenjem $n + (n-1) + \ldots + 1$ množenja i n sabiranja, ali se može izračunati i znatno efikasnije korišćenjem sledećeg *Hornerovog* zapisa istog polinoma:

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

Naredna funkcija, zasnovana na navedenom zapisu, koristi samo n množenja i n sabiranja.

```
double vrednost_polinoma(double x, double a[], int n) {
  int i;
  double v=a[n];
  for (i = n-1; i >= 0; i--)
    v = x*v+a[i]
}
```

Pitanja i zadaci za vežbu

Pitanje 6.57. *Napisati funkciju za efikasno izračunavanje vrednost n^k.*

Pitanje 6.58. *Funkcija koja efikasno izračunava x^k se može implementirati na sledeći način:*

```
float power(float x, unsigned k) {
   if (k == 0)
      return ____;
   else if (k % 2 == 0)
      return _____________;
   else
      return _____________;
}
```

Pitanje 6.59. *Dopuniti implementaciju fukcije za računanje vredosti polinoma:*

```
  double vrednost_polinoma(double x, double a[], int n) {
  int i;
  double v=a[n];
  for (i = n-1; i >= 0; i--)
    v = ______________
  return v;
}
```

Pitanje 6.60. *Funkcija* `int stepen_brzo(int n, int k)` *efikasno racuna vrednost n^k. Koje vrednosti izračunava i vraća ova funkcija ako je k neparan broj?*

6.4 Generisanje kombinatornih objekata

6.4.1 Varijacije sa ponavljanjem

Varijacija sa ponavljanjem dužine k nad n elemenata je niz dužine k, pri čemu je svaki element niza jedan od n zadatih elemenata. Za listanje varijacija proizvoljnog skupa od n objekata dovoljno je imati mehanizam za listanje varijacija brojeva od 1 do n. Naime, svakom od n objekata može se pridružiti jedan od brojeva od 1 do n, a onda varijaciju ovih brojeva prevesti u permutaciju zadatih objekata. Varijacija dužine k nad n elemenata ima n^k. Kao primer, u nastavku su izlistane sve varijacija sa ponavljanjem dužine 2 nad 3 brojeva (od 1 do 3):

```
1 1, 1 2, 1 3, 2 1, 2 2, 2 3, 3 1, 3 2, 3 3
```

Naredna rekurzivna funkcija ispisuje sve varijacije sa ponavljanjem dužine k brojeva između 1 i n .

```
void varijacije_(int a[], int i, int n, int k) {
   int j;
   if(i == k)
     ispisi(a, k);
   else
     for(j=1; j<=n; j++) {
       a[i] = j;
       varijacije_(a, i+1, n, k);
     }
}
```

Kako bi se korisnik oslobodio potrebe za kreiranjem pomoćnog niza `a`, moguće je koristiti naredni omotač.

```
void varijacije(int n, int k) {
  int* a = (int*) malloc(n*sizeof(int));
  varijacije(a, 0, n, k);
  free(a);
}
```

U nekim slučajevima, potrebno je za datu varijaciju pronaći sledeću varijaciju u leksikografskom poretku. To može biti urađeno korišćenjem naredne funkcije:

```
int sledeca_varijacija(int a[], int n, int k) {
  int i;
  for (i = k - 1; i >= 0 && a[i] == n; i--)
     a[i] = 1;
  if (i < 0)
     return 0;
  a[i]++;
  return 1;
}
```

Prethodna funkcija se takođe može iskoristiti za listanje svih varijacija. Na primer,

```
  for(i=0; i<k; i++)
    a[i] = 1;
  do {
    ispisi(a, k);
  } while (sledeca_varijacija(a, n, k));
```

6.4.2 Kombinacije

Kombinacija dužine k nad n elemenata je jedan podskup veličine k skupa veličine n. Kombinacija dužine k nad n elemenata ima $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Kao primer, u nastavku su izlistane sve kombinacije dužine 3 nad 5 brojeva (od 1 do 5):

```
1 2 3, 1 2 4, 1 2 5, 1 3 4, 1 3 5, 1 4 5, 2 3 4, 2 3 5, 2 4 5, 3 4 5
```

Naredni kôd lista sve kombinacije dužine k od elemenata $1, \ldots, n$.

```
void kombinacije_(int a[], int i, int k, int min, int n) {
  if (k == 0)
    ispisi(a, i);
  else
    if (k <= n - min + 1) {

      a[i] = min;
      kombinacije_(a, i+1, k-1, min+1, n);
```

```
      kombinacije_(a, i, k, min+1, n);
    }
}
```

Kako bi se korisnik oslobodio potrebe za kreiranjem pomoćnog niza `a`, moguće je koristiti naredni omotač.

```
void kombinacije(int k, int n) {
  int* a = (int*)malloc(k*sizeof(int));
  kombinacije_(a, 0, k, 1, n);
  free(a);
}
```

Ukoliko je data neka kombinacija, naredna funkcija pronalazi sledeću kombinaciju u leksikografskom poretku.

```
int sledeca_kombinacija(int a[], int k, int n) {
  int i = k-1;
  while(i >= 0 && a[i] == n-k+i+1)
     i--;

  if (i < 0)
    return 0;

  a[i]++;
  for (i=i+1; i<k; i++)
    a[i] = a[i-1]+1;
  return 1;
}
```

Prethodna funkcija se takođe može iskoristiti za listanje svih kombinacija. Na primer,

```
  for(i=0; i<k; i++)
    a[i] = i+1;
  do {
    ispisi(a, k);
  } while (sledeca_kombinacija(a, k, n));
```

6.4.3 Permutacije

Permutacija je jedan poredak zadatog niza objekata. Ako zadatih objekata ima n, onda permutacija nad tim objektima ima $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$. Za listanje permutacija proizvoljnog skupa od n objekata dovoljno je imati mehanizam za listanje brojeva od 1 do n. Naime, svakom od n objekata može se pridružiti jedan od brojeva od 1 do n, a onda permutaciju ovih brojeva prevesti u permutaciju zadatih objekata. Kao primer, u nastavku su izlistane sve kombinacije brojeva od 1 do 3:

```
1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1
```

Postoji više različitih algoritama za listanje svih permutacija. Problem generisanja slučajne permutacije je takođe interesantan i on se suštinski razlikuje od problema listanja svih permutacija. U nastavu su data dva algoritma za listanje svih permutacija brojeva od 1 do n.

```
void permutacije_(int a[], int upotrebljen[], int n, int k) {
  int i;

  if(k == n) {
    ispisi(a, n);
    return;
```

```
  }

  for (i=0; i<n; i++)
    if (!upotrebljen[i])  {
      a[k]=i+1;
      upotrebljen[i] = 1;
      permutacije_(a, upotrebljen, n, k+1);
      upotrebljen[i] = 0;
    }
  }
}
```

Naredna funkcija predstavlja omotač koji oslobađa korisnika potrebe za alociranjem pomoćnih nizova.

```
void permutacije(int n) {
  int* upotrebljen = (int*)calloc(n, sizeof(int));
  int* a = (int*) malloc(n*sizeof(int));
  permutacije_(a, upotrebljen, n, 0);
  free(upotrebljen); free(a);
}
```

Druga varijanta (u ovoj varijanti permutacije nisu poređane leksikografski):

```
void permutacije_(int a[], int n, int k) {
  int i;
  if(k == n)
    ispisi(a, n);
  else
    for (i=k; i<n; i++) {
      razmeni(a, k, i);
      permutacije_(a, n, k+1);
      razmeni(a, k, i);
    }
}
```

Glavna funkcija koja predstavlja omotač se ponovo jednostavno implementira.

```
void permutacije(int n) {
  int i;
  int* a = (int*) malloc(n*sizeof(int));
  for (i=0; i<n; i++) {
    a[i]=i+1;
  permutacije_(a, n, 0);
  free(a);
}
```

Nekad je značajno za datu permutaciju pronaći sledeću permutaciju u leksikografskom poretku.

```
int sledeca_permutacija(int a[], int n) {
  int i, j;

  for (i = n - 2; a[i] > a[i+1]; i--)
    if (i < 0)
      return 0;

  for (j = n - 1; a[i] > a[j]; j--)
    ;

  razmeni(a, i, j);
```

```
  obrni(a, i+1, n-1);
  return 1;
}
```

Pri tom, funkcija `obrni` obrće deo niza između pozicija `i` i `j` (uključujući i njih.

```
void obrni(int a[], int i, int j) {
  int i1, j1;
  for (i1 = i, j1 = j; i1 < j1; i1++, j1--)
    razmeni(a, i1, j1);
}
```

Funkcija koja pronalazi sledeću permutaciju može biti upotrebljena i za generisanje svih permutacija. Na primer,

```
  for (i = 0; i < N; i++)
    a[i] = i+1;
  do {
    ispisi(a, N);
  } while (sledeca_permutacija(a, N));
```

6.4.4 Particionisanje

Particija pozitivnog celog broja n je niz pozitivnih celih brojeva čiji je zbir n. Na primer, za $n = 4$ sve particije su:

```
4
3 1
2 2
2 1 1
1 3
1 2 1
1 1 2
1 1 1 1
```

Sledeća funkcija lista sve particije zadatog broja k.

```
void particije_(int a[], int i, int n) {
  int j;
  if (n == 0)
     ispisi(a, i);
  else
  for(j=n; j>0; j--) {
    a[i]=j;
    particije_(a, i+1, n-j);
  }
}
```

Naredni omotač oslobađa korisnika potrebe za obezbeđivanjem prostora za pomoćni niz `a`.

```
void particije(int n) {
  int* a = (int*)malloc(n*sizeof(int));
  particije_(a, 0, n);
  free(a);
}
```

Pitanja i zadaci za vežbu

Pitanje 6.61. *Koja je leksikografski sledeća varijacija sa ponavljanjem skupa* $\{1, 2, 3\}$ *dužine 4 u odnosu na varijaciju* 2313*?*

Pitanje 6.62. *Dopuniti naredni kod koji generiše sledeću varijaciju dužine* k *od elemenata* $1, 2, ..., n$*:*

```
int sledeca_varijacija(int a[], int n, int k) {
    int i;
    for (i = k - 1; i >= 0 && a[i] == ???; i--)
        a[i] = ___________;
    if (i < 0)
        return 0;
    a[i]++;
    return 1;
}
```

Pitanje 6.63. *Dopuniti implementaciju fukcije koja generiše sledeću varijaciju sa ponavljanjem dužine* k *od* n *elemenata:*

```
int sledeca_varijacija(int a[], int n, int k) {
  int i;
  for (i = k - 1; i >= 0 && a[i] == n; i--)
  ----------------------------
  if (i < 0)
    return 0;
  ----------------------------
  return 1;
}
```

Pitanje 6.64. *Dopuniti naredni kôd tako da lista sve varijacije sa ponavljanjem dužine* k *brojeva između* 1 *i* n*:*

```
void varijacije_(int a[], int i, int n, int k) {
  int j;
  if(i == k)
    ispisi(a, k);
  else
    for(j=1; j<=n; j++) {


    }
}
```

Pitanje 6.65. *Data je permutacija 41532. Koja je permutacija sledeća u leksikografskom poretku?*

Pitanje 6.66. *Dopuniti narednu fukciju tako da ona ispisuje sve kombinacije dužine* k *od elemenata* $1, 2, \ldots, n$*:*

```
void kombinacije_(int a[], int i, int k, int min, int n) {
  if (k == 0)
    ispisi(a, i);
  else
    if _______________________________

      a[i] = min;
      kombinacije_(a, i+1, k-1, min+1, n);
```

```
      kombinacije_(a, i, k, min+1, n);
    }
}
```

Pitanje 6.67. *Dopuniti naredni kod koji generiše sledeću kombinaciju dužine k od elemenata $1, 2, ..., n$:*

```
int sledeca_kombinacija(int a[], int k, int n) {
    int i = k-1;
    while(i >= 0 && a[i] == n-k+i+1)
      --------;
    if (i < 0)
        return 0;
    a[i]++;
    for (i=i+1; i<k; i++)
        a[i] = ???;
    return 1;
}
```

Pitanje 6.68. *Navesti sve particije broja 4:* _____

6.5 Algoritmi zasnovani na bitovskim operatorima

U nekim slučajevima, efikasnija rešenja mogu se dobiti korišćenjem pojedinačnih bitova u zapisu celobrojnih izraza. C podržava naredne bitovske operatore (moguće ih je primenjivati samo na celobrojne argumente):

`&` – **bitovsko *i*** — primenom ovog operatora vrši se konjunkcija pojedinačnih bitova dva navedena argumenta (i-ti bit rezultata predstavlja konjunkciju i-tih bitova argumenata). Na primer, ukoliko su promenljive `x1` i `x2` tipa `unsigned char` i ukoliko je vrednost promenljive `x1` jednaka 74, a promenljive `x2` jednaka 87, vrednost izraza `x1 & x2` jednaka je 66. Naime, broj 74 se binarno zapisuje kao `01001010`, broj 87 kao `01010111`, i konjunkcija njihovih pojedinačnih bitova daje `01000010`, što predstavlja zapis broja 66. S obzirom da se prevođenje u binarni sistem efikasnije sprovodi iz heksadekadnog sistema nego iz dekadnog, prilikom korišćenja bitovskih operatora konstante se obično zapisuju heksadekadno. Tako bi u prethodnom primeru broj `x1` imao vrednost `0x4A`, broj `x2` bi imao vrednost `0x57`, a rezultat bi bio `0x42`.

`|` – **bitovsko *ili*** — primenom ovog operatora vrši se (obična) disjunkcija pojedinačnih bitova dva navedena argumenta. Za brojeve iz tekućeg primera, rezultat izraza `x1 | x2` bio bi `01011111`, tj. 95, tj. `0x5F`.

`^` – **bitovsko *ekskluzivno ili*** — primenom ovog operatora vrši se ekskluzivna disjunkcija pojedinačnih bitova dva navedena argumenta. Za brojeve iz tekućeg primera, rezultat izraza `x1 ^ x2` bio bi `00011101`, tj. 29, tj. `0x1D`.

`~` – **jedinični komplement** — primenom ovog operatora vrši se komplementiranje (invertovanje) svakog bita argumenta. Na primer, vrednost izraza `~x1` u tekućem primeru je `10110101`, tj. `B5`, tj. 181.

`«` – **levo pomeranje (šiftovanje)** — primenom ovog operatora bitovi prvog argumenta se pomeraju u levo za broj pozicija naveden kao drugi argument. Početni bitovi prvog argumenta se zanemaruju, dok se na završna mesta rezultata uvek upisuju nule. Levo pomeranje broja za jednu poziciju odgovara množenju sa dva. Na primer, ukoliko promenljiva `x` ima tip `unsigned char` i vrednost `0x95`, tj. `10010101`, vrednost izraza `x << 1` je `00101010`, tj. `0x2A`.

`»` – **desno pomeranje (šiftovanje)** — primenom ovog operatora bitovi prvog argumenta se pomeraju u desno za broj pozicija naveden kao drugi argument. Krajnji bitovi prvog argumenta se zanemarju, a što se tiče početnih bitova rezultata, postoji mogućnost da se oni popunjavaju uvek nulama (tzv. *logičko pomeranje*) ili da se oni popune vodećim bitom (koji predstavlja znak) prvog argumenta (tzv. *aritmetičko pomeranje*). Osnovna motivacija aritmetičkog pomeranja je da

desno pomeranje broja za jednu poziciju odgovara celobrojnom deljenju sa dva. U C-u, tip prvog argumenta određuje kako će se vršiti pomeranje. Ukoliko je argument neoznačen, početni bitovi rezultata će biti postavljeni na nulu, bez obzira na vrednost vodećeg bita prvog argumenta. Ukoliko je argument označen, početni bitovi rezultata će biti postavljeni na vrednost vodećeg bita prvog argumenta. Na primer, ukoliko promenljiva `x` ima tip `signed char` i vrednost `0x95`, tj. `10010101`, vrednost izraza `x >> 1` je `11001010`, tj. `0xCA`. Ukoliko je tip promenljive `x` `unsigned char`, tada je vrednost izraza `x >> 1` broj `01001010`, tj. `0x4A`.

`&=, |=, «=, »=` – bitovske dodele — ovi operatori kombinuju bitovske operatore sa dodelom (analogno, na primer, operatoru `+=`).

Napomenimo da bitovske operatore ne treba mešati sa logičkim operatorima. Na primer, vrednost izraza `1 && 2` je `1` (tačno i tačno je tačno), dok je vrednost izraza `1 & 2` jednaka `0` (`000...001 & 000..010 = 000...000`).

Kao unarni operator, operator `~` ima najveći prioritet i desno je asocijativan. Prioritet operatora pomeranja je najveći od svih binarnih bitovskih operatora — nalazi se između prioriteta aritmetičkih i relacijskih operatora. Ostali bitovski operatori imaju prioritet između relacijskih i logičkih operatora i to `&` ima veći prioritet od `^` koji ima veći prioritet od `|`. Ovi operatori imaju levu asocijativnost. Bitovski peratori dodele imaju niži prioritet (jednak ostalim operatorima dodele) i desnu asocijativnost.

Maskiranje. Kako bi se postigao željeni efekat nad bitovima datog broja, običaj je da se vrši njegovo kombinovanje bitovskim operatorima sa specijalno pripremljenim konstantama koje se obično nazivaju *maske*. U nastavku ćemo pokazati upotrebu nekih od najčešće korišćenih operacija ovog oblika.

Osobina konjunkcije je da je za svaki bit `b`, vrednost `b & 0` jednaka `0`, dok je vrednost `b & 1` jednaka `b`. Ovo znači da se konjunkcijom sa nekom maskom dobija rezultat koji je jednak broju koji se dobije kada se u broj `x` upišu nule na sve one pozicije na kojima maska ima bit `0` (ostali bitovi ostaju neizmenjenim). Na primer, konjunkcijom sa maskom `0xFF` „izdvaja" se poslednji bajt broja. Pretposlednji bajt bi se mogao izdvojiti korišćenjem `x >> 8 & 0xFF`. Ispitivanje da li bit na poziciji `i` broja `x` postavljen je moguće uraditi ispitivanjem `if (x & mask)` gde je maska `mask` napravljena tako da ima jedan jedini bit postavljen na jedan i to na poziciji `i` (na primer, `if (x & 1 << i)`).

Osobina disjunkcije je da je za svaki bit `b`, vrednost `b | 0` jednaka `b`, dok je vrednost `b | 1` jednaka `1`. Dakle, disjunkcijom broja `x` sa nekom maskom dobija se rezultat u kojem su upisane jedinice na sve one pozicije na kojima maska ima bit `1` (a ostali bitovi ostaju neizmenjeni). Na primer, disjunkcijom sa maskom `0xFF` „upisuju se jedinice" u poslednji bajt broja.

Kombinacijom prethodnih operacija maskiranja, moguće je umetati određene fragmente bitova u dati broj. Tako se na primer, izrazom `(x & ~0xFF) | 0xAB` upisuje 8 bitova `10101011` u poslednji bajt broja `x` (konjunkcijom se čiste postojeći bitovi, da bi se disjunkcijom upisao novi sadržaj).

Osobina ekskluzivne disjunkcije je da za svaki bit `b`, vrednost `b ^ 0` jednaka `b`, dok je vrednost `b ^ 1` jednaka `~b`. Ovo znači da se ekskluzivnom disjunkcijom sa nekom maskom dobija rezultat koji je jednak broju koji se dobije kada se invertuju bitovi na onim pozicijama na kojima se u maski nalazi jedinica. Na primer, konjunkcija sa maskom `0xFF` invertuje poslednji bajt broja.

