UNIVERZITET METROPOLITAN BEOGRAD

KI102 - OSNOVE PROGRAMIRANJA

Uvod u algoritme i tehnike rešavanja problema

Lekcija 03

PRIRUČNIK ZA STUDENTE

KI102 - OSNOVE PROGRAMIRANJA

Lekcija 03

UVOD U ALGORITME I TEHNIKE REŠAVANJA PROBLEMA

- Uvod u algoritme i tehnike rešavanja problema
- Poglavlje 1: Osnovi o algoritmima
- Poglavlje 2: Postupak rešavanja problema
- Poglavlje 3: Apstrakcija upravljanja
- Poglavlje 4: Tehnike rešavanja problema
- Zakljucak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

Uvod

UVOD

U okviru ove lekcije naučićemo sledeće:

Računar nam omogućava da postavljene zadatke obavimo što brže, efikasnije i tačnije nego što bi te iste zadatke uradili sami i ručno — mada neke čak ne možemo da rešimo ni ručno. Kako bi od moćne mašine kao što je računar napravili jedan koristan alat, neophodno je da ga isprogramiramo. To znači da mi moramo da specificiramo računaru šta želimo da on uradi za nas i na koji način. To radimo kroz programiranje.

Računar naime nije inteligentan. On ne može da analizira problem i da onda osmisli rešenje za njega. Čovek (programer) je taj koji mora da analizira problem, da razvije skup instrukcija za postavljeni problem, a tek onda da prosledi računaru te instrukcije koje on treba da izvrši. Jednom kada napišemo program koji se izvršava na računaru, računar taj program može da izvši nebrojano puta i to veoma brzo i konzistentno, i na taj način nas oslobađa od ponavljanja dosadnih poslova i zadataka.

Da bi napisali program koji će računar da izvrši, moramo da sprovedemo proces koji se sastoji iz nekoliko faza i koraka. U okviru ovog kursa ćemo se upravo baviti tim koracima, kao i metodama koje treba koristiti da bi se postavljeni problem rešio što efikasnije.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 1

Osnovi o algoritmima

UVODNA RAZMATRANJA

Algoritam je opis za rešavanje nekog problema. To je konačna i precizno definisana procedura (niz dobro definisanih pravila) kojom se ulazne vrednosti transformišu u izlazne

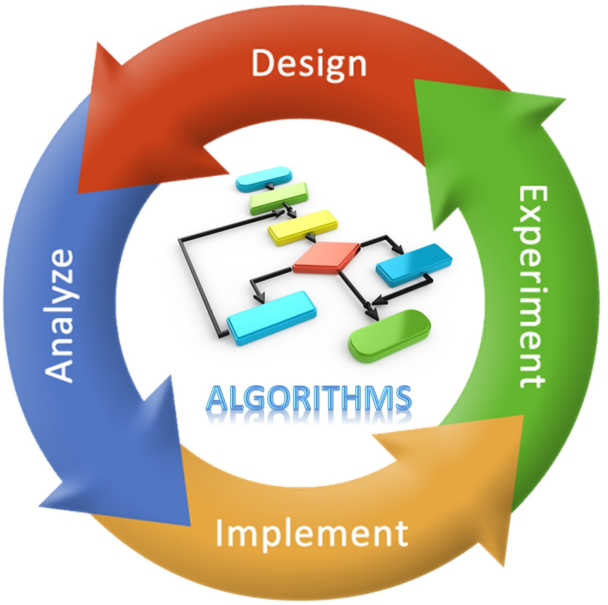
Algoritam je opis za rešavanje nekog problema. Reč potiče od prezimena persijskog matematičara Al Horezmija. Oko 1600. godine u Evropi, algoritam je bio izraz koji opisuje način računanja decimalnim brojevima. U novije vreme, algoritam je pojam koji se gotovo isključivo vezuje za informatiku i, obično, podrazumeva se da je u pitanju procedura za obavljanje nekog posla.