6.5.1 Primeri.

Štampanje bitova broja. Naredna funkcija stampa bitove datog celog broja `x`. Vrednost bita na poziciji `i` je 0 ako i samo ako se pri konjunkciji broja x sa maskom `000..010....000` (sve nule osim jedinice na poziciji `i`) dobija 0. Funkcija kreće od pozicije najveće težine kreirajući masku koja ima jednu jedinicu i to na mestu najveće težine, i zatim pomerajući ovu masku za jedno mesto u levo u svakoj sledećoj iteraciji sve dok maska ne postane 0. Primetimo da je ovde bitno da maska ima tip `unsigned` kako bi se vršilo logičko pomeranje.

```
void print_bits(int x) {
int wl = sizeof(int)*8; /* Broj bitova tipa int */
unsigned mask;
for (mask = 1 << wl-1; mask; mask >>= 1)
putchar(x & mask ? '1' : '0');
}
```

Brojanje bitova. Naredna funkcija izračunava koliko ima bitova sa vrednošću 1 u zadatom broju n:

```
int bit_count(int n) {
  int i;
  int count = 0;
  for (i = 0; i < 8*sizeof(n); i++)
    if (n & (1 << i))
      count++;
  return count;
}
```

Naredna mogućnost bi bila da se uvek ispituje poslednji bit broja, a da se u svakoj iteraciji broj pomera za jedno mesto u desno.

```
int bit_count(int n) {
  int i;
  int count = 0;
  for (i = 0; i < 8*sizeof(n); i++, n >>= 1)
    if (n & 1)
      count++;
}
```

Ukoliko je broj neoznačen, nije potrebno vršiti sve iteracije već je moguće zaustaviti se u trenutku kada broj postane nula.

```
int bit_count(unsigned n) {
  int count = 0;
  while(n) {
    if (n & 1) count++;
    n >>= 1;
  }
  return count;
}
```

Ukoliko broj `n` nije nula, izraz `n & (n-1)` invertuje poslednju jedinicu u njegovom binarnom zapisu (ostale cifre ostaju nepromenjene). Zaista, ukoliko je broj `n` oblika `...10000`, u slučaju potpunog komplementa broj `n-1` je oblika `...01111` pa je njihova konjunkcija oblika `...00000`, pri čemu je prefiks (označen sa `...`) nepromenjen. Imajući ovo u vidu, moguće je implementirati prethodnu funkciju tako da broj iteracija bude jednak broju jedinica u broju `n`.

```
int bit_count(unsigned n) {
  int count = 0;
  while(n) {
    count++;
    n = n & (n-1);
  }
  return count;
}
```

Ovo je moguće iskoristiti kako bi se ispitalo da li je broj stepen broja dva. Naime, broj je stepen broja dva ako i samo ako ima tačno jednu jedinicu u binarnom zapisu (specijalan slučaj nule se mora posebno obraditi).

```
int is_pow2(unsigned n) {
  return n != 0 && !(n & (n - 1));
}
```

Asimptotski brše rešenje za problem brojanja bitova moguće je dobiti ako se primeni tzv. *paralelno brojanje bitova*. Ideja je posmatrati broj kao niz od 16 uzastopnih parova jednobitnih brojeva i zatim sabiranjem svakog para dobiti 16 uzastopnih parova dvobitnih brojeva koji sadrše njihove sume. Ovo je moguće uraditi izdvajanjem bitova na neparnim pozicijama (konjunkcijom sa maskom `0x55555555`), zatim pomeranjem broja za jedno mesto udesno i ponovnim izdvajanjem bitova na neparnim pozicijama (ono što se dobije su bitovi na parnim pozicijama polaznog broja pomereni za jednu poziciju desno) i zatim primenom operatora sabiranja. Na primer, ukoliko je polazni broj

`01011011001010100110110101011011`,

sabiraju se brojevi

`01 01 00 01 00 00 00 00 01 00 01 01 01 01 00 01` i

`00 00 01 01 00 01 01 01 00 01 01 00 00 00 01 01`.

Dobija se rezultat

`01 01 01 10 00 01 01 01 01 01 10 01 01 01 01 10`

Rezultat ukazuje da se u svakom od prva tri para bitova polaznog broja nalazi po jedna jedinica, da se u sledeće paru nalaze dve jedinice, zatim ide par sa nula jedinica itd.

Prilikom ovog sabiranja „paralelno" se sabiraju svi parovi pojedinačnih bitova (s obzirom da je svaki bit na parnim pozicijama arugmenata 0 sabiranja pojedinih parova su nezavisna u smislu da ne može da nastupi prekoračenje koje bi se sa jednog para proširilo na sledeći).

Ovaj postupak se dalje nastavlja tako što se dalje sabira 8 parova dvobitnih brojeva i dobiti 8 četvorobitnih brojeva koji čuvaju brojeve jedinica u svakoj četvorki bitova polaznog broja. U prethodnom primeru se dobija

`0010 0011 0001 0010 0010 0011 0010 0011`

što ukazuje na to da prva četvorka bitova polaznog broja ima dve jedinice, druga tri jedinice, treća 4, itd.

Zatim se ova 4 para četvorobitnih brojeva sabiraju i dobijaju se četiri osmobitna broja koja sadrže brojeve jedinica u pojedinačnim bajtovima polaznog broja. U prethodnom primeru se dobija

`00000101 00000011 00000101 00000101`

što govori da prvi bajt ima 5 jedinica, drugi 3, treći 5 i četvrti 5.

Zatim se ova dva para osmobitnih brojeva sabiraju i dobijaju se dva šesnaestobitna broja koji sadrže brojeve jedinica u dvobajtima polaznog broja.U prethodnom primeru se dobija se

`0000000000001000 0000000000001010`

što govori da prvi dvobajt ima 8 jedinica, a drugi 10.

Napokon, sabira se i ovaj par šesnaestobitnih brojeva i dobija se tridesetdvobitni broj koji sadrži konačan rezultat.

`00000000000000000000000000010010`

Prethodni postupak implementira funkcija:

```
int bit_count(unsigned n) {
   n  = (n & 0x55555555) + ((n >> 1) & 0x55555555);
   n  = (n & 0x33333333) + ((n >> 2) & 0x33333333);
   n  = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
   n  = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF);
   n  = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF);
   return n;
}
```

Još jedna od mogućih ideja je napraviti statički niz u kome je preračunat broj bitova za svaku od 256 vrednosti bajta, a zatim čitati podatke iz tog niza.

```
int bit_count(unsigned n) {
  static int lookup[256] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
  return lookup[n & 0xFF] + lookup[n >> 8 & 0xFF] +
         lookup[n >> 16 & 0xFF] + lookup[n >> 24 & 0xFF];
}
```

Izdvajanje bitova. Naredna funkcija vraća `n` bitova broja `x` koji počinju na poziciji `p` (pozicije se broje od 0 s desna).

```
unsigned get_bits(unsigned x, int p, int n) {
   /* Gradimo masku koja ima poslednjih n jedinica:
         0000000...00011111 */
   unsigned last_n_1 = ~(~0 << n);
   /* x pomerimo u desno za odgovarajuci broj mesta, a zatim
      konjunkcijom sa  maskom obrisemo pocetne cifre */
   return (x >> p+1-n) & last_n_1;
}
```

Izmena bitova. Naredna funkcija vraća broj koji se dobija od broja `x` izmenom `n` bitova pocevši od pozicije `p`, tako što se na ta mesta upisuje poslednjih `n` bitova broja y.

```
unsigned set_bits(unsigned x, int p, int n, unsigned y) {
   /* Maska 000000...000111111  - poslednjih n jedinica */
   unsigned last_n_1 = ~(~0 << n);
   /* Maska 1111100..000111111 - n nula pocevsi od pozicije p */
   unsigned middle_n_0 = ~(last_n_1 << p+1-n);
   /* Brisemo n bitova pocevsi od pozicije p */
   x = x & middle_n_0;
   /* Izdvajamo poslednjih n bitova broja y i
      pomeramo ih na poziciju p */
   y = (y & last_n_1) << p+1-n;
   /* Upisujemo bitove broja y u broj x i vracamo rezultat */
   return x | y;
}
```

Rotacija bitova. Naredna funkcija vrsi rotaciju neoznačenog broja `x` za `n` pozicija u desno.

```
unsigned right_rotate(unsigned x, int n) {
   int i, wl = sizeof(unsigned)*8;
   /* Postupak se ponavlja n puta */
   for (i = 0; i < n; i++) {
      /* Poslednji bit broja x */
      unsigned last_bit = x & 1;
      /* x pomeramo za jedno mesto u desno */
      x >>= 1;
      /* Zapamceni poslednji bit stavljamo na pocetak broja x*/
      x |= last_bit<<wl-1;
   }
   return x;
}
```

Refleksija bitova. Naredna funkcija obrće binarni zapis neoznačenog broja `x` tako što bitove „čita unatrag".

```
unsigned mirror(unsigned x) {
   int i, wl = sizeof(unsigned)*8;
   /* Rezultat inicijalizujemo na poslednji bit broja x */
   unsigned y = x & 1;
   /* Postupak se ponavlja wl-1 puta */
   for (i = 1; i<wl; i++) {
      /* x se pomera u desno za jedno mesto */
      x >>= 1;
      /* rezultat se pomera u levo za jedno mesto */
```

```
        y <<= 1;
        /* Poslednji bit broja x upisujemo na kraj rezultata */
        y |= x & 1;
    }
    return y;
}
```

Pitanja i zadaci za vežbu

Pitanje 6.69. *Ako je promenljiva tipa* `unsigned int`*, kada izraz* `n & (n-1)` *ima vrednost* `0`*?*

Pitanje 6.70. *Ukoliko broj* `n` *nije nula, čemu je jednaka vrednsot izraza* `n & (n-1)` *?*

Pitanje 6.71. *Koju vrednost ima izraz* `0xA3 & 0x24 << 2` *?*

Pitanje 6.72. *Koju vrednost ima izraz* `0xB8 | 0x51 << 3` *?*

Pitanje 6.73. *Koju vrednost ima izraz* `0x12 ^ 0x34` *?*

Pitanje 6.74. *Koju vrednost ima izraz* `0x34 << 2` *?*

Pitanje 6.75. *Koju vrednost ima izraz* `(13 & 17) << 2` *?*

Pitanje 6.76. *Napisati izraz čija je vrednost broj koji se dobije invertovanjem poslednjeg bajta broja* `x`*?*

Pitanje 6.77. *Napisati izraz koji vraća treći bit najmanje težine celog broja* `n`*.*

Pitanje 6.78. *Napisati izraz (ne funkciju!) koji je jednak vrednosti* `c` *tipa* `unsigned char` *u kojoj su promenjenje vrednosti tri poslednja bita (tri bita najmanje težine).*

Pitanje 6.79. *Napisati izraz čija je vrednost broj koji se dobija tako što se upišu sve jedinice u poslednji bajt u zapisu neoznačenog celog broja* `x`*.*

Pitanje 6.80. *Neka su* `x` *i* `y` *tipa* `unsigned char`*. Napisati izraz koji je tačan ako i samo ako su različiti poslednji bit vrednosti* `x` *i početni bit vrednosti* `y` *(bitovi se broje zdesna nalevo).*

Pitanje 6.81. *Neka su* `x` *i* `y` *tipa* `unsigned char`*. Napisati izraz koji je tačan ako i samo ako su jednake vrednosti bitova na trećem bitu najmanje težine (treći bit zdesna, broji se i nulti bit) za* `x` *i* `y`*.*

Pitanje 6.82. *Neka su* `x` *i* `y` *tipa* `unsigned char`*. Napisati izraz koji je tačan ako i samo ako su razlicite vrednosti bitova na osmom bitu najmanje težine (osmi bit zdesna) za* `x` *i* `y`*.*

Pitanje 6.83. *Neka su* `x` *i* `y` *tipa* `unsigned char`*. Napisati izraz koji je tačan ako i samo ako su jednake vrednosti bitova na trećem bitu najmanje težine (treći bit zdesna) za* `x` *i* `y`*.*

Pitanje 6.84. *Ako je* `c` *tipa* `unsigned char`*, napisati izraz koji vraća vrednost u kojoj je na poslednjem bitu (zdesna nalevo) odgovarajući bit vrednosti* `c`*, a svi ostali bitovu su jednaki* `0`*.*

Pitanje 6.85. *Ako je* `c` *tipa* `unsigned char`*, napisati izraz koji vraća vrednost u kojoj je na pretposlednjem bitu (zdesna nalevo) odgovarajući bit vrednosti* `c`*, a svi ostali bitovu su jednaki* `1`*.*

Pitanje 6.86. *Neka je* `x` *32-bitan neoznačeni ceo broj, a* `b` *neoznačeni karakter. Izraz kojim se* `b` *inicijalizuje na vrednost bajta najveće težine broja* `x` *je* _____.

Pitanje 6.87. *Napisati funkciju* `f` *sa argumentom* `c` *tipa* `unsigned char`*, koja vraća vrednost* `c` *u kojoj su razmenjene vrednosti prvog i poslednjeg bita.*

Pitanje 6.88. *Napisati funkciju koja invertuje* `n` *bitova počevši od pozicije* `p` *u broju* `x`*.*

Pitanje 6.89. *Napisati funkciju koja izračunava razliku suma bitova na parnim i neparnim pozicijama broja x.*

Zadatak 6.5.1. *Napisati funkciju* `f` *sa argumentom* `c` *tipa* `unsigned char`*, koja vraća vrednost* `c` *u kojoj su zamenjene vrednosti poslednjeg i pretposlednjeg bita (poslednji bit je bit najmanje težine).*

Zadatak 6.5.2. *Napisati funkciju* `unsigned int f6(unsigned int x)` *koja vraća broj dobijen izdvajanjem prvih 8 bitova broja (bitovi na najvećim tezinama), a ostatak broja popunjava jedinicama. Testirati pozivom u main-u.*

Zadatak 6.5.3. *Napisati funkciju unsigned int fja(unsigned int x) koja vraca broj dobijen od broja x kada se prvih 8 bitova broja (bitovi na najvecim tezinama) postave na 0, poslednjih 4 bita(bitovi na najmanjim tezinama) se postave na 0110, a ostatak broja ostaje nepromenjen. Testirati pozivom u main-u.*

Zadatak 6.5.4. *Sa standarnog ulaza unosi se ceo pozitivan broj i pozitivan broj* `p`*. Napisati funkciju* `unsigned int f3(unsigned int x,unsigned int p)` *koja menja mesta prvih i poslednjih* `p` *bitova (npr. p = 3 i broj x = 111....010 treba da se dobije izlaz x = 010...111)*

Zadatak 6.5.5. *Napisati funkciju koja pronalazi zbir parnih brojeva na neparnim pozicijama u nizu koji se zadaje kao argument komandne linije. Svi brojevi su pozitivni (ovo ne treba proveravati). Brojanje pozicija pocinje indeksom 0. Rezultat se ispisuje na standardni izlaz.*

Zadatak 6.5.6. *Napisati program koji u zavisnosti od toga da li je suma bitova broja parna ili neparna, šiftuje bitove broja za jedno mesto ulevo odnosno u desno.*

Zadatak 6.5.7. *Napisati funkciju* `int count_zero_pairs(unsigned x)` *koja broji koliko se puta kombinacija* 00 *(dve uzastopne nule) pojavljuje u binarnom zapisu celog neoznačenog broja (ako se u bitskoj reprezentaciji nekog broja jave tri uzastopne nule, središnja se broji dva puta, u levom i desnom paru). Napisati program koji za broj iz prvog argumenta komandne linije na standardni izlaz ispisuje rezultat funkcije.*

Zadatak 6.5.8. *Napisati funkciju* `unsigned int f3(unsigned int x)` *koja vraća broj koji predstavlja odraz u ogledalu polaznog broja* `x`*. Testirati pozivom u main funkciji – sa standardnog ulaza se unosi broj, rezultat ispisati na standarni izlaz. Na primer, ako je ulaz broj čiji je binarni zapis* `00101`*, izlaz je broj čiji je binarni zapis* `10100`*.*

Zadatak 6.5.9. *Napisati funkciju* `int f1(int x, int p, int q)` *koja kao argumente dobija 3 cela broja* `x`*,* `p`*,* `q`*, a kao rezultat vraća broj koji je jednak broju* `x` *kome je invertovan svaki drugi bit između pozicije* `p` *i* `q`*, čitano sa desna na levo. Ukoliko* `p` *ili* `q` *izlaze iz opsega ili* `p` $>$ `q` *vratiti* `-1` *kao rezultat.*

Glava 7

Fundamentalne strukture podataka

Dinamička alokacija memorije omogućava građenje specifičnih, dinamičkih struktura podataka koje su u nekim situacijama pogodnije od statičkih zbog svoje fleksibilnosti. Najznačajnije među njima su povezane liste i stabla (čiji elementi mogu da sadrže podatke proizvoljnog tipa).

7.1 Liste

Povezane liste su najjednostavnije samoreferišuće dinamičke strukture. Povezanu listu čine elementi koji sadrže podatke izabranog tipa i pokazivače. Svaki pokazivač pokazuje na jedan (sledeći) element liste i ti pokazivači povezuju elementi u jednu celinu — u listu.

7.1.1 Odnos lista, nizova i dinamičkih blokova

Povezana lista se po više pitanja razlikuje od (statički alociranog) niza. Lista se može proizvoljno smanjivati i proširivati (tj. broj njenih elemenata se može smanjivati i povećavati). Dodavanje elementa u listu zahteva vreme $O(1)$ (mada, zbog fragmentisanja memorije, konkretno vreme može da raste kako se program izvršava). Elementi niza su smešteni u uzastopnim memorijskim lokacijama i pozicija u memoriji i-tog elementa se može jednostavno izračunati na osnovu i i pozicije početnog elementa. Zbog toga se i-tom elementua niza pristupa u vremenu $O(1)$. S druge strane, elementi liste su smešteni u memorijskim lokacijama koje nisu nužno uzastopne i mogu biti razbacane po memoriji. Da bi se pristupilo jednom elementu liste, potrebno je krenuti od početnog elementa i pratiti pokazivače sve dok se ne naiđe na traženi element, te je vreme potrebno za pristup $O(n)$ (gde je n broj elemenata liste).

Povezana lista se po više pitanja razlikuje i od dinamički alociranih blokova memorije (koji mogu da sadrže nizove elemenata istog tipa). Alokacija dinamičkog bloka zahteva postojanje u memoriji povezanog bloka slobodne memorije (veličine dovoljne da primi zadati skup elemenata). S druge strane, korišćenje lista zahteva alociranje memorije samo za jedan po jedan element. Brisanje elemenata se takođe vrši pojedinačno (te česte dodavanja i brisanja elemenata liste dovode do fragmentisanja memorije). Veličina dinamičkog bloka se može menjati samo od njegovog kraja (a i to može da bude zahtevna operacija). Veličina liste se menja jednostavno dodavanjem novih pojedinačnih elemenata. Elementima u dinamičkom bloku se pristupa, kao elementina niza, u vremenu $O(1)$, a elementima liste u vremenu $O(n)$.

Sve u svemu — nijedna od navedenih struktura podataka (liste, nizovi, dinamički blokovi) nije uvek najbolji izbor i nema svojstva uvek bolja od druga dva. Najbolji izbor je vezan za specifičnosti konkretnog problema i najvažnije zahteve.

7.1.2 Elementi liste

Tip elementa (ili čvora) liste se obično definiše na sledeći način:

```
struct cvor_liste
{
  <type> podatak;
```

```
  struct cvor_liste *sledeci;
};
```

U navedenoj definiciji, `<type>` označava tip podatka koji element sadrži. To može biti proizvoljan tip (moguće i struktura), a element može da sadrži i više od jednog podatka (i, naravno, ti podaci ne moraju da budu istog tipa). Član strukture `struct cvor_liste *sledeci;` je dozvoljen u okviru same definicije strukture `struct cvor_liste`, jer se zna koliko memorije zahteva (onoliko koliko zahteva bilo koji pokazivač).

Koristeći `typedef` može se uvesti ime strukture koje ne sadrži ključnu reč `struct` (element `sledeci` ipak mora da se deklariše tipom `struct cvor_liste`):

```
typedef struct cvor_liste
{
  <type> podatak;
  struct cvor_liste *sledeci;
} cvor;
```

Sâma liste se ne definiše kao struktura podataka. Umesto toga, listi se obično pristupa preko njenog kraja i/ili početka. Zato se njen kraj i/ili početak moraju čuvati jer omogućavaju pristup svim elementima liste.

7.1.3 Stek

Stek (eng. stack) je struktura koja funkcioniše na principu LIFO (*last in, first out*). Stek ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

push dodaje element na vrh steka;

pop skida element sa vrha steka.

Kao stek ponaša se, na primer, šipka na koju su naređani kompakt diskovi. Ako sa štapa može da se skida samo po jedan disk, da bi bio skinut disk koji je na dnu — potrebno je pre njega skinuti sve druge diskove.

Stek se može implementirati kao povezana lista. Osnovne operacije nad stekom implementiranom kao povezana lista ilustrovane su na slici 7.1.

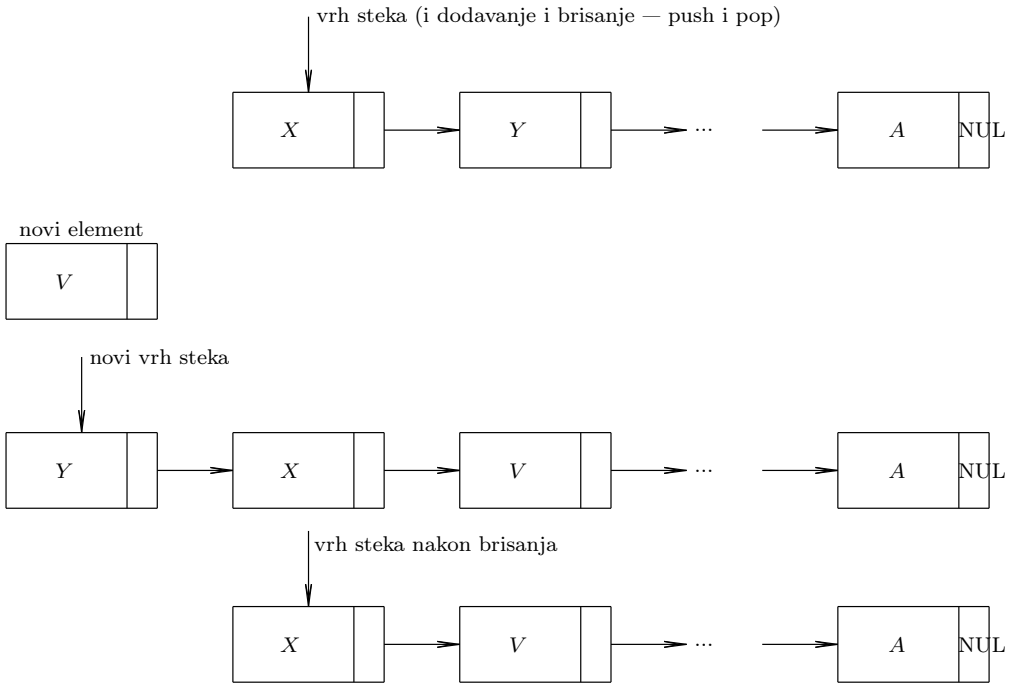
Ako je element liste definisan na sledeći način (čvor liste, pored pokazivača na sledeći, sadrži jedan podatak tipa `int`):

```
typedef struct cvor_liste
{
  int broj;
  struct cvor_liste *sledeci;
} cvor;
```

onda se funkcija `push` može implementirani na sledeći način:

```
cvor* push(cvor* vrh, int broj)
{
  cvor* novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->broj = broj;
  novi->sledeci = vrh;
  return novi;
}
```

Navedena funkcija kao argumente ima tekući vrh steka i broj koji treba da bude sadržaj novog čvora. Ukoliko alokacija novog čvora ne uspe vraća se `NULL`. Inače, u novi čvor upisuje se zadati broj, novi čvor



Slika 7.1: Osnovne operacije nad stekom implementiranom kao povezana lista

pokazuje na vrh i vraća se kao novi vrh steka. `novi->broj` je kraći zapis za `(*novi).broj`. (isto važi za bilo koji pokazivač na strukturu, dakle umesto `(*a).member` može se pisati `a->member`).

U navedenoj implementaciji, novi vrh steka se vraća kao povratna vrednost funkcije. Alternativa je da se kao argument prosleđuje pokazivač na vrh liste (koji se može promeniti u okviru funkcije):

```
int push(cvor** vrh, int broj)
{
  cvor* novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return -1;
  novi->broj = broj;
  novi->sledeci = *vrh;
  *vrh = novi;
  return 0;
}
```

Funkcija `pop` može se implementirati na sledeći način:

```
int pop(cvor** vrh, int *broj)
{
  cvor *tmp;
  if (*vrh == NULL)
    return -1;

  tmp = *vrh;
  *broj = tmp->broj;
  *vrh = tmp->sledeci;
  free(tmp);
  return 0;
```

```
}
```

Navedena funkcija vraća (kroz argument) broj upisan u element koji je vrh steka i briše ga i vraća vrednost `0`, ako stek nije prazan. Ako je stek prazan, vraća se vrednost `-1`.

Sadržaj steka se može ispisati sledećom funkcijom:

```
void ispisi_elemente_steka(cvor* vrh)
{
  cvor* t;
  for (t = vrh; t != NULL; t=t->sledeci);
    printf("%d ", t->broj);
}
```

ili njenom rekurzivnom verzijom:

```
void ispisi_elemente_steka(cvor* vrh)
{
  if(vrh != NULL) {
    printf("%d ", vrh->broj);
    ispisi_elemente_steka(vrh->sledeci);
  }
}
```

Rekurzivna verzija se može (za razliku od iterativne) lako prepraviti da ispisuje elemente steka u obratnom poretku:

```
void ispisi_elemente_steka(cvor* vrh)
{
  if(vrh != NULL) {
    ispisi_elemente_steka(vrh->sledeci)
    printf("%d ", vrh->broj);
  }
}
```

Oslobađanje steka (tj. brisanje svih njegovih elemenata) može se izvršiti sledećom funkcijom:

```
void obrisi_stek(cvor* vrh)
{
   while (vrh)  {
      cvor* tmp = vrh->sledeci;
      free(vrh);
      vrh = tmp;
   }
}
```

ili njenom rekurzivnom verzijom:

```
void obrisi_stek(cvor* vrh)
{
  if (vrh) {
     obrisi_stek(vrh->sledeci);
     free(vrh);
  }
}
```

Sledeći kôd ilustruje upotrebu navedenih funkcija za rad sa stekom (`...` označava kôd gore navedenih funkcija):

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

...

void main()
{
    int i;
    cvor* vrh= NULL;

    for (i = 0; i<5; i++)
      if (push(&vrh, i)==-1) {
        obrisi_stek(vrh);
        return;
      }

    ispisi_elemente_steka(vrh);
    printf("\n");

    while (pop(&vrh, &i)!=-1)
      printf("%d ", i);
}
```

U ovako implementiranom steku moguće je dodavanje elementa na bilo kojem mesto, na primer, na dno steka. Međutim, ta operacija je složenosti $O(n)$ (gde je n broj elemenata steka), za razliku od operacije `push` koja ima složenost $O(1)$.

```
int ubaci_na_dno_steka(cvor** vrh, int broj)
{
  cvor* novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return -1;
  novi->sledeci = NULL;
  novi->broj = broj;

  if (*vrh == NULL)
    *vrh=novi;

  cvor* t;
  for (t = *vrh; t->sledeci ; t=t->sledeci)
    ;
  t->sledeci = novi;
}
```

U nastavku je data rekurzivna verzija iste funkcije:

```
int ubaci_na_dno_steka(cvor** vrh, int br)
{
  if (*vrh == NULL) {
    cvor* novi = (cvor*)malloc(sizeof(cvor));
    if (novi == NULL)
      return -1;
    novi->sledeci = NULL;
    novi->broj = broj;
  }

  return ubaci_na_dno_steka(&vrh->sledeci, broj);
}
```

7.1.4 Red

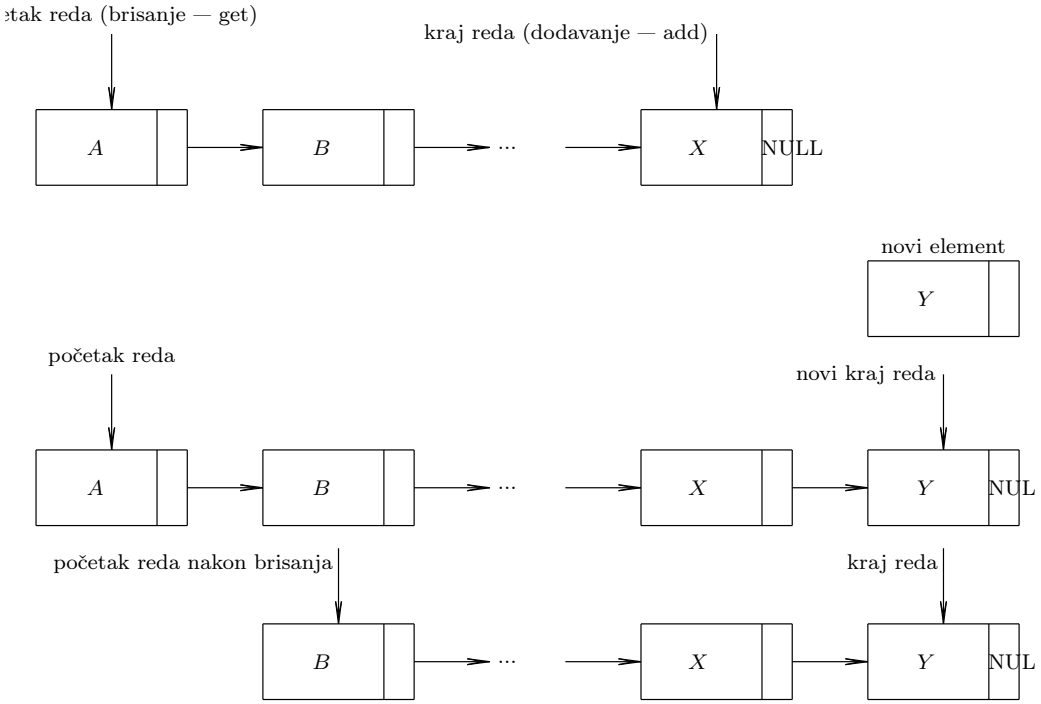
Red (eng. queue) je struktura koja funkcioniše na principu FIFO (*first in, first out*). Red ima sledeće dve osnovne operacije (koje treba da budu složenosti $O(1)$):

add dodaje element na kraj reda;

get skida element sa početka reda.

Red se može ilustrovati i na primereu čekaonice u kojoj svako zna ko je na redu posle njega. U red se dodaje na kraj, a prvi iz reda izlazi onaj ko je prvi došao.

Red se može implementirati kao povezana lista. Osnovne operacije nad redom implementiranom kao povezana lista ilustrovane su na slici 7.2.



Slika 7.2: Osnovne operacije nad redom implementiranom kao povezana lista

.

Ako je element liste definisan na sledeći način (čvor liste, pored pokazivača na sledeći, sadrži jedan podatak tipa `int`):

```
typedef struct cvor_liste
{
  int broj;
  struct cvor_liste *sledeci;
} cvor;
```

onda se funkcija `add` može implementirani na sledeći način:

```
int add(cvor** pocetak, cvor** kraj, int broj)
{
  cvor* novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return -1;
```

```
  novi->broj = broj;
  novi->sledeci = NULL;

  if(*pocetak==NULL && *kraj==NULL) {
    *pocetak=novi;
    *kraj=novi;
  }
  else {
    (*kraj)->sledeci = novi;
    (*kraj) = novi;
  }
  return 0;
}
```

U navedenoj funkciji, ukoliko red nema nijedan element, onda će i pokazivač na početak da nakon funkcije ukazuje na novi element.

```
int get(cvor** pocetak, cvor** kraj, int *broj)
{
  cvor *tmp;
  if (*pocetak== NULL)
    return -1;

  tmp = *pocetak;
  *broj = tmp->broj;
  *pocetak= tmp->sledeci;
  if(*pocetak==NULL)
    *kraj=NULL;
  free(tmp);
  return 0;
}
```

U navedenoj funkciji, ukoliko je red imao samo jedan element, onda će na kraju izvršavanja funkcije i pokazivač na početak i pokazivač na kraj reda da ukazuju na `NULL`.

Ispis elemenata i brisanje elemenata vrši se analogno kao kod steka.

Sledeći kôd ilustruje upotrebu navedenih funkcija za rad sa redom (`...` označava kôd gore navedenih funkcija):

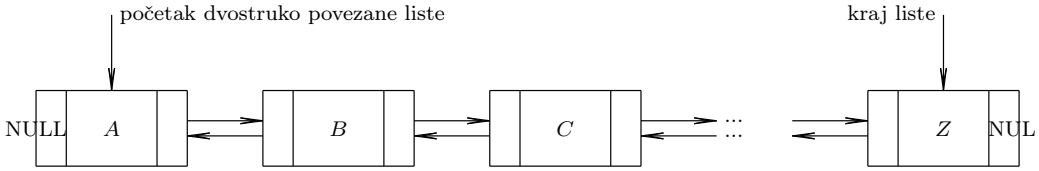
```
#include <stdio.h>
#include <stdlib.h>

...

void main()
{
    int i;
    cvor* pocetak=NULL;
    cvor* kraj=NULL;

    for (i = 1; i<5; i++)
      if (add(&pocetak, &kraj, i)==-1) {
        obrisi_red(pocetak);
        return;
      }

    ispisi_elemente_reda(pocetak);
    printf("\n");
```

```
    while (get(&pocetak, &kraj, &i)!=-1)
      printf("%d ", i);
}
```

I u red i u stek moguće je umetnuti element na bilo koje mesto u listi, ali ta operacija, naravno, više nije složenosti $O(1)$. Za tu operaciju, ako je potrebno umetnuti element e nakon elementa e', najpre se pamti pokazivač koji sadrži e', zatim se e' usmerava da ukazuje na e, a e na zapamćenu lokaciju. Slično se briše proizvoljni element liste.

7.1.5 Dvostruko povezane (dvostruko ulančane) liste

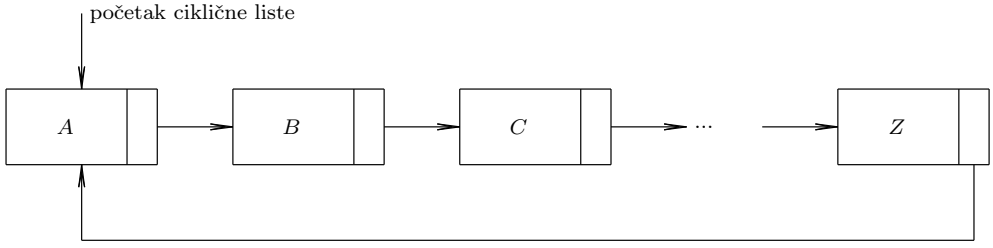
Svaki element u (jednostruko) povezanoj listi ima jedan pokazivač — pokazivač na sledeći element liste. Zato je listu jednostavno obilaziti u jednom smeru ali je vremenski zahtevno u suprotnom. U dvostruko povezanoj listi, svaki element sadrži dva pokazivača — jedan na svog prethodnika a drugi na svog sledbenika. Dvostruko povezana lista omogućava jednostavno kretanje unapred i unazad kroz listu. U dvostruko povezanoj listi brisanje elementa sa oba kraja liste zahteva vreme $O(n)$. Dvostruka lista ilustrovana je na slici 7.3.



Slika 7.3: Dvostruko povezana lista

7.1.6 Kružne (ciklične, cirkularne) liste

U kružnoj listi, poslednji element liste ukazuje na prvi element liste. To omogućava da se do bilo kog elementa može doći pošavši od bilo kog drugog elementa. Zbog toga, izbor "prvog" i "poslednjeg" elementa je u kružnoj listi relativan. Kružna lista ilustrovana je na slici 7.4.



Slika 7.4: Kružna lista

.

Pitanja i zadaci za vežbu

Pitanja

Pitanje 7.1. *Koliko najmanje bajtova može da zauzima jedan element povezane liste (obrazložiti odgovor)?*

Pitanje 7.2. *Napisati rekurzivnu funkciuju* `void obrisi(cvor *l)` *koja briše sve elemete jednostruko povezane liste.*

Pitanje 7.3. *Ukoliko se znaju pokazivači na prvi i poslednji element jednostruko povezane liste, složenost operacije brisanja prvog elementa je _____, a složenost operacije brisanja poslednjeg elementa je _____.*

Pitanje 7.4. *Poznata je adresa početnog čvora jednostruko povezane liste. Složenost operacije uklanjanja njenog poslednjeg čvora, ako nije poznata njegova adresa je _____, a ako je poznata njegova adresa je _____.*

Pitanje 7.5. *Koja apstraktna struktura podataka može da modeluje lift (iz lifta ljudi obično izlaze u obratnom redosledu od redosleda ulaska)? _____.*

Pitanje 7.6. *Koje su osnovne operacije steka i koja je njihova složenost u standardnoj implementaciji baziranoj na listama?*

Pitanje 7.7. *Koje su složenosti operacije dodavanja na dno steka i dodavanja na kraj reda?*

Pitanje 7.8. *Iz lifta ljudi uobičajeno izlaze u obratnom redosledu od onoga u kojemu su ušli, te lift predstavlja _____.*

Pitanje 7.9. *Stek je (zaokružiti sve tačne odgovore):* *LIFO* *FILO* *FIFO* *LILO*

Pitanje 7.10. *Ako su s_1 i s_2 prazni stekovi, šta je njihova vrednost nakon operacija $push(s_1, 1)$, $push(s_2, 2)$, $push(s_1, 3)$, $push(s_2, pop(s_1))$?*

Pitanje 7.11. *Koje su osnovne operacije strukture stek i koja je njihova složenost (za implementaciju na bazi lista)?*

Pitanje 7.12. *Dopuniti implementaciju funkcije* `push` *koja dodaje element na vrh steka:*

```
int push(cvor** vrh, int broj)  {
  cvor* novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)

     ----------------------

  ----------------------
  novi->sledeci = *vrh;
  *vrh = novi;

  ----------------------
}
```

Pitanje 7.13. *Dopuniti implementaciju funkcije* `pop` *koja skida element na vrh steka:*

```
int pop(cvor** vrh, int *broj)
{
  cvor *tmp;
  if (*vrh == NULL)
    return -1;
  tmp = *vrh;
  ----------------------------
  ----------------------------
```

```
  free(tmp);
  return 0;
}
```

Pitanje 7.14. *Red je (zaokružiti sve tačne odgovore):* *LIFO* *FILO* *FIFO* *LILO*

Pitanje 7.15. *Ukoliko je stuktura red implementirana korišćenjem povezanih listi, red karakterišu: pokazivač na početak reda i* _____.

Pitanje 7.16. *Koje su osnovne operacije strukture red i koja je njihova složenost (za implementaciju na bazi lista)?*

Pitanje 7.17. *Dopuniti implementaciju funkcije* `get` *koja čita i briše element sa kraja reda:*

```
int get(cvor** pocetak, cvor** kraj, int *broj) {
  cvor *tmp;
  if (*pocetak== NULL)
    return -1;
  tmp = *pocetak;
  *broj = tmp->broj;
  *pocetak= tmp->sledeci;

  -----------------

  -----------------
  free(tmp);
  return 0;
}
```

Pitanje 7.18. *Opisati dinamičku strukturu kružna lista.*

Definisati strukturu čvora kružne jednostruko povezane liste.

Pitanje 7.19. *Navesti primer strukture koja opisuje elemente dvostruko povezane liste.*

Pitanje 7.20. *Složenost operacije brisanja poslednjeg elementa jednostruko povezane liste je* _____, *a dvostruko povezane liste je* _____.

Pitanje 7.21. *Ukoliko se znaju pokazivači na prvi i poslednji element jednostruko povezane liste, složenost operacije brisanja prvog elementa je* _____, *a složenost operacije brisanja poslednjeg elementa je* _____.

Zadaci

Zadatak 7.1.1. *Napisati program koji formira listu od niza celih brojeva koji se unose sa standardnog ulaza. Oznaka za kraj unosa je* 0. *Napisati funkcije za formiranje čvora liste, ubacivanje elemenata na kraj liste, ispisivanje elemenata liste i oslobaanje liste i u programu demonstrirati pozive ovih funkcija.*

Zadatak 7.1.2. *Napisati funkciju koja sažima listu tako što izbacuje svaki element koji se više puta pojavljuje u listi.*

Zadatak 7.1.3. *Napisati funkciju* `cvor* f2(cvor* L)` *koja dobija glavu liste L kao argument, obrće listu L i vraća novu glavu.*

Zadatak 7.1.4. *Data je lista. Svaki čvor liste opisan je strukturom:*

```
typedef struct CVOR
{
    int vr;
    struct CVOR* sled;
}cvor;
```

Napisati funkciju `void udvaja(cvor* lista, int br)` *koja udvaja svako pojavljivanje elementa* `br` *u listi* `lista`.
Npr. za listu `1->7->6->7->1->4->7` *i br = 7 potrebno je dobiti listu:* `1->7->7->6->7->7->1->4->7->7`.

Zadatak 7.1.5. *Napisati funkciju* `void f4(cvor* lista)` *koja iz liste izbacuje svaki drugi element. U listi se nalaze celi brojevi. Lista se unosi sa standardnog ulaza sve dok se ne unese 0.*

Zadatak 7.1.6. *Napisati funkciju* `void umetni(cvor* lista)` *koja izmedju svaka dva el- ementa u listi umece element koji predstavlja razliku susedna dva.*

Zadatak 7.1.7. *Napisati funkciju* `Cvor* izbaci(Cvor *lista1, Cvor *lista2)` *koja iz* `liste1` *izbacuje sve elemente koji se pojavljuju u* `listi2`*. Testirati funkciju pozivom u main-u, sa standarnog ulaza se učitaju elementi prve liste sve dok se ne unese 0. Potom se učitavaju elementi druge liste sve dok se ne učita 0. Elemente liste dodavati na kraj. Potom pozvati funkciju i novodobijenu listu ispisati na standarni izlaz. Dozvoljeno je pravljenje nove liste.*

Zadatak 7.1.8. *Napisati program koji formira sortiranu listu od celih brojeva koji se unose sa standardnog ulaza. Oznaka za kraj unosa je* 0*.*

Napisati funkcije za:

- *Formiranje čvora liste,*
- *Ubacivanje elementa u već sortiranu listu,*
- *Ispisivanje elemenata liste u rastućem poretku u vremenu* $O(n)$*,*
- *Ispisivanje elemenata liste u opadajućem poretku u vremenu* $O(n)$*,*
- *Oslobaạnje liste.*

Napomena: potrebno je da lista bude takva da funkcije za ispis liste u rastućem i u opadajućem poretku ne koriste rekurziju niti dodatnu alociranu memoriju *a rade u vremenu* $O(n)$*.*

Zadatak 7.1.9. *Sa standardnog ulaza unosi se lista celih brojeva dok se ne unese 0. Odrediti broj pojavljivanja datog broja u listi.*

U datoteci `liste.h` *nalaze se funkcije za rad sa listom:*