Algoritam je konačna i precizno definisana procedura kojom se opisuje izvršavanje nekog postupka (Slika-1). Algoritam, uopšteno, možemo smatrati kao uputstvo kako rešiti neki zadatak ili problem. Tako se i uputstvo za slanje čoveka na Mesec i uputstvo za pravljenje ruske salate sastoji od niza koraka, postupaka, koje treba uraditi i koji vode ispunjenju cilja ili rešavanju problema. Međutim, korektno izvršavanje svakog koraka ne rešava zadatak ako je algoritam bio pogrešan.

Različiti algoritmi mogu rešiti isti problem različitim nizom postupaka uz manje ili više napora, za kraće ili duže vreme. S obzirom da je rezultat isti, algoritmi se mogu porediti po svojoj efikasnosti, brzini ili složenosti. Algoritmi mogu biti prikazani ili realizovani na više načina:

- prirodnim jezikom (razumljiv samo govornicima tog jezika)
- grafički, dijagramom toka (blok-dijagramom) ili strukturnim dijagramima ili

- tekstualno - pseudokodom, veštačkim precizno definisanim jezikom koji liči na programski jezik
- odgovarajućim programskim jezikom.



Slika 1.1 Koraci u konstrukciji pravilnog algoritma

OSOBINE, IMPLEMENTACIJA I PREDSTAVLJANJE ALGORITAMA

Osnovne osobine algoritama su diskretnost, rezultativnost, determinisanost i masovnost. Algoritmi se obično predstavljaju na dva načina: grafički i pomoću pseudokoda

Algoritmi poseduju sledeće osobine:

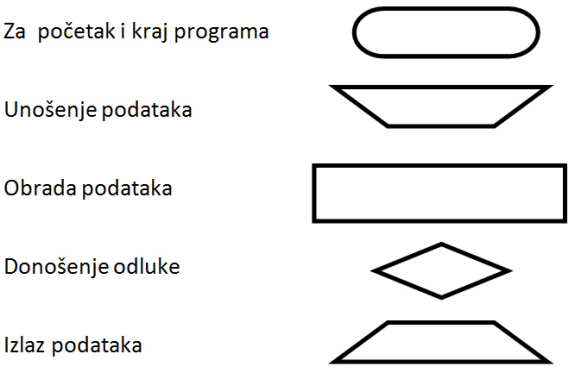
- diskretnost — u odvojenim koracima se izvršavaju diskretne operacije algoritma koji vode ka konačnom cilju;
- rezultativnost — označava sposobnost algoritma da posle konačnog broja koraka daje izlazne podatke;
- determinisanost — za iste ulazne podatke algoritam uvek generiše iste vrednosti na izlazima;
- masovnost — algoritam je primenljiv na veći broj ulaznih vrednosti.

Implementacija

Analiza i proučavanje algoritama je jedna oblast računarstva i često se obavlja apstraktno bez upotrebe konkretnog programskog jezika. Nalik sličnim matematičkim disciplinama u ovim oblastima se izučavaju zakonitosti i principi algoritama, a ne konkretne implementacije. Algoritam se ipak zapiše na nekakav način i to najčešće upotrebom pseudokoda, opšteg jezika za opis algoritma.

Predstavljanje algoritma

Na Slici-2 su prikazani osnovni elementi grafički predstavljenih algoritama.



Slika 1.2 Osnovni elementi za predstavljanje algoritama

Alternativa dijagramima je *pseudokod*. *Pseudokod* (engl. pseudocode) je tekstualna reprezentacija algoritma koja aproksimira konačni izvorni kod. U nastavku je dat primer pseudokoda jednog algoritma:

```
DECLARE CHARACTER ch
DECLARE INTEGER count = 0
DO
  READ ch
  IF ch IS '0' THROUGH '9' THEN
    count++
  END IF
UNTIL ch IS '\n'
PRINT count
END
```

Poglavlje 2

Postupak rešavanja problema

OSNOVI REŠAVANJA PROBLEMA

Polje računarskih nauka (computer science) se svodi na rešavanje problema korišćenjem računara