```
void oslobodi(cvor* lista)
cvor* ubaci_na_pocetak(cvor* lista, int br)
cvor* novi_cvor(int br)
void ispis(cvor* lista)
```

Napraviti main.c u kome se testira rad programa. U main.c treba da stoji `#include "liste.h"`*.*
Kompaliranje: `gcc main.c liste.c`
Pokretanje: `./a.out`

Zadatak 7.1.10. *Napisati program koji implementira stek pomoću jednostruko povezane liste. Sa standardnog ulaza se unose brojevi koje smeštamo u stek sa nulom (*0*) kao oznakom za kraj unosa. Napisati funkcije za:*

- *Kreiranje elementa steka;*
- *Ubacivanje elementa na početak steka;*
- *Izbacivanje elementa sa početka steka;*
- *Ispisivanje steka;*
- *Oslobaạnje steka.*

Zadatak 7.1.11. *Napisati program koji implementira red pomoću jednostruko povezane liste (čuvati pokazivače i na početak i na kraj reda). Sa standardnog ulaza se unose brojevi koje smeštamo u red sa nulom (*0*) kao oznakom za kraj unosa. Napisati funkcije za:*

- *Kreiranje elementa reda;*
- *Ubacivanje elementa na kraj reda;*
- *Izbacivanje elementa sa početka reda;*
- *Ispisivanje reda;*
- *Oslobaạnje reda.*

Pitanje 7.22. *Definisana je struktura* `struct cvor { int x; struct cvor * sl; }` *koja opisuje elemente jednostruko povezane liste. Napisati kôd koji ubacuje novi čvor* `n` *iza čvora liste* `p` *(pretpostavlja se da su* `p` *i* `n` *različiti od* `NULL`*).*

```
void ubaci(struct cvor* n, struct cvor* p) {
 ...
}
```

Zadatak 7.1.12. *Napisati program koji simulira rad sa stekom. Napraviti funkcije push (za ubacivanje elementa u stek), pop (za izbacivanje elementa iz steka) i funkciju peek (koja na standardni izlaz ispisuje vrednost elementa koji se nalazi na vrhu steka).*

Zadatak 7.1.13. *Napisati program koji formira sortiranu listu od niza celih brojeva koji se unose sa standardnog ulaza. Oznaka za kraj unosa je* 0*. Napisati funkcije za formiranje čvora liste, ubacivanje elementa u već sortiranu listu, ispisivanje elemenata liste i oslobaạnje liste.*

Zadatak 7.1.14. *Sa ulaza čitamo niz reči (dozvoljeno je ponavljanje reči) i smeštamo ih u listu (koja osim reči čuva i broj pojavljivanja za svaku reč). Napisati funkciju koja sortira listu algoritmom* bubble sort *po broju pojavljivanja reči. Napisati funkciju koja ispisuje listu počevši od reči koja se pojavila najviše puta.*

Zadatak 7.1.15. *Napisati funkcije za ispisivanje elemenata reda: (a) redom, iterativno; (b) unatrag, rekurzivno. Elementi reda su tipa* `CVOR` *iz prethodnog zadatka.*

Zadatak 7.1.16. *Napraviti program za ažuriranje reda čekanja u studentskoj poliklinici. Treba obezbediti:*

- *funkciju koja sa standardnog ulaza čita broj indeksa studenta koji je došao (samo jedan ceo broj);*
- *funkciju koja ubacuje studenta (sa datim brojem indeksa) na kraj liste čekanja u vremenu* $O(1)$*;*
- *funkciju koja određuje sledećeg studenta za pregled i briše ga sa liste u vremenu* $O(1)$*;*
- *funkciju koja štampa trenutno stanje reda čekanja.*
- *funkciju koja oslobadja celu listu.*

7.2 Stabla

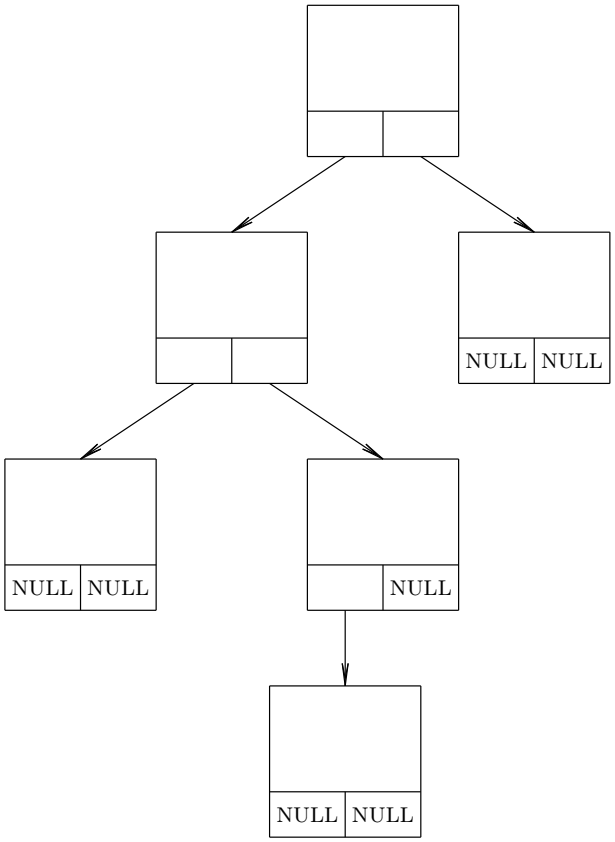
Stablo je struktura koja prirodno opisuje određene vrste hijerarhijskih objekata (npr. porodično stablo, logički izraz, aritmetički izraz, ...). Stablo se sastoji od čvorova i grana između njih. Svaka grana povezuje jedan čvor (u tom kontekstu — roditelj) sa njegovim detetom. Čvor koji se zove *koren stabla* nema nijednog roditelja. Svaki drugi čvor ima tačno jednog roditelja. *List* je čvor koji nema potomaka. Čvor A je predak čvora B ako je A roditelj čvora B ili ako je A predak roditelja čvora B. Koren stabla je predak svim čvorovima stabla (osim sebi). Zbog toga je moguće do bilo kog čvora stabla doći od korena (jedinstveno određenim putem).

7.2.1 Binarna stabla

Binarno stablo je stablo u kojem svaki čvor ima najviše dva deteta. Stablu se jednostavno dodaju novi čvorovi kao deca postojećih čvorova. Binarna stabla omogućavaju efikasnije obrade podataka u stilu binarne pretrage (jer su podaci ispod svakog čvora podeljeni na one koji su "levo"i one koji su đesno"). Međutim, ta efikasnost se gubi ako je binarno stablo degenerisano (svaki čvor ima npr. samo desnog potomka) ili skoro degenerisano. Postoje vrste binarnog stabla u kojima se garantuje da se visina levog i desnog podstabla u svakom čvoru razlikuju najviše za jedan.

Binarna stabla mogu se implementirati koristeći dinamičku alokaciju i pogodne strukture sa pokazivačima (slika 7.5). Na primer, struktura koja opisuje čvor binarnog stabla koji sadrži jedan podatak tipa `int` može se definisati na sledeći način:

```
typedef struct cvor_stabla
{
    int podatak;
    struct cvor_stabla *levo;
    struct cvor_stabla *desno;
} cvor;
```

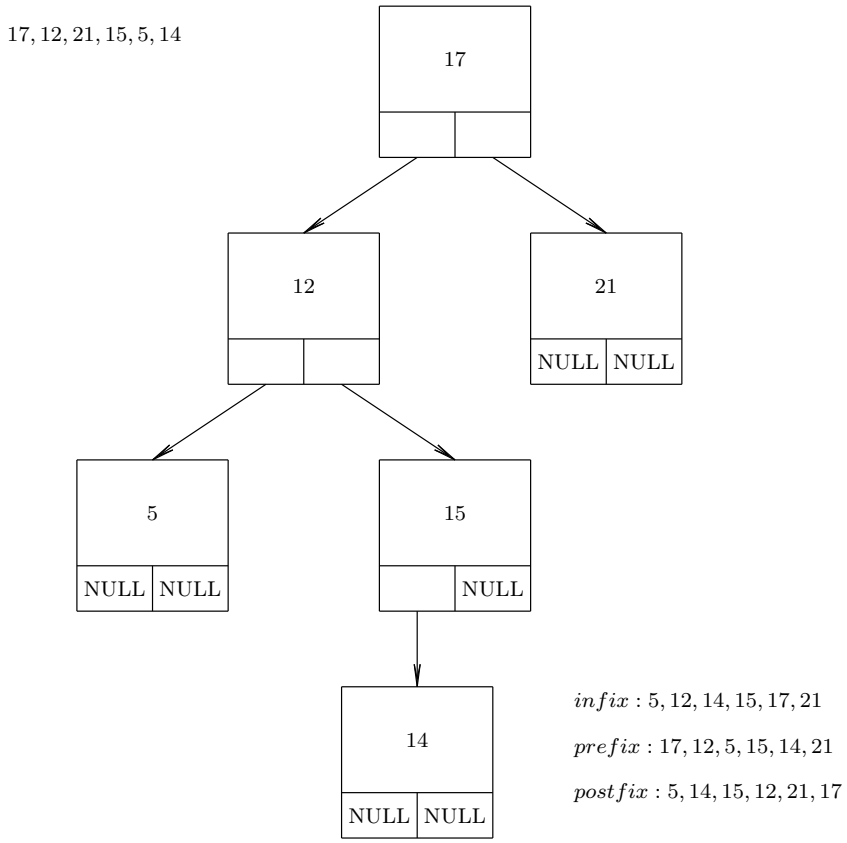


Slika 7.5: Stablo

7.2.2 Uređena binarna stabla

U uređenom binarnom stablu (naziva se i binarno pretraživačko stablo), za svaki čvor n, vrednosti svakog čvora iz njegovog levog podstabla su manje ili jednake od vrednost u čvoru n, a vrednosti svakog čvora iz njegovog desnog podstabla su veće od vrednost u čvoru n (ilustrovano na slici 7.6). U uređenom

stablu, novi čvor se dodaje jednostavno i to na jedinstven način. U uređenom stablu podaci su sortirani i jednostavno se mogu pretraživati.



Slika 7.6: Uređeno binarno stablo

Dodavanje elementa u binarno uređeno stablo

Binarno stablo se, od niza ulaznih vrednosti kreira jednostavno

- Kreiraj koren stabla i prvu vrednost smestiti u njega;
- Za svaku sledeću vrednost v:
  - Uporedi vrednost v sa vrednošću korena; ako je vrednost v manja ili jednaka od vrednosti korena, idi levo, inače idi desno;
  - Nastavi sa poređenjem vrednosti v sa vrednostima čvorova stabla sve dok ne dođeš do lista. Dodaj novi čvor na to mesto.

Naredna funkcija kreira čvor stabla koji sadrži zadatati podatak `vrednost` i umeće ga u binarno uređeno stablo sa korenom `**koren`. Ukoliko ne uspe kreiranje čvora vraća se ne-nula vrednost, inače se vraća nula. Prosleđeni koren može da se promeni u slučaju da je stablo inicijalno bilo prazno.

```
typedef enum {
OK,
NEUSPESNA_ALOKACIJA_MEMORIJE,
} status;
```

```
status dodaj_cvor_u_stablo(cvor** koren, int vrednost)
{
  cvor *tmp;
  cvor *novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return NEUSPESNA_ALOKACIJA_MEMORIJE;
  novi->levo = NULL;
  novi->desno = NULL;
  novi->podatak= vrednost;

  if(*koren==NULL) { /* ako je stablo bilo prazno */
    *koren=novi;
    return OK;
  }

  tmp = *koren;
  while(1) {
    if (vrednost <= tmp->podatak)
      if (tmp->levo == NULL) {
        tmp->levo = novi;
        return OK;
      }
      else
        tmp = tmp->levo;
    else
      if (tmp->desno == NULL) {
        tmp->desno = novi;
        return OK;
      }
      else
        tmp = tmp->desno;
   }
   return OK;
}
```

Naredni kôd je rekurzivna verzija dodavanja čvora u stablo:

```
int dodaj_cvor_u_stablo(cvor** koren, int vrednost)
{
  cvor *novi = (cvor*)malloc(sizeof(cvor));
  if (novi == NULL)
    return -1;
  novi->levo = NULL;
  novi->desno= NULL;
  novi->podatak= vrednost;

  if(*koren==NULL)
    *koren=novi;
  else
    dodaj_cvor_u_stablo_(*koren, novi);
  return 0;
}

void dodaj_cvor_u_stablo_(cvor *koren, cvor* novi)
{
  if (novi->podatak <= koren->podatak)
```

```
    if (koren->levo == NULL) {
      koren->levo= novi;
      return;
    }
    else
      dodaj_cvor_u_stablo_(koren->levo, novi);
  else
    if (koren->desno== NULL)  {
      koren->desno = novi;
      return;
    }
    else
      dodaj_cvor_u_stablo_(koren->desno, novi);
}
```

Pronalaženje elementa u binarnom uređenom stablu

Pretraživanje uređenog binarnog stabla (traženje čvora koji ima vrednost v) je jednostavno i slično umetanju elementa u stablo:

1. Ukoliko je tekuće stablo prazno, vrati rezultat da ne postoji traženi element u stablu.
2. Počni pretraživanje od korena stabla.
3. Ukoliko je v jednako vrendosti u tekućem čvoru stabla, vrati tekući čvor.
4. Ukoliko je v manje ili jednako vrendosti u tekućem čvoru stabla, pretraži levo podstablo.
5. Inače, pretraži desno podstablo.

Naredna funkcija implementira opisani način za pronalaženje elementa u stablu:

```
cvor *pronadji_cvor(cvor *koren, int vrednost)
{
  if (koren == NULL)
    return NULL;
  if (vrednost == koren->podatak)
    return koren;
  if (vrednost < koren->podatak)
    return pronadji_cvor(koren->levo, vrednost);
  return pronadji_cvor(koren->desno, vrednost);
}
```

Obilazak stabla u dubinu

Elementi binarnog uređenog stabla su uređeni i pogodnom obradom stabla mogu biti obiđeni u rastućem poretku. Najmanji element je "najlevlji", a najveći element "najdesniji". Obrada redom svih čvorova stabla naziva se *obilazak stabla*. Postoji nekoliko vrsta obilazaka:

- infiksni, od najmanjeg na najvećem: za svaki čvor se obilazi (rekurzivno) najpre njegovo levo podstablo, pa sâm taj čvor, pa njegovo desno podstablo (rekurzivno). Kratko to zapisujemo L-N-R (od *left-node-right*).
- infiksni, od najvećeg ka najmanjem: R-N-L.
- prefiksni: N-L-R
- postfiksni: L-R-N

Ispis elemenata binarnog stabla u infiksnom obilasku od najmanjeg ka najvećem može se implementirati na sledeći način:

```
void ispisi_stablo(cvor *p)
{
  if (p == NULL)
    return;
  ispisi_stablo(p->levo);
  printf("%d\n", p->podatak);
  ispisi_stablo(p->desno);
}
```

Obilazak stabla u širinu

U obilasku stabla u širinu, obilazi se (i obrađuje) nivo po nivo stavla. Obilazak u širinu se obično implementira korišćenjem strutkure red.

7.2.3 Izrazi u formi stabla

Matematički izrazi prirodno se mogu reprezentovati u vidu stabla. Na primer, izraz `3*(4+5)` može se reprezentovati kao stablo u čijem je korenu `*`, sa levim podstablom `3`, i sa desnim potomkom čvor `+`, koji ima potomke `4` i `5`. Ispisivanjem elemenata ovog stabla u infiksnom obilasku (sa dodatnim zagradama) dobija se izraz u uobičajenoj formi. Ispisivanjem elemenata ovog stabla u prefiksnom obilasku dobija se izraz zapisan u takozvanoj prefiksnoj poljskoj notaciji. Vrednost izraza reprezentovanog stablom može se izračunati jednostavnom funkcijom.

7.3 Grafovi

7.4 Heš tabele

Pitanja i zadaci za vežbu

Pitanja

Pitanje 7.23. *Ako binarno stablo ima dubinu* 9*, koliko najviše čvorova ono može da ima?*

Pitanje 7.24. *Ako binarno stablo ima 15 čvorova, koja je njegova najmanja moguća a koja najveća moguća dubina?*
(a) 3 i 14; (b) 3 i 16; (c) 5 i 14; (d) 5 i 16; (e) ne znam.

Pitanje 7.25. *Navesti definiciju strukture koja opisuje čvor binarnog stabla.*

Pitanje 7.26. *Šta je to uređeno binarno stablo?*

Pitanje 7.27. *U binarnom uređenom stablu sve vrednosti u čvorovima levog podstabla čvora X su manje ili jednake od vrednosti ...*

Pitanje 7.28. *Po analizi najgoreg slučaja, složenost dodavanja novog elementa u binarno uređeno stablo je _____. Po analizi najgoreg slučaja, slovženost pronalaženja elementa u binarnom uređenom stablu je _____.*

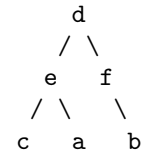
Pitanje 7.29. *Koja je složenost dodavanja elementa u binarno uredjeno stablo koje ima k elemenata, a n nivoa?* _____

Pitanje 7.30. *Koje vrste obilaska u dubinu binarnog stabla postoje?*

Pitanje 7.31. *Koja dodatna struktura podataka se koristi prilikom nerekurzivne implementacije obilaska stabla u u širinu?*

Pitanje 7.32. *Da li se postfiksnim ispisivanjem elemenata binarnog stabla dobija isti niz elemenata koji se dobija prefiksnim obilaskom, ali u obratnom poretku?* ....

Pitanje 7.33. *Dato je stablo:*



Prefiksni obilazak stabla (N-L-R): _____
Infiksni obilazak stabla (L-N-R): _____
Postfiksni obilazak stabla (L-R-N): _____
Obilazak stabla u širinu: _____

Pitanje 7.34. *Složenost operacije ispitivanja da li element pripada proizvoljnom uređenom binarnom stablu je:* _____ *a balansiranom uređenom binarnom stablu je:* _____.

Ukoliko je dato binarno stablo sa n čvorova, koja je složenost operacije konstruisanja istog takvog stabla? _____.

Pitanje 7.35. *Složenost operacije pronalaženja elementa u uređenom binarnom stablu zavisi od strukture stabla. Kada je ta složenost najmanja a kada najveća (i kolika je ona tada)?*

Pitanje 7.36. *Da li se postfiksnim ispisivanjem elemenata binarnog stabla dobija isti niz elemenata koji se dobija prefiksnim obilaskom, ali u obratnom poretku?* ....

Pitanje 7.37. *Ukoliko stablo sa korenom* `struct node* root` *reprezentuje izraz nad celim brojevima i nad operatorima $+, \cdot$ (na standardni način), napisati funkciju* `int izracunaj_vrednost(struct node* root)`.

Pitanje 7.38. *Kako se reprezentuju izrazi u vidu stabala?*

Pitanje 7.39.

Binarnim drvetom reprezentovati izraz `2+3*(4+2)`.

Pitanje 7.40. *Prikazati u vidu stabla izraz koji ima sledeći prefiksni zapis:* `* + * a 3 + b 4 c`

Pitanje 7.41. *Nacrtati stablo za izraz $y = x * 5 < 3 + 4$ u skladu sa prioritetom operacija jezika C i napisati šta se dobija njegovim postfiksnim obilaskom.*

Pitanje 7.42. *Prikazati u vidu stabla izraz koji ima sledeći prefiksni zapis:* `+ * * x 2 + x 3 7`

Zadaci

Zadatak 7.4.1. *Napisati program koji za uredjeno binarno stablo ispisuje elemente na najvećoj dubini. Napisati funkcije za kreiranje čvora, unošenje elemenata u stablo, oslobadjanje stabla.*

Zadatak 7.4.2. *Celobrojni aritmetički izraz koji uključuje jednu promenljivu predstavljen je binarnim stablom. Izraz može da sadrži samo binarne operatore* `+` *i* `*`.

- *Definisati strukturu* `cvor` *kojom se mogu opisati čvorovi ovakvog stabla.*
- *Napisati funkciju*

  `int vrednost(char ime_promenljive, int vrednost_promenljive, cvor* koren)`

  koja za promenljivu `ime_promenljive` *sa vrednošću* `vrednost_promenljive` *računa vrednost izraza koji je predstavljen stablom čiji je koren* `*koren` *i vraća tu vrednost kao povratnu vrednost funkcije. Ukoliko u izrazu postoji promenljiva čija vrednost nije zadata, ispisuje se poruka o grešci i vraća vrednost 0.*

  Na primer, ako je `izraz` *pokazivač na koren stabla koje opisuje navedni izraz, onda se pozivom funkcije*

  `vrednost('x', 1, izraz)`.

  dobija vrednost 5.

  Ako je `izraz` *pokazivač na koren stabla koje opisuje navedni izraz, onda se pozivom funkcije*

`vrednost('y', 2, izraz)`.

dobija poruka:

`Promenljiva x nije definisana`

i povratna vrednost 0.

- *Napisati funkcije za ispis u prefiksnom i u infiksnom poretku drveta koje opisuje izraz.*

Podrazumevati da su svi izrazi koji se dobijaju kao argumenti ispravno formirani.

Zadatak 7.4.3. *Napisati program koji formira ureẹno binarno stablo bez ponavljanja elemenata. Elementi stabla su celi brojevi i unose se sa ulaza, a oznaka za kraj unosa je* 0*. Napisati funkciju koja proverava da li je uneto stablo uravnoteženo. Stablo je uravnoteženo ako za svaki čvor stabla važi da mu se visina levog i desnog podstabla razlikuju najviše za jedan.*

Zadatak 7.4.4. *Napisati program koji formira binarno uredjeno stablo. Napisati funkcije za:*

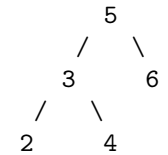
(a) ubacivanje elementa u stablo, ispisivanje stabla, oslobadjanje stabla

(b) sabiranje elemenata u listovima stabla, izračunavanje ukupnog broja čvorova stabla i izračunavanje dubine stabla.

Zadatak 7.4.5.

Napisati program koji za uredjeno binarno stablo ispisuje sve listove (list je čvor stabla koji nema potomaka). Napisati funkcije za kreiranje čvora, unošenje elementa u stablo, ispis stabla i oslobadjanje stabla.

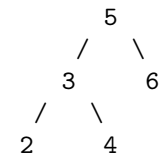
Na primer: za stablo



program treba da ispiše: 2 4 6.

Zadatak 7.4.6. *Napisati program koji za uredjeno binarno stablo računa ukupan broj listova u stablu (list je čvor stabla koji nema potomaka). Napisati funkcije za kreiranje čvora, unošenje elementa u stablo, ispis stabla i oslobadjanje stabla.*

Na primer: za stablo



program treba da ispiše: 3.

Zadatak 7.4.7. *Napisati program koji formira ureẹno binarno stablo koje sadrži cele brojeve. Brojevi se unose sa standardnog ulaza sa nulom kao oznakom za kraj unosa. Napisati funkcije za:*

- *Ubacivanje elementa u ureẹno stablo (bez ponavljanja elemenata);*
- *Ispisivanje stabla u inorder (infix) redosledu;*
- *Oslobaạnje stabla;*
- *Odreịvanje najmanje dubine lista stabla.*

Zadatak 7.4.8. *Napisati program koji formira ureẹno binarno stablo koje sadrži cele brojeve. Brojevi se unose sa standardnog ulaza sa nulom kao oznakom za kraj unosa. Napisati funkcije za:*

- *Ubacivanje elementa u ureẹno stablo (bez ponavljanja elemenata);*
- *Ispisivanje stabla u preorder (prefix) redosledu;*
- *Oslobaạnje stabla;*
- *Odreịvanje najveće dubine lista stabla.*

Zadatak 7.4.9. *Napisati program koji formira ureẹno binarno stablo bez ponavljanja elemenata. Elementi stabla su celi brojevi i unose se sa standardnog ulaza (oznaka za kraj unosa je 0). Napisati funkcije za kreiranje elementa stabla, umetanje elementa u ureẹno stablo, štampanje stabla, brisanje stabla i odreịvanje sume elemenata u listovima stabla.*

Zadatak 7.4.10. *Napisati program koji simulira rad sa stekom. Napisati funkcije* push *(za ubacivanje elementa u stek),* pop *(za izbacivanje elementa iz steka) i funkciju* peek *(koja na standardni izlaz ispisuje vrednost elementa koji se nalazi na vrhu steka - bez brisanja tog elementa iz steka).*

Zadatak 7.4.11. *Napisati program koji formira ureẹno binarno stablo bez ponavljanja elemenata čiji elementi su imena studenata i brojevi njihovih indeksa (struktura). Pretpostavka je da ime studenta nije duže od* 30 *karaktera i da je indeks dat kao ceo broj. Napisati program koji sa standardnog ulaza čita podatke o studentima, smešta ih u stablo (ureẹno prema brojevima indeksa) i štampa podatke o studentima u opadajućem poredku prema brojevima indeksa. Oznaka za kraj unosa je kada se umesto imena studenta unese niska "kraj". Napisati funkcije za kreiranje čvora stabla, umetanje studenta u stablo, brisanje stabla i ispis stabla na opisan način.*

Zadatak 7.4.12. *Napisati program koji simulira red u studentskoj poliklinici. Napisati funkcije* add *(za ubacivanje studenta na kraj reda),* get *(za izbacivanje studenta sa početka reda) i funkciju* peek *(koja na standardni izlaz ispisuje ime studenta koji se nalazi na početku reda - bez brisanja tog studenta iz reda). Podaci o studentu se sastoje od imena studenta (karakterska niska dužine ne veće od* 30*) i broja indeksa studenta (ceo broj).*

Zadatak 7.4.13. *Napisati program koji implementira red pomoću jednostruko povezane liste (čuvati pokazivače i na početak i na kraj reda). Sa standardnog ulaza se unose brojevi koje smeštamo u red sa nulom (*0*) kao oznakom za kraj unosa. Napisati funkcije za:*

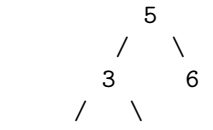
- *Kreiranje elementa reda;*
- *Ubacivanje elementa na kraj reda;*
- *Izbacivanje elementa sa početka reda;*
- *Ispisivanje reda;*
- *Oslobaạnje reda.*

Zadatak 7.4.14. *Napisati program koji formira ureẹno binarno stablo bez ponavljanja elemenata. Elementi stabla su celi brojevi i unose se sa ulaza, a oznaka za kraj unosa je nula. Napisati funkciju koja proverava da li je uneto stablo* lepo*. Stablo je* lepo *ako za svaki čvor stabla važi da mu se broj čvorova u levom i u desnom podstablu razlikuju najviše za jedan.*

Zadatak 7.4.15. *Napisati program koji formira sortiranu listu od niza celih brojeva koji se unose sa standardnog ulaza. Oznaka za kraj unosa je* 0*. Napisati funkcije za formiranje čvora liste, ubacivanje elementa u već sortiranu listu, ispisivanje elemenata liste i oslobaạnje liste.*

Zadatak 7.4.16. *Napisati funkcije potrebne za ispisivanje elemenata koji se nalaze na najvećoj dubini binarnog stabla.*

Na primer, za stablo



treba ispisati: `2 4`.

(Pretpostavljamo da je stablo već zadato. Ne treba *pisati dodatne funkcije za kreiranje čvora, unošenje elemenata u stablo i oslobaạnje stabla)*

Zadatak 7.4.17. *Napisati program koji kreira jednostruko povezanu listu. Elementi (celi brojevi) se unose sa standardnog ulaza dok se ne unese nula kao oznaka za kraj. Napisati:*

- *funkciju koja kreira jedan čvor liste,* `CVOR* napravi_cvor(int br)`
- *funkciju za ubacivanje broja na kraj liste,* `void ubaci_na_kraj(CVOR** pl, int br)`
- *funkciju koja skida element sa početka liste (pri čemu se menja početak liste) i vraća vrednost tog broja kao povratnu vrednost,*

  `int izbaci_sa_pocetka(CVOR** pl)`
- *funkciju za ispisivanje elemenata liste,* `void ispisi_listu(CVOR* l)`
- *funkciju za oslobaạnje liste,* `void oslobodi_listu(CVOR* l)`

Zadatak 7.4.18. *Napisati funkcije za rad sa stekom čiji su elementi celi brojevi. Treba napisati (samo) funkcije za dodavanje elementa u stek* `void push(cvor** s, int br)` *, brisanje elementa iz steka* `void pop(cvor** s)` *i funkciju za izračunavanje zbira svih elemenata koji su parni brojevi* `int zbir_parnih(cvor* s)`.

Zadatak 7.4.19. *Napisati funkcije za rad sa ureẹnim binarnim stablom čiji su elementi celi brojevi. Treba napisati (samo) funkcije za dodavanje elementa u stablo* `void dodaj(cvor** s, int br)` *, brisanje stabla* `void obrisi(cvor* s)` *i funkciju za izračunavanje zbira listova stabla* `int zbir_listova(cvor* s)`.

Zadatak 7.4.20. *Napisati program koji iz datoteke "reci.txt"čita redom sve reči (maksimalne dužine* 20*) i smesta ih u:*

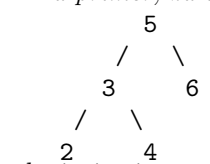
1. *Niz struktura (pretpostaviti da različitih reči u datoteci nema više od* 100*) u kome će se za svaku reč čuvati i njen broj pojavljivanja.*
2. *Red koji će za svaku reč čuvati i njen broj pojavljivanja. Napisati funkcije za ubacivanje elementa u red, štampanje reda, brisanje reda i odreịvanja najduže reči koja se pojavila u redu.*
3. *Ureẹno binarno drvo (ureẹno leksikografski) koje u svakom čvoru čuva reč i broj pojavljivanja te reči. Napisati funkcije za dodavanje elementa u stablo, ispisivanje stabla, brisanje stabla i računanje ukupnog broja pojavljivanja svih reči u stablu.*

Zadatak 7.4.21. *Napisati program koji iz datoteke čije se ime zadaje kao argument komandne linije učitava cele brojeve i smešta ih u listu. Napisati funkcije za rad sa listom, pravljenje elementa liste, ubacivanje elementa na kraj liste, ispisivanje liste i brisanje liste. Pretpostavka je da datoteka sadrži samo cele brojeve.*

Zadatak 7.4.22. *Napisati program koji sa standardnog ulaza učitava cele brojeve dok se ne unese* 0 *kao oznaka za kraj. Brojeve smestiti u ureẹno binarno stablo i ispisati dobijeno stablo. Napisati funkcije za formiranje elementa stabla, ubacivanje elementa u stablo, ispisivanje stabla u prefiksnom poretku, brisanje stabla.*

Zadatak 7.4.23. *Napisati funkcije potrebne za ispisivanje elemenata koji se nalaze na najvećoj dubini binarnog stabla.*

Na primer, za stablo



treba ispisati: `2 4`.

(Pretpostavljamo da je stablo već zadato. Ne treba *pisati dodatne funkcije za kreiranje čvora, unošenje elementa u stablo i oslobaạnje stabla)*

Zadatak 7.4.24. *Napisati program koji formira ureẹno binarno stablo bez ponavljanja elemenata čiji elementi su imena studenata (karakterska niska do* 30 *karaktera). Napisati program koji sa standardnog ulaza čita podatke o studentima, smešta ih u stablo (ureẹno leksikografski) i štampa podatke o studentima infiksno. Oznaka za kraj unosa je kada se umesto imena studenta unese niska "kraj". Napisati funkcije za kreiranje čvora stabla, umetanje studenta u stablo, brisanje stabla i ispis stabla na opisan način.*

Zadatak 7.4.25. *Napisati program koji simulira red u studentskoj poliklinici. Napisati funkcije* add *(za ubacivanje studenta na kraj reda),* get *(za izbacivanje studenta sa početka reda) i funkcije za štampanje i brisanje reda. Podaci o studentu se sastoje od imena studenta (karakterska niska dužine ne veće od* 30*) i broja indeksa studenta (ceo broj).*

Zadatak 7.4.26. *Napisati funkciju* `cvor* nova_lista(cvor* L1, cvor* L2);` *koja od dve date liste* `L1` *i* `L2`*, u kojima se čuvaju celi brojevi, formira novu listu koja koja sadrži alternirajuće raspoređene elemente iz lista* `L1` *i* `L2` *(prvi element iz* `L1`*, prvi element iz* `L2`*, drugi element* `L1`*, drugi element* `L2` *itd.) sve dok ima elemenata u* obe *liste. Ne formirati nove čvorove, već samo postojeće čvorove povezati u jednu listu, a kao rezultat vratiti početak te formirane liste.*

Zadatak 7.4.27. *Neka su čvorovi binarnog stabla koje opisuje aritmetički izraz opisani sledećom strukturom:*

```
typedef struct tagCvor
{
    int tipCvora;
    int tipOperatora;
    int vrednostKonstante;
    char oznakaPromenljive;

    struct tagCvor *levo;
    struct tagCvor *desno;
} CVOR;
```

Član `tipCvora` *moze imati sledeće vrednosti:*

0 - čvor je operator i mora imati oba deteta;

1 - čvor je konstanta i ne sme imati decu, vrednost je u članu `vrednostKonstante`*;*

2 - čvor je promenljiva sa oznakom koju definiše član `oznakaPromenljive` *i ne sme imati decu.*

Član `tipOperatora` *može imati sledeće vrednosti:*

0 - sabiranje

1 - množenje

Zadatak 7.4.28. *(a) Napisati funkciju* `void dodela(CVOR* koren, char promenljiva, int vrednost);` *koja menja izraz dat korenom u izraz u kojem je svako pojavljivanje promenljive sa datom oznakom zamenjeno konstantom sa datom vrednošću.*

(b) Napisati funkciju `int imaPromenljivih(CVOR *koren);` *koja ispituje da li izraz dat korenom sadrži promenljive. Funkcija vraća vrednost različitu od nule ako izraz ima promenljivih, odnosno nulu u suprotnom.*

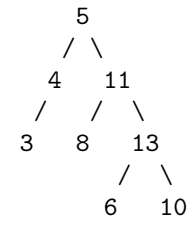
(c) Napisati funkciju `int vrednost(CVOR *koren, int* v);` *koja na adresu na koju pokazuje drugi argument funkcije smešta vrednost izraza datog korenom* `koren` *i vraća vrednost* `0`*, u slučaju da izraz nema promenljivih. U slučaju da izraz ima promenljivih funkcija treba da vraća* `-1`*.*

Podrazumevati da je drvo izraza koje se prosleđuje funkcijama ispravno konstruisano.

Zadatak 7.4.29. *Napisati funkciju* `void zamene(CVOR** p)` *koja menja povezanu listu koja je zadata svojim početkom tako da zameni mesta 1. i 2. elementu, potom 3. i 4. itd. Na primer, lista* `1->2->3->4` *postaje* `2->1->4->3`*. Lista* `1->2->3->4->5->6` *postaje* `2->1->4->3->6->5`*. Lista može sadržati i neparan broj elemenata, pri čemu u tom slučaju poslednji ostaje na svom mestu. Nije dozvoljeno formiranje nove liste - lista se može jedino preurediti u okviru postojećih struktura.*

Zadatak 7.4.30. *Dato je binarno stablo koje sadrži cele brojeve. Napisati funkciju* `CVOR* nadjiNajblizi(CVOR* koren)` *koja vraća pokazivač na čvor koji je najbliži korenu i pri tom je deljiv sa 3. Ako ima više čvorova na istom rastojanju od korena koji zadovoljavaju svojstvo, vratiti pokazivač na bilo koji.*

Na primer, u binarnom stablu



Čvorovi sa vrednostima 3 i 6 zadovoljavaju uslov, ali je 3 bliži korenu, pošto su potrebna dva poteza da bi se stiglo do njega, odnosno 3 poteza da bi se stiglo do broja 6.

Zadatak 7.4.31. *Neka je čvor binarnog stabla definisan kao*

```
typedef struct cvorStabla
{
  int broj;
  struct cvorStabla* levi;
  struct cvorStabla* desni;
} CVOR;
```

Napisati funkciju `void obrni(CVOR* koren);` *koja menja mesto levom i desnom podstablu za svaki čvor. Dozvoljeno je isključivo reorganizovanje strukture drveta pomoću postojećih pokazivača, ne i formiranje novog drveta.*

Zadatak 7.4.32. *Binarno drvo je perfektno balansirano akko za svaki čvor važi da se broj čvorova levog i desnog podrveta razlikuje najviše za jedan. Napisati funkciju koja proverava da li binarno drvo perfektno balansirano.*

Zadatak 7.4.33. *Date su dve liste. Čvor liste opisan je strukturom*

```
tupedef struct cvorliste
{
    int vr;
    struct cvorliste *sled;
}cvor;
```

Napisati funkciju `void uredjuje(cvor **lista1, cvor* lista2)` *koja izbacije iz liste* `lista1` *sve elemente koji se nalaze u listi* `lista2`*.*

Zadatak 7.4.34. *Napisati funkciju* `Cvor* dodaj_u_listu(Cvor *glava, Cvor *novi, int n)` *koja dodaje novi čvor na n-to mesto u listi. Ukoliko lista ima manje od n elemenata novi čvor se dodaje na kraj liste. Kao rezultat funkcija vraća adresu nove glave liste.*

Zadatak 7.4.35. *Definisati tip podataka koji definiše tačku u ravni. Sa standardnog ulaza se unosi broj tačaka , a zatim i njihove koordinate. Maximalni broj tačaka nije unapred poznat. Sortirati dati niz tačaka na osnovu rastojanja od koordinatnog početka.*

Zadatak 7.4.36. *Napisati funkciju* `void umetni(cvor* lista)` *koja izmeu svaka dva elementa u listi umeće element koji predstavlja razliku susedna dva.*

Zadatak 7.4.37. *Grupa od n plesača (na čijim kostimima su u smeru kazaljke na satu redom brojevi od 1 do n) izvodi svoju plesnu tačku tako što formiraju krug iz kog najpre izlazi k-ti plesač (odbrojava se pov cev od plesača označenog brojem 1 u smeru kretanja kazaljke na satu). Preostali plesači obrazuju manji krug iz kog opet izlazi k-ti plesač (odbrojava se počev os sledećeg suseda prethodno izbačenog, opet u smeru kazaljke na satu). Izlasci iz kruga se nastavljaju sve dok svi plesači ne budu isključeni. Celi brojevi* `n, k, (k<n)` *se učitavu sa standardnog ulaza. Napisati program koji će na standardni izlaz ispisati redne brojeve plesača u redosledu napuštanja kruga. PRIMER: za* `n = 5, k = 3` *redosled izlazaka je 3 1 5 2 4. NAPOMENA: Zadatak rešiti korišćenjem kružne liste.*

Zadatak 7.4.38. *Ime datoteke je argument komadne linije. U svakom redu datoteke nalazi se podatak: broj ili ime druge datoteke (zavrsava se sa .txt). Svaka datoteka je istog sadržaja. Napisati program koji posećuje sve datoteke do kojih može stići iz prve (rekurzivno), pri tome ne posećuje istu datoteku dva puta i štampa na standardni izlaz brojeve iz njih. NAPOMENA: Za pamćenje imena datoteka koristiti binarno pretraživačko drvo.*

Zadatak 7.4.39. *Dva binarna stabla su identična ako su ista po strukturi i sadržaju, odnosno oba korena imaju isti sadržaj i njihova odgovarajuća podstabla su identična. Napistati funkciju koja proverava da li su dva binarna stabla identična.*

Zadatak 7.4.40. *Neka je dat pokazivač na koren binarnog stabla čiji čvorovi sadrže cele brojeve. Napisati sledeće funkcije:*

(a) Funkciju koja vraća broj čvorova koji su po sadržaju veći od svih svojih potomaka.

(b) Funkciju koja ispisuje čvorove koji su veći od sume svih svojih potomaka.

Zadatak 7.4.41. *Sa standardnog ulaza unosi se lista celih brojeva dok se ne unese 0, a potom se unosi jedan broj. Odrediti poziciju prvog pojavljivanja broja u listi i ispisati na standardni izlaz. Ukoliko broja nema u listi ispisati -1. Pozicija u listi se broji počevši od 1.*

U datoteci `liste.h` *nalaze se funkcije za rad sa listom:*