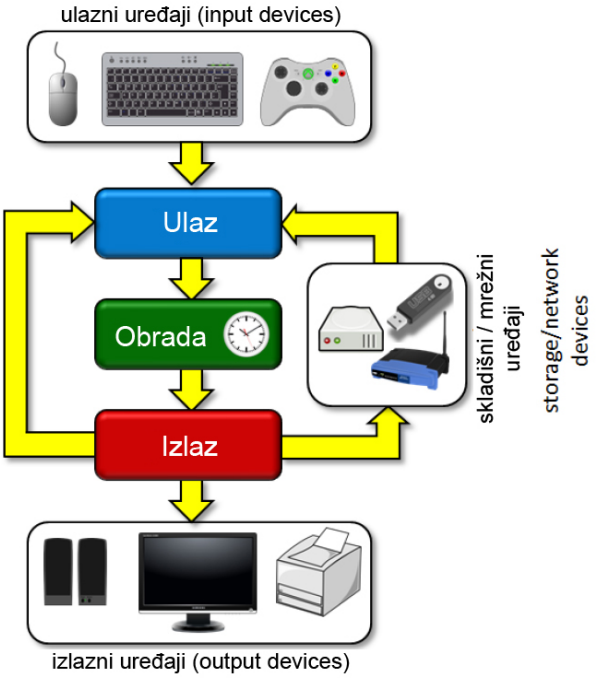
Bez obzira na oblast istraživanja, polje računarskih nauka (computer science) se svodi na rešavanje problema korišćenjem računara. Problemi koje želimo da rešimo mogu da dolaze iz bilo koje sfere realnog sveta ili eventualno iz apstraktnog sveta. U tu svrhu je neophodno je da imamo standardni sistematski pristup da bi rešili ove probleme.

Pošto ćemo koristiti računare da bi rešili probleme, neophodno je da prvo razumemo model procesiranja informacija u računaru. Model prikazan na Slici-1 pretpostavlja računar sa jednim procesorom (CPU - Central Processing Unit). Mnogi računari današnjice imaju i više procesora, tako da samo možete da zamislite koliko puta će se duplirati ovaj model u okviru računara.

Tipičan računar sa jednim procesorom procesira informacije kao na Slici-1. Problemi se rešavaju pomoću računara uzimanjem informacija od korisnika (npr, tastatura ili miš), procesiranjem ulaza i proizvodnjom neke vrste izlaza (npr, slika, test, zvuk). Ponekad ulazni i izlazni podaci mogu biti smešteni na hard disku ili na mrežnim uređajima.

U pogledu rešavanja problema, mi ćemo primeniti prethodni model na način da ćemo pretpostaviti da programu dajemo neke ulazne podatke koji su neophodni da bi se proizveo željeni izlaz ili rešenje problema. Da bi razumeli suštinu dovoljno je da razumemo postupak sa Slike-1 iako je on znatno prostiji od realnih situacija.

Za veće i složenije probleme, neophodno je da izvršavamo iteracije (tj. ponavljanja) kroz faze ulaz/obrada/izlaz više puta za redom, proizvodeći posredna (prelazna) rešenja na putu kojim rešavamo deo našeg problema, ali ne obavezno i ceo problem .



Slika 2.1 Postupak rešavanja problema

Za prosta izračunavanja, prethodni model je više nego dovoljan.

PROCES REŠAVANJA PROBLEMA

Rešavanje problema je sekvencijalni (korak po korak) proces analiziranja informacija koje se odnose na datu situaciju i generisanje odgovarajućih odziva

Najinteresantni deo prethodnog procesa je ustvari proces *"rešavanja problema"*, pa ćemo se u nastavku baviti njime. Postoji puno načina da se definiše šta je ustvari *"rešavanje problema"*. U nastavku je jedan:

Rešavanje problema je sekvencijalni (korak po korak) proces analiziranja informacija koje se odnose na datu situaciju i generisanje odgovarajućih odziva (response options).

Postoji 6 koraka koje treba izvršavati redom da bi ste uspeli da rešite neki problem:
1. Razumevanje problema
2. Formulisanje modela
3. Razvoj algoritma
4. Pisanje programa
5. Testiranje rešenja
6. Evaluacija (ocena) rešenja problema

Razmotrimo prost primer da bi razumeli kako se proces ulaz/obrada/izlaz izvršava za jedan prost problem:

Primer: Odrediti prosečnu ocenu za sve studente u razredu (klasi).