```
void oslobodi(cvor* lista)
cvor* ubaci_na_pocetak(cvor* lista, int br)
cvor* novi_cvor(int br)
void ispis(cvor* lista)
```

Napraviti main.c u kome se testira rad programa. U main.c treba da stoji `#include "liste.h"`.
Kompaliranje: `gcc main.c liste.c`
Pokretanje: `./a.out`

NAPOMENA: Ukoliko se zadatak uradi bez korišćenja liste broj osvojenih poena je 0.

Zadatak 7.4.42. *Napisati funkciju* `int f7(cvor* lista)` *koja menja elemente liste na sledeći način: ako je tekući element paran, sledeći element uvećava za 1, a ako je element neparan sledeći element smanjuje za 1. Parnost broja se odnosi na tekuću, promenjenu vrednost broja. Funkcija vraća vrednost poslednjeg elementa liste.*

Zadatak 7.4.43. *Napisati funkcju* `int f8(cvor* drvo, int nivo)` *koja vraća broj elemenata drveta koji se nalaze na nivou* `nivo` *(*`nivo>0`*). Testirati pozivom u main-u.*

Zadatak 7.4.44. *Napisati funkciju* `void f5(cvor* l1, cvor* l2)` *koja spaja dve rastuće sortirane liste tako da rezultat i dalje bude sortiran. Rezultat sačuvati u prvoj listi. Rezulatujuću listu ispisati na standardni izlaz. Lista se učitava sve dok se ne unese 0.*

Zadatak 7.4.45. *Napisati funkciju koja iz datoteke* `karakteri.txt` *ucitava karaktere i smešta ih u binarno pretraživacko drvo. Uz svaki karakter čuvati i broj pojavljivanja karaktera u datoteci. Ispisati na standardni izlaz karakter koji se pojavljuje najveći broj puta u datoteci.*

Zadatak 7.4.46. *Stablo se učitava tako da su u listovima realni brojevi, a u unutrasnjim cvorovima neka od operacija* `+`*,* `-`*,* `*` *i* `/`*. Funkcija* `float f3(svor* drvo)` *računa izraz u stablu i vraća rezultat. Ukoliko doe̩ do deljenja nulom funkcija vraća -1.*

Zadatak 7.4.47. *U datoteci* `kupci.txt` *se nalaze podaci o kupcima oblika* `ime kolicina` *gde je* `ime` *ime kupca koji je kupio proizvod, a* `kolicina` *količina proizvoda. Kupac se može više puta pojaviti u datoteci. Napraviti binarno pretraživačko drvo prema imenima kupaca. Na standarni izlaz ispisati ime onog kupca koji je uzeo najviše proizvoda. Pretpostavlja se da je datoteka dobro strukturirana.*

Zadatak 7.4.48. *Napisati funkciju int prebroj(cvor* drvo) koja vraca broj elemenata stabla drvo koji su iste parnosti kao oba svoja sina(sva tri parna ili neparna). Ukoliko je cvor list ili ima samo jednog sina ne ulazi u zbir.*

Zadatak 7.4.49. *Napisati funkciju* `int ravnomerno_izbalansirano(Cvor *stablo)` *koja proverava da li je stablo ravnomerno izbalansirano. Stablo je ravnomerno izbalansirano ako za svaki čvor važi da je pozitivna razlika izmeu̩ dubine levog i dubine desnog podstabla najviše* `1`*. Testirati funkciju pozivom u main-u, stablo se učitava sve dok se ne unese 0. Ukoliko jeste izbalansirano ispisati* `1`*, a u suprotnom* `0`*.*

Zadatak 7.4.50. *Funkcija vrši rekurzivnu rotaciju drveta oko svih čvorova, dakle dobija se odraz prvobitnog drveta u ogledalu.*

Zadatak 7.4.51. *Napisati program koji formira ureẹno binarno stablo koje sadrži cele brojeve. Brojevi se unose sa standardnog ulaza sa nulom kao oznakom za kraj unosa. Napisati funkcije za:*

- *Kreiranje jednog čvora stabla;*
- *Ubacivanje elementa u ureẹno stablo (bez ponavljanja elemenata);*
- *Ispisivanje stabla u preorder (prefix) redosledu;*
- *Odreị vanje zbira elemenata stabla;*
- *Ispisivanje listova stabla;*
- *Oslobaạnje stabla.*

Primer, za stablo:

```
    5
   / \
  3   7
 / \
2   4
```

zbir svih elemenata je $5 + 3 + 7 + 2 + 4 = 21$ *a listovi su:* 2, 4 *i* 7.

Deo III

Osnove programskih jezika

Glava 8

Programski jezici i paradigme

8.1 Istorijat razvoja viših programskih jezika

Programiranje prvih elektronskih računara 1940-tih godina vršeno je isključivo korišćenjem mašinskog i asemblemblerskog jezika. Ipak, već u to vreme, postojala je ideja o razvoju apstraktnijih programskih jezika koji bi automatski bili prevođeni na mašinski jezik. Tako je 1948. Konrad Cuze (nem. Konrad Zuse) objavio rad o višem programskom jeziku *Plankalkül*, međutim prevodilac za ovaj jezik nikada nije implementiran.

Najznačajni jezici nastali 1950-tih godina su *Fortran*, *Lisp* i *Cobol*.

Prvi viši programski jezik koji je stekao širok krug korisnika je programski jezik *Fortran*, nastao u periodu 1953-1957 godine. Vođa tima u okviru kompanije IBM koji je projektovao ovaj jezik i implementirao prvi prevodilac za njega, bio je Džon Bekus (engl. John Backus). Ime Fortran je skraćenica za *IBM mathematical FORmula TRANslator system*. Jezik se i danas uspešno koristi (naravno, uz velike izmene u odnosu na prvobitne verzije) i namenjen je, pre svega, za numerička i naučna izračunavanja.

Programski jezik *Lisp* razvio je Džon Makarti (engl. John McCarthy) 1958. godine na univerzitetu MIT. Lisp se smatra jezikom veštačke inteligencije. Zasnovan je na Čerčovom formalizmu λ-računa i spada u grupu funkcionalnih programskih i od njega su se dalje razvili svi savremeni funkcionalni programski jezici. Ime Lisp je nastalo od *LISt Processing* jer jezik podržava liste i drveta kao osnovne strukture podataka. Neke varijante Lisp-a se i danas koriste.

Programski jezik *Cobol* napravljen je 1959. godine zajedničkom inicijativom nekoliko vodećih firmi i ljudi sa univerziteta da se napravi jezik pogodan za poslovnu upotrebu. Ime predstavlja skraćenicu od *COmmon Business Oriented Language* što ga i definiše kao jezik koji se primenjuje za izradu poslovnih, finansijskih i administrativnih aplikacija za firme i vlade. Aplikacije programirane u Cobol-u se i danas mogu sresti, pri čemu se sam Cobol danas veoma retko koristi.

Jedan od najuticajnijih programskih jezika je svakako programski jezik *Algol*.

Pascal

Basic

PL/I

Prolog Simula, *Smalltalk ML*

C

SQL

Ada C++ Perl

Haskell Python Java JavaScript Php C#

Najznačajniji događaji u razvoju programiranja i programskih jezika:

- *Plankalkul* (1948) Konrad Zuse, Nemački inženjer, definisao je, za svoj računar Z3, prvi programski jezik, ali jezik nije implementiran.
- FORTRAN: John Backus, 1957 (blizak matematičkoj notaciji, efikasno prevodenje na mašinski jezik, novi korisnici, paketi, prenosivost, čitljivost)
- LISP (1958) McCarthy, rad sa listama;

- COBOL
- Operativni sistemi
- Prolog (1972) Kovalski (neproceduralan jezik, deskriptivno programiranje);
- Algol 60 (60-tih g.)
- Algol W (1966): poboljšano struktuiranje podataka
- Pascal (1971): dalja poboljšanja
- Modula 2 (1983): koncept modula (Wirth)
- Oberon (1988): Wirth
- CPL (Strachy 1966) - Combined Programming Language: nije u potpunosti implementiran; uvodi koncepte
- BCPL (Richards 1969): Basic CPL - za pisanje kompilatora
- C 1972 - Dennis Ritchie - implementacioni jezik za softver vezan za operativni sistem UNIX (1973. prepisan u C)
- 1977 usavršen sistem tipova C-a
- C++ (Stroustrup 1986) - definisanje novih tipova; klase pozajmljene od Simule 67; strogo tipiziran sistem
- Jezici IV generacije - specijalne namene - za rad sa bazama podataka, raširenim tabelama, obradu teksta
- Internet - teleizracunavanje

8.2 Programske paradigme

U ovom poglavlju biće opisani neki način da se klasifikuju međusobno slični programski jezici. Važno je naglasiti da je veoma često da programski jezik deli osobine nekoliko različitih klasa i da granice ovakvih klasifikacija nikada ne mogu biti oštre.

U odnosu na način rešavanja problema programski jezici mogu biti:

Proceduralni - Proceduralni (ili imperativni) programski jezici zahtevaju od programera da precizno specifikuje algoritam kojim se problem rešava navodeći naredbe oblika: „Uradi ovo, uradi ono". Značajni proceduralni programski jezici su npr. C, Pascal, Java, ...

Deklarativni - Deklarativni programski jezici ne zahtevaju od programera da opisuje korake rešenja problema, već samo da opiše problem, dok se programski jezik sâm stara o pronalaženju rešenja korišćenjem nekog od ugrađenih algoritama. Ovo u mnogome olakšava proces programiranja, međutim, zbog nemogućnosti automatskog pronalaženja efikasnih algoritama koji bi rešili široku klasu problema, domen primene deklarativnih jezika je često ograničen. Najznačajniji deklarativni programski jezici su npr. Prolog i SQL. S obzirom da se programi u krajnjoj instanci moraju izvršavati na računarima fon Nojmanove arhitekture, oni se moraju prevesti na mašinski programski jezik koji je imperativan.

S obzirom na stil programiranja razlikujemo nekoliko *programskih paradigmi*. Paradigme se razlikuju po konceptima i apstrakcijama koje se koriste da bi se predstavili elementi programa (npr. promenljive, funkcije, objekti, ograničenja) i koracima od kojih se sastoje izračunavanja (dodele, sračunavanja vrednosti izraza, tokovi podataka, itd.). Najznačajnije programske paradigme su:

Imperativni jezici - Jezici imperativne paradigme posmatraju izračunavanje kao niz iskaza (naredbi) koje menjaju stanje programa određeno tekućom vrednošću promenljivih. Vrednosti promenljivih i stanje programa se menja naredbom dodele, a kontrola toka programa se vrši korišćenjem sekvencijalnog, uslovnog i cikličkog izvršavanja programa.

Najznačajniji imperativni programski jezici su Fortran, Algol i njegovi naslednici, Pascal, C, Basic, ...

Imperativni jezici, uz objektno orijentisane jezike, se najčešće koriste u industrijskom sistemskom i aplikativnom programiranju.

Imeperativni jezici su obično izrazito proceduralni.

Funkcionalni jezici - Jezici funkcionalne paradigme posmatraju izračunavanje kao proces izračunavanja matematičkih funkcija i izbegavaju koncept stanja i promenljive. Koreni funkcionalnog programiranja leže u λ-računu razvijenom 1930-tih kako bi se izučavao proces definisanja i primene matematičkih funkcija i rekurzija. Mnogi funkcionalni programski jezici se mogu smatrati nadogradnjama λ-računa.

U praksi, razlika između matematičkog pojma funkcije i pojma funkcija u imperativnim programskim jezicima je da imperativne funkcije mogu imati *bočne efekte* odnosno da uz izračunavanje rezultata menjaju tekuće stanje programa. Ovo dovodi do toga da poziv iste funkcije sa istim argumentima može da proizvede različite rezultate u zavisnosti od stanja u kojem je poziv izvršen, što pojam funkcije u imperativnim jezicima čini prilično drugačijim od matematičkih funkcija koje uvek predstavljaju isti rezultat za iste vrednosti ulaznih argumenata. Pošto funkcionalni programski jezici zabranjuju koncept promenljivih i stanja programa, u njima nisu mogući bočni efekti i one poseduju svojstvo *referencijalne transparentnosti* koje garantuje da funkcija uvek daje isti rezultat za iste ulazne argumente. Ovim je mnogo jednostavnije razumeti i predvideti ponašanje programa i eliminišu se mnoge greške koje nastaju tokom razvoja imperativnih programa, što je jedna od ključnih motivacija za razvoj funkcionalnih programskih jezika. Takođe, referencijalna transparentnost omogućava mnoge automatske optimizacije izračunavanja što današnje funkcionalne jezike čini veoma efikasnim. Jedna od značajnih optimizacija je mogućnost automatske paralelizacije i izvršavanja na višeprocesorskim sistemima.

Važnost funkcionalnih programskih jezika je obično bolje shvaćena u akademskom nego u industrijskom okruženju. Korektnost funkcionalnih programa se dokazuje mnogo lakše nego u imperativnom slučaju, tako da se funkcionalno programiranje koristi za izgradnju bezbednosno kritičnih sistema (npr. softver u nuklearnim elektranama, svemirskim programima, avionima...). Najpoznatiji funkcionalni programski jezici su Lisp, Scheme, ML, Haskell, ...

Logički jezici - Logički programski jezici (po najširoj definiciji ove paradigme) su oni jezici koji koriste matematičku logiku za programiranje računara. Na osnovu ove široke definicije koreni logičkog programiranja leže još u radovima Makartija u oblasti veštačke inteligencije 1950-tih godina. Ipak, najznačajniji logički jezik je Prolog razvijen 1970-tih godina. Logički jezici su obično deklarativni.

Objektno-orijentisani jezici - Objektno-orijentisani jezici koriste *objekte* - specijalizovane strukture podataka koje uz polja podataka sadrže i metode kojima se manipuliše podacima. Podaci se mogu obrađivati isključivo primenom metoda što smanjuje zavisnosti između različitih komponenata programskog koda i čini ovu paradigmu pogodnu za razvoj velikih aplikacija uz mogućnost saradnje većeg broja programera. Najčešće korišćene tehnike programiranja u OOP uključuju sakrivanje informacija, enkapsulaciju, apstraktne tipove podataka, modularnost, nasleđivanje i polimorfizam.

Najznačajniji OO jezici su C++, Java, Eifell, Simula, SmallTalk, ...

Glava 9

Uvod u prevođenje programskih jezika

9.1 Implementacija programskih jezika

- Various strategies depend on how much preprocessing is done before a program can be run, and how CPU-specific the program is.
- Interpreters run a program "as is"with little or no pre-processing, but no changes need to be made to run on a different platform.
- Compilers take time to do extensive preprocessing, but will run a program generally 2- to 20- times faster.
- Some newer languages use a combination of compiler and interpreter to get many of the benefits of each.
- Examples are Java and Microsoft's .NET, which compile into a virtual assembly language (while being optimized), which can then be interpreted on any computer.
- Other languages (such as Basic or Lisp) have both compilers and interpreters written for them.
- Recently, "Just-in-Timečompilers are becoming more common - compile code only when its used!

9.2 Kratka istorija razvoja kompilatora

- 1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a đefense calculator"in the Korean War. No high-level languages were available, so all programming was done in assembly
- As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.
- In 1953, John Backus came up with the idea of špeed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly. He was sure he could do better.
- In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."
- The initial release of FORTRAN I was in 1956, totaling 25,000 lines of assembly code. Compiled programs ran almost as fast as handwritten assembly! Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.

9.3 Moderni kompilatori

- Compilers have not changed a great deal since the days of Backus. They still consist of two main components:
- The front-end reads in the program in the source languages, makes sense of it, and stores it in an internal representation...
- ...and the back-end, which converts the internal representation into the target language, perhaps with optimizations. The target language used is typically an assembly language, but it is often easier to use a more established, higher-level language.

9.4 Struktura kompilatora

Source Language

⇓

Front End

⇓

Intermediate Code

⇓

Back End

⇓

Target Language

Front End:

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermetiate Code Generator

Front End:

- Code Optimizer
- Target Code Generator

Overall structure:

Source Language

⇓

Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermetiate Code Generator

⇓

Intermediate Code

⇓

Code Optimizer

Target Code Generator

⇓

Target Language

9.5 Leksička analiza

Lexical analysis is the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called "lexical tokens", or just "tokens". For instance, lexical analyzers (*lexers*) for many programming languages will convert the character sequence `123abc` into the two tokens `123` and `abc`. The purpose of producing these tokens is usually to forward them as input to another program, such as a parser.

Consider the code:

```
if (i==j);
    z=1;
else;
    z=0;
endif;
```

This is really nothing more than a string of characters:

`if_(i==j);\n\tz=1;\nelse;\n\tz=0;\nendif;`

During lexical analysis phase we must divide this string into meaningful sub-strings.

The output of our lexical analysis phase is a streams of *tokens*. A token is a syntactic category. In English this would be types of words or punctuation, such as a "noun", "verb", "adjective"or "end-mark". In a program, this could be an "identifier", a "floating-point number", a "math symbol", a "keyword", etc.

A sub-string that represents an instance of a token is called a *lexeme*.

For the token `IDENTIFIER`, possible lexemes are `a`, `b`,...

The class of all possible lexemes in a token is described by the use of a pattern. The pattern to describe the token `IDENTIFIER` is a string of letters, numbers, or underscores, beginning with a non-number. Patterns are typically described using regular expressions.

For lexical analysis:

- Regular expressions describe tokens
- Finite automata are mechanisms to generate tokens from input stream.

`lex` is a program that generates lexers in the C programming language.

9.6 Sintaksna analiza

Syntax analysis is a process in compilers that recognizes the structure of programming languages. It is also known as parsing. After lexical analysis, it is much easier to write and use a parser, as the language is far simpler.

Context-free grammar is usually used for describing the structure of languages and BNF notation is typical to define that grammar. Grammatical tokens include numerical constants and literal strings and control structures such as assignments, conditions and loops.

Programs or code that do parsing are called *parsers*.

Goal: we must determine if the input token stream satisfies the syntax of the program

What do we need to do this?

- An expressive way to describe the syntax
- A mechanism that determines if the input token stream satisfies the syntax description

Regular expressions don't have enough power to express any non-trivial syntax of a programming language. Syntax of programming languages is often described by context-free grammars.

Parse Tree:

- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges:
  - From Nonterminal of LHS of production
  - To Nodes from RHS of production
- Captures derivation of string

```
    Expr
Expr Op Expr Int     Int 2    -   1
```

`Yacc` (yet another compiler compiler) is a program that generates parsers in the C programming language.

9.7 Primer

- Source Code:

  ```
  cur_time = start_time + cycles * 60
  ```

- Lexical Analysis:

  ```
  ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)
  ```

- Syntax Analysis:

  ```
      ASSIGN
  ID(1)      ADD
        ID(2)   MULT
             ID(3) INT(60)
  ```

- Sematic Analysis:

  ```
      ASSIGN
  ID(1)      ADD
        ID(2)   MULT
             ID(3) int2real
                   INT(60)
  ```

- Intermediate Code:

  ```
  temp1 = int2real(60) temp2 = id3 * temp1 temp3 = id2 + temp2 id1 =
  temp3
  ```

- Optimized Code :

  Step 1:

  ```
  temp1 = 60.0 temp2 = id3 * temp1 temp3 = id2 + temp2 id1 = temp3
  ```

  Step 2:

```
temp2 = id3 * 60.0 temp3 = id2 + temp2 id1 = temp3
```

Step 3:

```
temp2 = id3 * 60.0 id1 = id2 + temp2
```

Optimized Code:

```
temp1 = id3 * 60.0 id1 = id2 + temp1
```

- Target Code Generator
  Target Code:

```
MOVF id3, R2 MULF #60.0, R2 MOVF id2, R1 ADDF R2, R1 MOVF R1, id1
```

Kako bi bilo moguće prevođenje programa u odgovarajuće programe na mašinskom jeziku nekog konkretnog računara, neophodno je precizno definisati sintaksu i semantiku programskih jezika. Podsetimo se, pitanjima ispravnosti programa bavi se *sintaksa programskih jezika* (i njena podoblast *leksika programskih jezika*). Leksika se bavi opisivanjem osnovnim gradivnim elementima jezika, a sintaksa načinima za kombinovanje tih osnovnih elemenata u ispravne jezičke konstrukcije. Pitanjem značenja programa bavi *semantika programskih jezika.*

9.8 Teorija formalnih jezika

Naglašeno je da izgradnja jezičkih procesora zahteva matematički precizne definicije ispravnih konstrukcija programskih jezika. Oblast matematike koja se bavi formalnim definisanjem pojma jezika i formalnim opisima leksike i sintakse programskih jezika naziva se *formalna teorija jezika (engl. formal language theory).* U nastavku će biti ukratko uvedeni osnovni pojmovi ove grane računarstva i biće date smernice na njihovu primenu u okviru opisa programskih jezika.

Slovo, azbuka, reč, jezik

Azbuke. Za zapis kako prirodnih tako i veštačkih jezika koriste se unapred fiksirani skupovi simbola (slova). Skup simbola koji se koriste za zapis nekog jezika naziva se *azbuka* tog jezika.

Definicija 9.1. Azbuka (alfabet) *je konačan, neprazan skup simbola (slova) za koje pretpostavljamo da nijedan od njih ne sadrži neki drugi. Azbuka se obično obeležava sa* Σ.

Reči. Reči nad nekom azbukom su konačni nizovi simbola te azbuke. Specijalno, *prazna reč* je je reč koja ne sadrži nijedan simbol i označavamo je sa ε. Definišimo sada i formalno skup Σ^* svih reči nad azbukom Σ.

Definicija 9.2. Σ^* *je najmanji skup za koji važi:*

1. $\varepsilon \in \Sigma^*$,
2. *za svako slovo* $a \in \Sigma$ *i svaku reč* $x \in \Sigma^*$ *važi da je* $ax \in \Sigma^*$.

Elementi skupa Σ^* *nazivaju se* reči nad azbukom Σ.
Skup $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ *je skup svih* nepraznih reči nad azbukom Σ.

Dakle, svaka reč je ili prazna reč ε ili je oblika ax za neko slovo a i reč x.

Primetimo da je skup Σ^* beskonačan, ali prebrojiv (ima istu kardinalnost kao skup prirodnih brojeva, tj. svi njegovi elementi mogu se poredjati u niz).

Osnovna operacija koja se može izvoditi nad rečima je operacija *dopisivanja (konkatenacije).* Ova operacija se obično označava simbolom $\cdot$ ili se, još češće, ovaj simbol izostavlja.

Operaciju dopisivanja, moguće je uvesti narednom (rekurzivnom) definicijom:

Definicija 9.3. *Neka su x i y reči iz Σ^*.* Konkatenacija *reči x i y je reč $x \cdot y$ iz Σ^* takva da važi:*

1. *ako je $x = \varepsilon$, tada je $x \cdot y = y$,*
2. *ako je $x = ax'$, tada je $x \cdot y = a(x' \cdot y)$.*

Skup svih reči Σ^* u odnosu na operaciju dopisivanja čini tzv. *slobodni monoid*. U ovom monoidu važi i zakon skraćivanja (kancelacije).

Teorema 9.1. *Svojstva konkatenacije (dopisivanja):*

1. *Konkatenacija je asocijativna:*
$$(xy)z = x(yz)$$
2. *Prazna reč ε je jedinični element za konkatenaciju:*
$$\varepsilon x = x\varepsilon = x$$
3. *Za konkatenaciju je dozvoljeno "skraćivanje":*
$$xz = yz \;\Rightarrow\; x = y$$
$$zx = zy \;\Rightarrow\; x = y$$

Jezici. Jezici su proizvoljni skupovi reči.

Definicija 9.4. *Neka je Σ neka azbuka. Svaki podskup skupa svih reči Σ^* nad Σ zovemo* jezik *nad Σ.*

Nad jezicima moguće je primenjivati određene operacije.

Skupovne operacije - pošto su jezici skupovi reči, nad njima je moguće izvoditi uobičajene skupovne operacije: unija ($\cup$), presek ($\cap$), razlika ($\setminus$), komplement u odnosu na Σ^* ($'$), ...

Proizvod jezika - Operacija dopisivanja reči se prirodno uzdiže na nivo jezika.

Definicija 9.5. *Neka su L_1 i L_2 jezici nad azbukom Σ.* Proizvod jezika *$L_1 \cdot L_2$ je jezik nad azbukom Σ definisan na sledeći način:*

$$L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$$

Proizvod jezika podrazumeva nadovezivanje proizvoljne reči prvog jezika sa proizvoljnom reči drugog jezika.

Primer 9.1. *Neka je*
$$L_1 = \{\text{BG},\ \text{NS},\ \text{NI}\},$$
$$L_2 = \{000,\ 001,\ \ldots,\ 998,\ 999\}.$$
Tada je
$$L_1 \cdot L_2 = \{\text{BG000},\ \ldots,\ \text{BG999},\ \text{NS000},\ \ldots,\ \text{NS999},\ \text{NI000},\ \ldots, \text{NI999}\}.$$

S obzirom da je konkatenacija asocijativna, korišćenjem proizvoda, na uobičajeni način se može uvesti i stepen jezika i to narednom induktivnom definicijom.

Definicija 9.6. *Neka je L jezik nad azbukom Σ.* Stepeni jezika *L, definisani su sa:*
$$L^0 = \{\varepsilon\},$$
$$L^{i+1} = L \cdot L^i, \quad i \geq 0$$

Primetimo da n-ti stepen jezika podrazumeva nadovezivanje n proizvoljnih reči jezika L.

Klinijevo zatvorenje - Veoma često se razmatraju jezici koji se sastoje od reči nastalih dopisivanjem proizvoljnog broja reči nekog jezika L. Pošto i-ti stepen jezika L predstavlja nadovezivanje proizvoljnih i reči jezika L, od interesa je uniju i-tih stepena za sve brojeve i, tj. razmatrati *Klinijevo zatvorenje (iteraciju)* jezika L

$$L^* = \bigcup_{i \geq 0} L^i$$

Ipak, Klinijevo zatvorenje uvodimo narednom definicijom koja ne koristi pojmove proizvoda, stepena i unije.

Definicija 9.7. *Neka je L jezik nad azbukom Σ. Klinijevo zatvorenje L^* jezika L je najmanji jezik nad azbukom Σ koji sadrži ε i sve reči jezika L i zatvoren je u odnosu na konkatenaciju reči.*

Primer 9.2. *Neka je jezik $L = \{ab, ba\}$. Tada je*

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, ababab, \ldots\}$$

Regularni izrazi

U slučaju da je jezik konačan, on se računaru može specifikovati nabrajanjem njegovih elemenata. Sa druge strane, beskonačni jezici zahtevaju druge načine specifikacije koje će biti računarski čitljive. Neki složeni jezici se mogu izgraditi od jednostavnijih, konačnih jezika, ili čak jednočlanih jezika korišćenjem navedenih operacija nad jezicima.

Definicija 9.8. Klasa regularnih jezika Reg_Σ *nad azbukom Σ je najmanji skup koji zadovoljava:*

1. $\{\varepsilon\} \in Reg_\Sigma$,
2. *za svako slovo $a \in \Sigma$, važi $\{a\} \in Reg_\Sigma$,*
3. *za svako L_1 i L_2 iz Reg_Σ, važi $L_1 \cup L_2 \in Reg_\Sigma$,*
4. *za svako L_1 i L_2 iz Reg_Σ, važi $L_1 \cdot L_2 \in Reg_\Sigma$,*
5. *za svako L iz Reg_Σ, važi $L^* \in Reg_\Sigma$.*

Elemente skupa Reg_Σ nazivamo regularnim jezicima.

Iako je definicija klase regularnih jezika zahtevala samo zatvorenost u odnosu na uniju, moguće je pokazati da je ova klasa zatvorena i u odnosu na ostale skupovne operacije nad jezicima.

Primer 9.3. *Identifikatori u programskom jezikom C mogu da se sastoje od slova (bilo malih, bilo velikih), cifara i podvlaka _, pri čemu ne smeju da počnu cifrom. Neka je $L_S = \{a, \ldots, z, A, \ldots, Z\}$, $L_C = \{0, \ldots, 9\}$ i $L_\_ = \{\_\}$. Tada za jezik svih identifikatora L_i važi:*

$$L_I = (L_s \cup L_\_) \cdot (L_S \cup L_C \cup L_\_)^*.$$

Pošto su L_S i L_C konačni jezici, oni se mogu dobiti operacijom unije krenuvši od jednočlanih skupova koji sadrže slova i cifre. Dakle, jezik L_i je regularan jezik.

Kako bi se regularni jezici mogli specifikovati računaru, koriste se regularni izrazi koji su već ranije bili opisani. Iako se regularnim izrazima predstavljaju regularni jezici, obično se u zapis regularnih izraza uvode određene skraćenice kojim se olakšava rad, pri čemu se suštinski ne proširuje klasa dopuštenih jezika. Ovakvi regularni izrazi se nazivaju i *prošireni regularni izrazi.*

Formalne gramatike

Definicija 9.9. *Neka su Σ i N dve disjunktne azbuke i neka je S slovo azbuke N ($S \in N$). Formalna gramatika (gramatika Čomskog) je uredjena četvorka*

$$G = (N, \Sigma, P, S),$$

gde je P (konačan) skup pravila izvodjenja *(*pravila zamene *ili* produkcionih pravila*) oblika:*

$$\alpha \to_G \beta \ ,$$

pri čemu je $\alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^*$.

Skup Σ zovemo skupom završnih (terminalnih) simbola*, skup N zovemo* skupom nezavršnih (neterminalnih) simbola*, a slovo S zovemo* početnim (ili polaznim) simbolom *ili* aksiomom gramatike*.*

Pravila izvodjenja $\alpha \to_G \beta_1$, $\alpha \to_G \beta_2$, ..., $\alpha \to_G \beta_n$, kraće zapisujemo na sledeći način:

$$\alpha \ \to_G \ \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_n \ .$$

Definicija 9.10. *Neka su reči x i y reči nad azbukom $N \cup \Sigma$. Kažemo da da je reč y neposredna posledica reči x i da je reč y neposredno izvodiva iz reči x, u gramatici $G = (N, \Sigma, S, P)$ i pišemo*

$$x \Rightarrow_G y,$$

ako i samo ako postoje reči $\phi, \psi, \alpha, \beta$ takve da je $x = \phi\alpha\psi$, $y = \phi\beta\psi$ i $\alpha \to_G \beta \in P$.

Definicija 9.11. *Relacija $\Rightarrow_G^k$ je k-ti stepen relacije $\Rightarrow_G$. Dakle, ako važi $x \Rightarrow_G x_1$, $x_1 \Rightarrow_G x_2$, ..., $x_{k-1} \Rightarrow_G y$, onda pišemo kraće $x \Rightarrow_G^k y$, i kažemo* reč y se izvodi u k koraka iz reči x*. Ovaj lanac izvođenja nekada kraće zapisujemo sa $x \Rightarrow_G x_1 \Rightarrow_G x_2 \Rightarrow_G \ldots \Rightarrow_G x_{k-1} \Rightarrow_G y$.*

Definicija 9.12. *Tranzitivno zatvorenje relacije $\Rightarrow_G$ označavamo sa $\Rightarrow_G^+$, a tranzitivno i refleksivno sa $\Rightarrow_G^*$. Vezu $x \Rightarrow_G^* y$ ili $x \Rightarrow_G^+ y$ čitamo:* iz y se posredno izvodi x *ili* reč y je posredna posledica reči x *ili* iz x posredno sledi y.

Definicija 9.13. *Jezik gramatike $G = (N, \Sigma, P, S)$, u oznaci $L(G)$ definišemo na sledeći način:*

$$L(G) = \{x \mid x \in \Sigma^* \ \wedge \ S \Rightarrow_G^+ x\} \ .$$

Primetimo da je formalna gramatika konačan opis potencijalno beskonačnog jezika. Ako je jezik generisan formalnom gramatikom beskonačan, onda je on prebrojiv.

Primer 9.4. *Neka je $\Sigma = \{a, b\}$, $N = \{S\}$ i $P = \{S \to \varepsilon \mid aSb\}$. Tada važi*

$$S \Rightarrow_G aSB \Rightarrow_G aaSbb \Rightarrow_G aaaSbbb \Rightarrow_G aaabbb.$$

Dakle, važi da $S \Rightarrow_G^+ aaabbb$, pa pošto je $aaabbb \in \Sigma^*$, reč $aaabbb$ pripada jeziku gramatike $G = (\Sigma, N, S, P)$. Slično, svaka reč oblika $a^n b^n$ pripada jeziku $L(G)$. Važi i obratno, da je svaka reč jezika $L(G)$ oblika $a^n b^n$ tako da je jezik gramatike G jezik $\{a^n b^n, n \geq 0\}$. Podsetimo se da smo za ovaj jezik napomenuli da nije regularan.

Klasifikacija Čomskog.

Definicija 9.14. *Gramatika $G = (N, \Sigma, P, S)$ je*

1° Desno linearna (DL) *ako su sva njena pravila oblika*

$$A \to w \quad ili \quad A \to wB \quad ,$$

gde je $A, B \in N$ i $w \in \Sigma^*$.

2° Levo linearna (LL) *ako su sva njena pravila oblika*

$$A \to w \quad ili \quad A \to Bw \quad ,$$

gde je $A, B \in N$ i $w \in \Sigma^*$.

3° Kontekst slobodna (KS) *ako su sva njena pravila oblika*

$$A \to \alpha \quad ,$$

gde je $A \in N$ *i* $\alpha \in (N \cup \Sigma)^*$.

4° Kontekst zavisna (KZ) *ako su sva njena pravila oblika*

$$\alpha \to \beta \quad ,$$

gde je $\alpha, \beta \in (N \cup \Sigma)^*$ *i* $|\alpha| \leq |\beta|$.

Primetimo da su sve desno (i levo) linearne gramatike kontekst slobodne. Kontekst slobodna gramatika je kontekst zavisna ukoliko ne sadrži nijedno $e-$pravilo (za pravilo $A \to e$ ne važi $1 = |A| \leq |e| = 0$).

Za jezik L se kaže da je desno linearan, odnosno kontekst slobodan, odnosno kontekst zavisan ako postoji gramatika odgovarajućeg tipa koja ga generiše, tj. $L = L(G)$.

Klase levo linearnih, desno linearnih i regularnih jezika se poklapaju.

9.9 Načini opisa leksike i sintakse programskih jezika

Kako bi se izvršila standardizacija i olakšala izgradnja jezičkih procesora potrebno je precizno definisati šta su ispravne leksičke, odnosno sintaksne konstrukcije programskog jezika. Opisi na govornom jeziku, iako mogući, obično nisu dovoljno precizni i potrebo je korišćenje preciznijih formalizama. Ovi formalizmi se nazivaju *metajezici* dok se jezik čija se sintaksa opisuje korišćenjem metajezika naziva *objektni jezik*. Meta jezici obično rezervišu neke simbole kao specijalne i imaju svoju posebnu sintaksu. Meta jezici se obično konstruišu tako da se za njihov zapis koriste ASCII karakteri i koji se može lako zadati računaru.

U praksi se kao metajezici obično koriste *regularni izrazi*, *BNF (Bekus-Naurova forma)*, *EBNF (proširena Bekus-Naurova forma)* i *sintaksni dijagrami*. BNF je pogodna notacija za zapis kontekstno slobodnih gramatika, EBNF proširuje BNF operacijama regularnih izraza čime se dobija pogodniji zapis, dok sintaksni dijagrami predstavljaju slikovni metajezik za predstavljanje sintakse. Dok se BNF može veoma jednostavno objasniti, precizna definicija EBNF zahteva malo više rada i ona je data kroz ISO 14977 standard.

Regularni izrazi Za opis leksičkih konstrukcija pogodno je koristiti formalizam *regularnih izraza (engl. regular expressions)*.

Osnovne konstrukcije koje se koriste prilikom zapisa regularnih izraza su

karakterske klase - navode se između `[` i `]` i označavaju jedan od navedenih karaktera. Npr. klasa `[0-9]` označava cifru.

alternacija - navodi se sa `|` i označava alternativne mogućnosti. Npr `a|b` označava slovo `a` ili slovo `b`.

opciono pojavljivanje - navodi se sa `?`. Npr. `a?` označava da slovo `a` može, a ne mora da se javi.

ponavljanje - navodi se sa `*` i označava da se nešto javlja nula ili više puta. Npr. `a*` označava niz od nula ili više slova `a`.

pozitivno ponavljanje - navodi se sa `+` i označava da se nešto javlja jedan ili više puta. Npr. `[0-9]+` označava neprazan niz cifara.

Primer 9.5. *Razmotrimo identifikatore u programskom jeziku C. Govornim jezikom, identifikatore je moguće opisati kao „neprazne niske koje se sastoje od slova, cifara i podvlaka, pri čemu ne mogu da počnu cifrom". Ovo znači da je prvi karakter identifikatora ili slovo ili podvlaka, za čim sledi nula ili više slova, cifara ili podvlaka. Na osnovu ovoga, moguće je napisati regularni izraz kojim se opisuju identifikatori:*

```
([a-zA-Z]|_)([a-zA-Z]|_|[0-9])*
```

Ovaj izraz je moguće zapisati još jednostavnije kao:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

Kontekstno slobodne gramatike Formalizam regularnih izraza je obično dovoljan da se opišu leksički elementi programskog jezika (npr. skup svih identifikatora, skup brojevnih literala, i slično). Međutim nije moguće konstruisati regularne izraze kojim bi se opisale neke konstrukcije koje se javljaju u programskim jezicima. Tako, na primer, nije moguće regularnim izrazom opisati skup reči $\{a^n b^n, n > 0\} = \{ab,\ aabb,\ aaabbb,\ \ldots\}$. Takođe, nije moguće napisati regularni izraz kojim bi se opisali svi ispravni aritmetičkih izrazi, tj. skup $\{a,\ a+a,\ a*a,\ a+a*a,\ a*(a+a),\ \ldots\}$.

Sintaksa jezika se obično opisuje gramatikama. U slučaju prirodnih jezika, gramatički opisi se obično zadaju neformalno, koristeći kao govorni jezik kao meta jezik u okviru kojega se opisuju ispravne konstrukcije, dok se u slučaju programskih jezika, koriste znatno precizniji i formalniji opisi. Za opis sintaksnih konstrukcija programskih jezika koriste se uglavnom *kontekstno slobodne gramatike (engl. context free grammars)*.

Kontekstno slobodne gramatike su izražajniji formalizam od regularnih izraza. Sve što je moguće opisati regularnim izrazima, moguće je opisati i kontekstno slobodnim gramatikama, tako da je kontekstno slobodne gramatike moguće koristiti i za opis leksičkih konstrukcija jezika (doduše regularni izrazi obično daju koncizniji opis).

Kontekstno slobodne gramatike su određene skupom pravila. Sa leve strane pravila nalaze se tzv. pomoćni simboli (neterminali), dok se sa desne strane nalaze niske u kojima mogu da se javljaju bilo pomoćni simboli bilo tzv. završni simboli (terminali). Svakom pomoćnom simbolu pridružena je neka sintaksna kategorija. Jedan od pomoćnih simbola se smatra istaknutim, naziva se početnim simbolom (ili aksiomom). Niska je opisana gramatikom ako je moguće dobiti krenuvši od početnog simbola, zamenjujući u svakom koraku pomoćne simbole desnim stranama pravila.

Primer 9.6. *Pokazano je da je jezik identifikatora programskog jezika C regularan i da ga je moguće opisati regularnim izrazom. Sa druge strane, isti ovaj jezik je moguće opisati i formalnom gramatikom. Razmotrimo naredna pravila (simbol ε označava praznu reč):*

$$\begin{array}{rcl} I & \to & XZ \\ X & \to & S \mid P \\ Z & \to & YZ \mid \varepsilon \\ Y & \to & S \mid P \mid C \\ S & \to & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z} \\ P & \to & \texttt{\_} \\ C & \to & \texttt{0} \mid \ldots \mid \texttt{9} \end{array}$$

Identifikator `x_1` *je moguće izvesti kao*

$$I \Rightarrow XZ \Rightarrow SZ \Rightarrow \texttt{x}Z \Rightarrow \texttt{x}YZ \Rightarrow \texttt{x}PZ \Rightarrow \texttt{x\_}Z \Rightarrow \texttt{x\_}YZ \Rightarrow \texttt{x\_}CZ \Rightarrow \texttt{x\_1}Z \Rightarrow \texttt{x\_1}.$$

Neterminal S odgovara slovima, neterminal P podvlaci, neterminal C ciframa, neterminal X slovu ili podvlaci, neterminal Y slovu, podvlaci ili cifri, a neterminal Z nizu simbola koji se mogu izvesti iz Y tj. nizu slova, podvlaka ili cifara.

Primer 9.7. *Neka je gramatika određena skupom pravila:*

$$\begin{array}{rcl} E & \to & E + E \\ & \mid & E * E \\ & \mid & (E) \\ & \mid & a. \end{array}$$

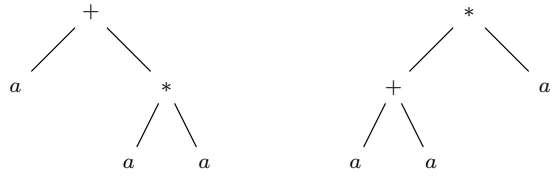
Ova gramatika opisuje ispravne aritmetičke izraze u kojima su dopuštene operacije sabiranja i množenja. Npr. izraz $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

Međutim, isti izraz je moguće izvesti i kao

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a.$$

Prvo izvođenje odgovara levo, a drugo desno prikazanom sintaksnom stablu:



Primer 9.8. *Neka je gramatika zadata sledećim pravilima:*

$$\begin{array}{rcl} E & \rightarrow & E + T \\ & | & T \\ T & \rightarrow & T * F \\ & | & F \\ F & \rightarrow & (E) \\ & | & a \end{array}$$

I ova gramatika opisuje ispravne aritmetičke izraze. Na primer, niska $a + a * a$ se može izvesti na sledeći način:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a.$$

Neterminal E odgovara izrazima, neterminal T sabircima (termima), a neterminal F činiocima (faktorima).

Primetimo da je ova gramatika je u određenom smislu preciznija od gramatike date u prethodnom primeru, s obzirom da je njome jednoznačno određen prioritet i asocijativnost operatora, što sa prethodnom gramatikom nije bio slučaj.

Konteksne gramatike čine samo jednu specijalnu vrstu formalnih gramatika. U konteksno slobodnim gramatikama sa leve strane pravila nalaze se uvek tačno jedan neterminalni simbol, a u opštem slučaju, sa leve strane pravila može se nalaziti proizvoljan niz terminalnih i neterminalnih simbola.

BNF. Metajezik pogodan za zapis pravila kontekstno slobodnih gramatika je *BNF*. Prvu verziju jezika kreirao je Džon Bakus, a ubrzo zatim poboljšao je Piter Naur i ta poboljšana verzija je po prvi put upotrebljena da se formalno definiše sintaksa programskog jezika Algol 60. BNF je u početku označavala skraćenicu od „Bakusova normalna forma" (engl. Backus Normal Form), međutim na predlog Donalda Knuta, a kako bi se naglasio doprinos Naura, ime je izmenjeno u Bakus-Naurova forma (engl. Backus Naur Form) (čime je i jasno naglašeno da BNF nije *normalna forma* u smislu normalnih formi gramatika Čomskog).

U BNF notaciji, sintaksa objektnog jezika se opisuje pomoću konačnog skupa *metalingvističkih formula (MLF)* koje direktno odgovaraju pravilima kontekstno slobodnih gramatika.

Svaka metalingvistička formula se sastoji iz leve i desne strane razdvojene specijalnim, tzv. „univerzalnim" metasimbolom (simbolom metajezika koji se koristi u svim MLF) `::=` koji se čita „po definiciji je", tj. MLF je oblika `A ::= a`, gde je `A` metalingvistička promenljiva, a `a` metalingvistički izraz. Metalingvističke promenljive su fraze prirodnog jezika u uglastim zagradama (<, >), i one predstavljaju pojmove, tj. sintaksne kategorije objekt-jezika. Ove promenljive odgovaraju pomoćnim (neterminalnim) simbolima formalnih gramatika. Na primer, u programskom jeziku, sintaksne kategorije su `<program>`, `<ceo broj>`, `<cifra>`, `<znak broja>`, `<identifikator>`, itd. U prirodnom jeziku, sintaskne kategorije su `<rec>`, `<recenica>`, `<interpunkcijski znak>`, itd. Metalingvističke promenljive ne pripadaju objekt jeziku. U nekim knjigama se umesto uglastih zagrada < i > metalingvističke promenljive označavaju korišćenjem masnih slova.

Metalingvističke konstante su simboli objekt jezika. To su, na primer, `0`, `1`, `2`, `+`, `-`, ali i rezervisane reči programskog jezika, npr. `if`, `then`, `begin`, `for`, itd.

Dakle, uglaste zagrade razlikuju neterminalne simbole tj. imena sintaksnih kategorija od terminalnih simbola objektnog jezika koji se navode tačno onako kakvi su u objekt jeziku.

Metalingvistički izrazi se grade od metalingvističkih promenljivih i metalingvističkih konstanti primenom operacija konkatenacije i alternacije (`|`).

Metalingvistička formula `A ::= a` ima značenje: ML promenljiva `A` po definiciji je ML izraz `a`. Svaka metalingvistička promenljiva koja se pojavljuje u metalingvističkom izrazu `a` mora se definisati posebnom MLF.

Primer 9.9. *Jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