1. **Ulaz**: učitati ocene svih studenata … kucanjem na tastaturi ili učitavanjem ocena sa USB fleša ili sa har diska.
2. **Obrada**: sabrati sve ocene i pronaći prosečnu ocenu.
3. **Izlaz**: oštampati poruku na monitoru, poslati štampaču, ili sačuvati na USB fleš ili hard disk … ili kombinacija bilo čega navedenog.

Kao što možete da vidite, problem je veoma lako rešen jednostavnim učitavanjem podataka, sračunavanjem onoga što je neophodno i formiranjem izlaza koga zatim štampamo.

Analizirajmo sada svaki od 6 koraka u rešavanju problema na prethodno opisanom primeru određivanja srednje vrednosti ocena.

KORAK 1: RAZUMEVANJE PROBLEMA

Pre svega budite sigurni da u potpunosti razumete problem koji treba da rešite

Deluje čudno, ali prvi korak u rešavanju bilo kog problema je da budete sigurni da u potpunosti razumete problem koji treba da rešite. Ono što je neophodno je da znamo:

- **Koji ulazni podaci / informacije su nam dostupni?**
- **Šta ti podaci predstavljaju ?**
- **U kom formatu su podaci zapisani ?**
- **Da li slučajno nešto nedostaje ?**
- **Da li imamo sve što je potrebno da bi rešili problem ?**
- **Koju izlaznu informaciju pokušavamo da kreiramo ?**
- **U kom obliku želimo da predstavimo rezultate … kao tekst, sliku, grafik … ?**
- **Šta ćemo sve morati da izračunamo da bi došli do rešenja ?**

Kada je u pitanju naš primer, potpuno nam je jasno da su ulazni podaci gomila ocena. Ali, moramo da razumemo format zapisa ocena. Svaka ocena može da bude broj od 0 do 100 ili može da bude slovo od A+ do F. Ako je broj, ocena može biti ceo broj poena, npr. 73, ili može da bude decimalan broj kao što je 73.42. Neophodno je da razumemo format u kom su zapisane ocene kako bi mogli da rešimo postavljeni problem.

Takođe treba da uzmemo u obzir i slučaj kada ocena nedostaje. Šta ako nemamo broj poena za svakog studenta (npr. neki student nije polagao test) ? Da li želimo da imamo mogućnost da uključimo tu osobu u naš prosek (tj, dobio je 0 poena) ili da ga ignorišemo kada računamo prosek ?

Takođe treba da razumemo kakav izlaz želimo da imamo iz programa. Ponovo se vraćamo na problem sa formatiranjem. Da li da kao izlaz štampamo ceo broj ili decimalan broj, ili eventualno ocenu? A možda želimo da prikažemo grafikon. Sve to je naš mogući izbor.

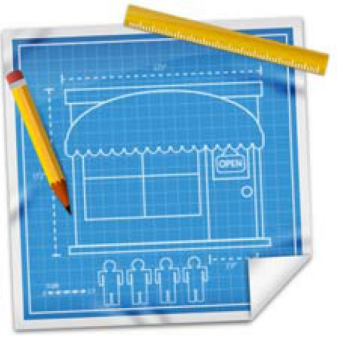
Konačno, treba da razumemo vrstu obrade koju treba da izvršimo nad podacima. To nas upravo vodi do sledećeg koraka u rešavanju problema.

KORAK 2: FORMULISANJE MODELA

U cilju pravilnog formulisanja modela, neophodno je da potpuno razumemo podatke koji su nam dostupni.

Sada je neophodno da razumemo deo procesiranja (obrade) našeg problema.

Mnogi problemi mogu da se iscepkaju na u manje probleme koji zahtevaju neku vrstu prostih matematičkih izračunavanja u cilju obrade (procesiranja) podataka. U našem primeru, mi treba da izračunamo prosečan broj poena koje smo učitali sa ulaza. Stoga je neophodno da poznajemo model (ili formulu) za izračunavanje prosečne vrednosti gomile raznih brojeva. Ako ne postoji takva"*formula*" onda moramo sami da je razvijemo. Veoma često, problem se podeli na prosta sračunavanja koja uglavnom dobro razumemo. Ali, ponekad, u slučaju da smo zaglavljeni i ne znamo da rešimo problem, neophodno je da formule i jednačine potražimo u nekoj knjizi ili na internetu.



Slika 2.2 Formulisanje modela

U cilju formulisanja modela, neophodno je da potpuno razumemo podatke koji su nam dostupni. Pod pretpostavkom da su ulazni podaci gomila celih ili decimalnih brojeva $x_1,x_2,\ldots,x_n$ koji predstavljaju ocene u procentima, mi možemo da koristimo sledeći model za izračunavanje (računski model):

$$\textbf{Average}_1 = (x_1 + x_2 + x_3 + \ldots + x_n) / n$$

pri čemu će broj biti u opsegu od 0 do 100.

Ova jednačina je sasvim logična pod pretpostavkom da znamo formulu za izračunavanje srednje vrednosti gomile brojeva.

KORAK 2: FORMULISANJE MODELA - PRIMER

Da bi mogli da generišemo odgovor na postavljeni problem neophodno je da razumemo koji je pravilan način korišćenja dostupnih podataka

Prethodno opisani pristup neće da radi ako su ocene predstavljene slovima, kao što su na primer **B-**, **C**, **A+**, **F**, **D-**, itd.. jer mi ne možemo da izvršimo operacije sabiranja i deljenja kada

radimo sa slovima. U ovom koraku rešavanja problema se mora osmisliti način da se napravi prosečna vrednost skupa slova, pa je neophodno dodatno razmišljanje o problemu.

Nakon malo razmišljanja i mozganja, možemo da se odlučimo na to da dodelimo odgovarajući broj svakom slovu (znaku) koji učitavamo sa ulaza, i to na sledeći način (Slika-3):

$A^+ = 12$	$B^+ = 9$	$C^+ = 6$	$D^+ = 3$	$F = 0$
$A = 11$	$B = 8$	$C = 5$	$D = 2$	
$A^- = 10$	$B^- = 7$	$C^- = 4$	$D^- = 1$	

Slika 2.3 Primer dodeljivanja odgovarajuće vrednosti svakom znaku ocena za koje računamo prosek

Ako pretpostavimo da su ovi novo dodeljeni brojevi označeni sa $y_1, y_2, \ldots, y_n$, onda možemo da koristimo sledeći računski model:

Average2 = $(y_1 + y_2 + y_3 + \ldots + y_n) / n$

pri čemu će rezultat biti neki broj u opsegu od 0 do 12.

A kada je u pitanju štampanje izlaznih podataka, ukoliko želimo da odštampamo procentualni rezultat, onda možemo da ili koristimo direktno vrednost *Average1* ili da koristimo (*Average2 / 12*), u zavisnosti od vrednosti koju smo originalno imali na ulazu. Ako želimo kao izlaz da oštampamo slovo (ili ocenu), onda bi smo morali da koristimo (*Average1/100*12*) ili (*Average1*0.12*) ili *Average2* a da onda to preslikamo u neku *"lookup tabelu"* koja nam omogućava da pogledamo koji znak odgovara broju iz opsega od 0 do 12.

Nadamo se da ste sada razumeli ovaj korak u procesu rešavanja problema ?

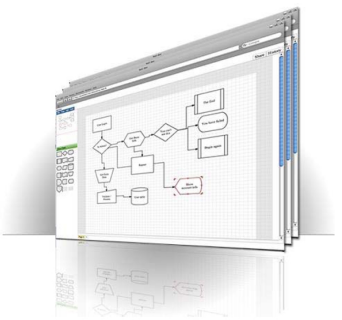
Kao što možete da vidite, sve je u razumevanju načina kako se pravilno koriste dostupni podaci, u cilju generisanja (ili sračunavanja) odgovora na postavljeni problem.