```
<ceo broj> ::= <neoznacen ceo broj>
            |  <znak broja> <neoznacen ceo broj>
<neoznacen ceo broj> ::= <cifra>
                       |  <neoznacen ceo broj><cifra>
<cifra> ::= 0|1|2|3|4|5|6|7|8|9 <znak broja> ::= +|-
```

Primer 9.10. *Gramatika aritmetičkih izraza može u BNF da se zapiše kao:*

```
<izraz> ::= <izraz> + <term> | <term> <term> ::= <term> * <faktor> |
<faktor> <faktor> := ( <izraz> ) | <ceo broj>
```

EBNF, ISO 149777 *EBNF*, je skraćenica od „Extend Backus-Naur Form" tj. „Proširena Bekus-Naurova forma". Ovaj formalizam ne omogućava da se definiše bilo šta što nije moguće definisati korišćenjem BNF, međutim uvodi skraćene zapise koji olakšavaju zapis gramatike i čine je čitljivijom. Nakon korišćenja BNF za definisanje sintakse Algola 60, pojavile su se mnoge njene modifikacije i proširenja.

EBNF proširuje BNF nekim elementima regularnih izraza:

- Vitičaste zagrade `{..}` okružuju elemente izraza koji se mogu ponavljati proizvoljan broj (nula ili više) puta. Alternativno, moguće je korišćenje i sufiksa `*`.
- Pravougaone zagrade `[..]` okružuju opcione elemente u izrazima, tj. elemente koji se mogu pojaviti nula ili jedan put. Alternativno, moguće je korišć enje i sufiksa `?`.
- Sufiks `+` označava elemente izraza koji se mogu pojaviti jednom ili više puta,

Svi ovi konstrukti se mogu izraziti i u BNF metajeziku, ali uz korišćenje znatno većeg broja MLF, što narušava čitljivost i jasnost. Izražajnost i BNF metajezika i EBNF metajezika jednaka je izražajnosti kontekst slobodnih gramatika, tj. sva tri ova formalizma mogu da opišu isti skup jezika.

Ponekad se usvaja konvencija da se terminali od jednog karaktera okružuju navodnicima `"` kako bi se razlikovali od meta-simbola EBNF.

Primer 9.11. *Korišćenjem EBNF identifikatori programskog jezika C se mogu definisati sa:*

```
<identifikator> ::= <slovo ili _> { <slovo ili _> | <cifra> } <slovo
ili _> ::= a|...|z|A|...|Z|_ <cifra> ::= 0|1|2|3|4|5|6|7|8|9
```

Primer 9.12. *Korišćenjem EBNF, jezik celih brojeva u dekadnom brojnom sistemu može se opisati sledećim skupom MLF:*

```
<ceo broj> ::= ["+"|"-"]<cifra>{<cifra>} <cifra> ::=
"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Primer 9.13. *Gramatika aritmetičkih izraza može u EBNF da se zapiše kao:*

```
<izraz> ::= <term> {"+" <term>} <term> ::= <faktor> {"*" <faktor>}
<faktor> := "(" <izraz> ")" | <ceo broj>
```

Primer 9.14. *Naredba grananja programskih jezika sa opcionim pojavljivanjem* `else` *grane se može opisati sa:*

```
<if_statement> ::=  if <boolean_expression> then
                         <statement_sequence>
                    [ else
                         <statement_sequence> ]
                    end if ";"
<statement_sequence> ::= <statement> { ";" <statement> }
```

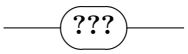
S obzirom na veliki broj različitih proširenja BNF notacije, u jednom trenutku je postalo neophodno standardizovati notaciju. Verzija koja se danas standardno podrazumeva pod terminom EBNF je verzija koju je definisao i upotrebio Niklaus Virt u okviru definicije programskog jezika Pascal, 1977. Međunarodni komitet za standardizaciju definiše ovu notaciju u okviru standarda *ISO 14977*.

- Završni simboli objektnog jezika se navode pod navodnicima kako bi se svaki karakter, uključujući i one koji se koriste kao meta simboli u okviru EBNF, mogli koristiti kao simboli objektnog jezika.
- Pravougaone zagrade `[` i `]` ukazuju na opciono pojavljivanje.
- Vitičaste zagrade `{` i `}` ukazuju na ponavljanje.
- Svako pravilo se završava eksplicitnom oznakom kraja pravila.
- Obične male zagrade se koriste za grupisanje.

Sintaksni dijagrami Sintaksni dijagrami ili sintaksni grafovi ili pružni dijagrami (engl. railroad diagrams) su grafička notacija za opis sintakse jezika koji su po izražajnosti ekvivalentni sa kontekstno slobodnim gramatikama tj. sa BNF. Primeri sintaksnih dijagrama:

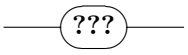
- Sintaksni dijagram za `BinarnaCifra`:

  ```
  <BinarnaCifra> ::= "0" | "1";
  ```

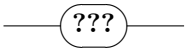
- Sintaksni dijagram za `NeoznacenCeoBroj`:

  ```
  <NeoznacenCeoBroj> ::= ( cifra ) +
  ```

- Sintaksni dijagram za `identifikator`:

  ```
  <identifikator> ::= <slovo> { <slovo> | <cifra> }
  ```

9.10 Načini opisa semantike programskih jezika

Semantika pridružuje značenje sintaksno ispravnim niskama jezika. Za prirodne jezike, ovo znači povezivanje rečenica sa nekim specifičnim objektima, mislima i osećanjima. Za programske jezike, semantika za dati program opisuje koje je izračunavanje određeno tim programom.

Dok većina savremenih jezika ima precizno i formalno definisanu sintaksu, formalna definicija semantike postoji samo za neke programske jezike. U ostalim slučajevima, semantika programskog jezika se opisuje neformalno, opisima zadatim korišćenjem prirodnog jezika. Čest je slučaj da neki aspekti semantike ostaju nedefinisani standardom jezika i prepušta se implementacijama kompilatora da samostalno odrede potpunu semantiku.

Formalno, semantika programskih jezika se zadaje na neki od naredna tri načina:

denotaciona sematika - programima se pridružuju matematički objekti (na primer funkcije koje preslikavaju ulaz u izlaz).

operaciona semantika - zadaju se korak-po-korak objašnjenja kako će se naredbe programa izvršavati na stvarnoj ili apstraktnoj mašini

aksiomatska semantika - opisuje se efekat programa na tvrđenja (logičke formule) koja opisuju stanje programa. Najpoznatiji primer aksiomatske semantike je *Horova logika*.

Na primer, semantika UR mašina definiše značenje programa operaciono, dejstvom svake instrukcije na stanje registara apstraktne UR mašine kao što je navedeno u tabeli **??**.

Pitanja i zadaci za vežbu

Pitanje 9.1. *U čemu je razlika izmedju interptetatora i kompilatora?*

Pitanje 9.2. *Da li se, generalno, brže izvršava program koji se interpretira ili onaj koji je preveden kompilatorom?*

Pitanje 9.3. *Kada je nastao prvi kompilator? Ko ga je napravio i za koji programski jezik?*

Pitanje 9.4. *Navesti primer logičkog jezika, primer funkcionalnog programskoj jezika i primer objektno-orijentisanog programskog jezika.*

Pitanje 9.5. *Kojoj grupi jezika pripada LISP?* _____
Kojoj grupi jezika pripada PROLOG? _____
Kojoj grupi jezika pripada C++? _____

Pitanje 9.6. *Na kojem formalizmu izračunavanja je zasnovan jezik LISP?* _____

Pitanje 9.7. *Na kojem logičkom metodu je zasnovan jezik PROLOG?*

Pitanje 9.8. *U objektno-orijentisanoj paradigmi, šta čini jednu klasu?*

Pitanje 9.9. *U objektno-orijentisanoj paradigmi, šta, osim polja podataka, sadrže objekti?*

Pitanje 9.10. *Navesti šest faza u prevođenju programskih jezika.*

Pitanje 9.11. *U kom vidu se opisuju pravila koja se primenjuju u fazi sintaksne analize?*

Pitanje 9.12. *Šta je to leksička analiza?*

Pitanje 9.13. *Navesti niz leksema i niz odgovarajućih tokena koji su rezultat rada leksičkog analizatora nad izrazom* `a=3;`

Pitanje 9.14. *Kako se zovu programi koji vrše leksičku analizu?*

Pitanje 9.15. *U kom vidu se opisuju pravila koja se primenjuju u fazi leksičke analize?*

Pitanje 9.16. *Na kom algoritmu (na kom apstraktnom automatu) se zasniva rad programa za leksičku analizu?*

Pitanje 9.17. *Šta je to* `lex`*?*

Pitanje 9.18. *Koja analiza se u kompilaciji programa vrši nakon leksičke analize?*

Pitanje 9.19. *Šta je to sintaksna analiza?*

Pitanje 9.20. *Kako se zovu programi koji vrše sintaksnu analizu?*

Pitanje 9.21. *U kom vidu se opisuju pravila koja se primenjuju u fazi sintaksne analize?*
Na kom algoritmu (na kom apstraktnom automatu) se zasniva njihov rad? _____

Pitanje 9.22. *U kom vidu se opisuju pravila koja se primenjuju u fazi sintaksne analize?*

Pitanje 9.23. *Na kom algoritmu (na kom apstraktnom automatu) se zasniva rad programa za sintaksnu analizu?*

Pitanje 9.24. *Šta je* `yacc`*?* _____

Pitanje 9.25. *Regularni jezici su obavezno i kontest-slobodni? (da/ne) Kontest slobodni jezici su obavezno i regularni? (da/ne) Metajezik BNF ima istu izražajnost kao regularni jezici? (da/ne) Metajezik BNF ima istu izražajnost kao kontekst-slobodni jezici? (da/ne) Metajezik BNF ima istu izražajnost kao metajezik EBNF (da/ne)*

Pitanje 9.26. *U opisu regularnih jezika, kako se označava ponavljanje jednom ili više puta?*

Pitanje 9.27. *Navesti tri reči koje pripadaju reglarnom jeziku* `[a-z][0-9]+[0-9]`

Pitanje 9.28. *Zapisati jezik studentskih naloga (npr.* `mi10123`*,* `mr10124`*,* `aa10125`*) na studentskom serveru Alas u vidu regularnog izraza.*

Pitanje 9.29. *Navesti regularni izraz koji opisuje niske u kojima se na početku mogu (a ne moraju) pojavljivati cifre (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi neprazan niz cifara.*

Pitanje 9.30. *Navesti regularni izraz koji opisuje niske u kojima se na početku može (a ne mora) pojaviti neka cifra (od 0 do 9), zatim sledi neprazan niz karaktera* `a`*, a zatim sledi niz (moguće prazan) karaktera* `b`*.*

Pitanje 9.31. *Navesti regularni izraz koji opisuje neprazne konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a`*, a zati sledi neprazan niz cifara.*

Pitanje 9.32. *Navesti regularni izraz koji opisuje konačne niske u kojima se na početku može (a ne mora) pojaviti karakter* `a` *ili karakter* `b`*, a zati sledi niz cifara (moguće i prazan).*

Pitanje 9.33. *Navesti regularni izraz koji opisuje neprazne konačne niske u kojima se pojavljuju ili samo karakteri* `a` *ili samo karakteri* `b`*.*

Pitanje 9.34. *Navesti regularni izraz koji opisuje neprazne konačne niske u kojima se pojavljuju samo karakteri* `a` *i* `b`*.*

Pitanje 9.35. *Od čega dolazi skraćenica EBNF?*

Pitanje 9.36. *Kako se u EBNF-u, zapisuje izraz koji se može ponavljati proizvoljan broj (nula ili više)?*

Pitanje 9.37. *Koji od sledećih metajezika je najizražajniji (1) kontekst-slobodne gramatike; (2) BNF; (3) EBNF; (4) svi navedeni metajezici imaju istu izražajnost.*

Pitanje 9.38. *U EBNF-u, izraz koji se može ponavljati proizvoljan broj (nula ili više) puta zapisuje se:*

Pitanje 9.39. *Skup jezika koji se mogu opisati kontekst-slobodnom gramatikom je širi/jednak/manje širok od skupa jezika koje se mogu opisati pomoću BNF. Skup jezika koji se mogu opisati kontekst-slobodnom gramatikom je širi/jednak/manje širok od skupa jezika koje se mogu opisati pomoću EBNF.*

Deo IV

Računari i društvo

Glava 10

Istorijski i društveni kontekst računarstva

Technologies cannot be divorced from a social framework. Computer scientists need to be educated to understand some of the complex linkages between the social and the technical . . . computer science education should not drive a wedge between the social and the technical.

Understanding of the impact of computer technology on society.

- pre prvih računara
- prvi računari i prve primene
- epoha personalnih računara

Social Impact: The social impact of artificial intelligence is explored by examining public perceptions and the potential social implications of existing AI technology, not only from the point of view of the responsibilities of the developer, but also from the point of view of society and how it has been influenced by AI. To increase the student's awareness of the public perceptions and of the potential social implications of Artificial Intelligence.

Impact of computers–Is the technology neutral?

10.1 Društveni značaj računarstva

10.2 Računarstvo kao naučna disciplina

10.3 Rizici i pouzdanost računarskih sistema

Risks, Liabilities, and Bias Considerations. Risks inherent in any software application. Misuse and misunderstanding as well as developer bias. An empirical study of the evaluation of a piece of software is included to sensitize students to identifying and eliminating bias during the design phase. To sensitize the software developer to the kinds of biases present in each of us that could be passed on to the software we develop, and to point out innate risks in software applications

Liability and Privacy: The kinds of risks that are intrinsic in the development of any computer application and how these risks can be taken into account during development are discussed. Risks include software or hardware bugs, unforeseen user interactions, security risks, violations of privacy, unethical uses, and inappropriate applications of the system. Protecting data and privacy with respect to a database management system are emphasized. To provide a clear understanding of risks that a developer faces with respect to his or her software, and to emphasize the importance of protecting the data from misuse or unauthorized access

Reliability of computer systems: errors, major and minor failures.

10.4 Privatnost i građanske slobode

- Freedom of speech and press – censoring the Internet

- Govern and private use of personal data.
- Encryption Policy
- Computers in the workplace: effects on employment, telecommuting, employee monitoring, e-mail privacy.

Glava 11

Pravni aspekti računarstva

11.1 Intelektualna svojina i licence

- Intellectual property including copyright, patents, trademarks, and trade secrets.
- The Concept of Fair Use.
- Computer Regulatory Measures/Laws
- Open source, public-ware, share-ware
- Obeveze poslodavca prema zaposlenima, obaveze zaposlenog prema poslodavcu.
- Zakoni različitih zemalja, uvoz/izvoz softverskih proizvoda

11.2 Piraterija i računarski kriminal

- Finansijski računarski kriminal,
- zloupotreba mejla i spam
- neovlašćeni upadi u sisteme
- distribuiranje nelegalnih sadržaja itd.

Glava 12

Ekonomski aspekti računarstva

Glava 13

Etički aspekti računarstva

Develop a code of ethics for computing professionals by presenting codes of ethics of other professionals.

- Odgovornost profesionalca u računarstvu Responsibility of the Computer Professional: Personal and professional responsibility is the foundation for discussions of all topics in this subject area. The five areas to be covered under the responsibility of the computer professional are:
  - 1) why be ethical?
  - 2) major ethical models,
  - 3) definition of computing as a profession, and
  - 4) codes of ethics and professional responsibility for computer professionals.
- Ethical claims can and should be discussed rationally,
- Osnovi etičke analize: Basis Skills of Ethical Analysis: Five basic skills of ethical analysis that will help the computer science student to apply ethics in their technical work are:
  - 1) arguing from example, analogy, and counter-example,
  - 2) identifying stakeholders in concrete situations,
  - 3) identifying ethical issues in concrete situations,
  - 4) applying ethical codes to concrete situations, and
  - 5) identifying and evaluating alternative courses of action.
- Osnovi socijalne analize: Basic Elements of Social Analysis: Five basic elements of social analysis are:
  - 1) the social context influences the development and use of technology,
  - 2) power relations are central in all social interaction,
  - 3) technology embodies the values of the developers,
  - 4) populations are always diverse, and
  - 5) empirical data are crucial to the design and development processes.

Ethical issues for computer professionals
A wide variety of issues from privacy and security to intellectual property, democracy and freedom of expression, equal opportunities, globalisation, use and misuse and many other issues.

- Ethical and Legal Issues of Data Data Protection legislation, security and privacy of data issues.

- Ethical and Legal Issues of Information Systems Organisational and social issues of IS, their construction and their effects. Security and privacy issues also enter here, as do legal and issues of professional responsibility
- Professional Responsibility, Codes of Conduct

Mikro-situacije:

- A contract requires an experienced Web designer to develop an e-business system. You have just been awarded the contract but you failed to mention that your sole knowledge was second-hand and that you had not worked on an e-business system before. During this contract, you decide to accept a better-paid contract that requires you to start immediately.
- You are working as a systems designer for a company producing radiation monitoring equipment. You feel that there are faults in the design, but your manager rejects your concerns.
- You are an agent for a software tools producer, that has a product DoItAll. You are being paid to advise an inexperienced small business about developing their software, so you recommend that they must purchase a DoItAll tool.
- You are working as a contractor for a small company and have become aware that some of their software might be unlicensed. You are also concerned about the safeguard of their customers' data. You are currently hoping that your contract will be extended.
- You work for a software house that specialises in helping organisations to set up their E-commerce site. You are worried about the following ethical issues. You are aware that the client is inexperienced in E-commerce and that the client plans to launch its publicity for its new site in time for the Christmas market. You, as a consultant, have just been moved on to this project. You think that the project will be too late for your client's Christmas marketing campaign and that, although in line with the client's specifications, the level of safety checks in the specification seems inadequate. You realise that the contract with the client is for a fixed price.