KORAK 3: RAZVOJ ALGORITMA

Algoritam je precizna sekvenca instrukcija za rešavanje postavljenog problema.

Sada kada razumemo problem i kada smo formulisali model, vreme je da da osmislimo precizan plan onogo što želimo da računar uradi za nas.

Algoritam je precizna sekvenca instrukcija za rešavanje postavljenog problema.



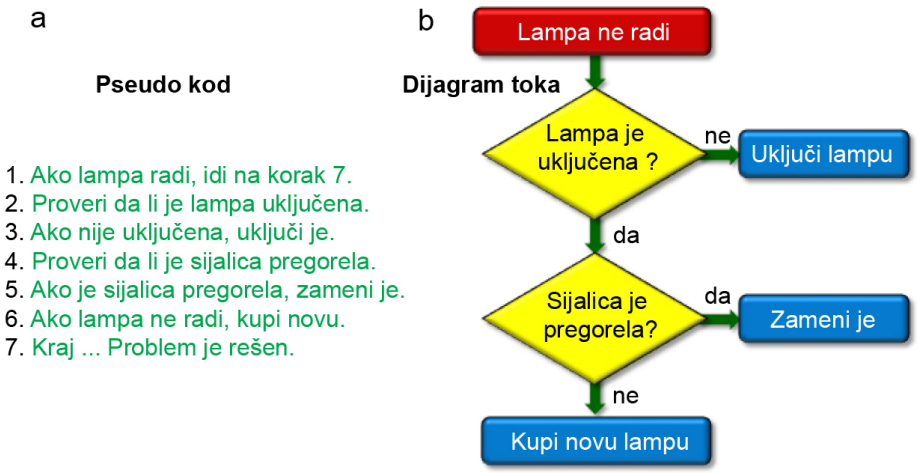
Slika 2.4 Algoritam

Iako u realnosti postoje znatno složeniji algoritmi kod kojih isnstrukcije nisu nužno poređane u sekvenci ili algoritmi možda nemaju konačan broj instrukcija, prethodna definicija je dovoljna da njome opišemo algoritme koje ćemo obraditi u okviru ovog kursa.

Da bi razvili jedan algoritam, neophodno je da predstavimo instrukcije na način koji je razumljiv osobama koje pokušavaju da shvate korake u okviru algoritma.

Dve najčešće korišćene reprezentacije algoritma su (1) *pseudo kod*, ili (2) *dijagram toka* (flow chart).

Uzmimo u obzir sledeći primer (sa Vikipedije) rešavanja problema razbijene lampe. Na desnoj strani Slike-5 je dijagram toka, dok je na levoj strani primer pseudo-koda za rešavanje istog problema:



Slika 2.5 Pseudo kod i dijagram toka

Primetićete sledeće:

Pseudokod je prostija i konciznija sekvenca instrukcija (najčešće napisanih na engleskom jeziku) koju koristimo u cilju rešavanja problema.

KORAK 3: RAZVOJ ALGORITMA: - PSEUDOKOD

Pseudokod je prostija i konciznija sekvenca instrukcija koju koristimo u cilju rešavanja problema.

Pseudokod je često korišćen kao način da se opiše računarski program nekome ko ne zna programiranje. Kada učite da programirate, bitno je da probate sami da napišete pseudo kod jer će vam on pomoći da potpuno razumete problem koji pokušavate da rešite.

Iako su dijagrami toka vizualno veoma privlačni, pseudo kod je često poželjniji izbor za razvoj algoritma jer :

- Nekada je teško da valjano nacrtate dijagram toka, naročito kada ste eventualno napravili greške.
- Mnogo lakše pseudokod staje na list papira, za razliku od dijagrama toka koji može da bude preveliki.
- Pseudokod može biti napisan na način koji je skoro približan realnom programskom kodu, što vam olakšava da kasnije napišete pravi program (npr. u koraku 4).
- Neophodno vam je manje vremena da napišete pseudokod nego što vam je potrebno vremena da nacrtate dijagram toka.

Pseudokod programa će se naravno razlikovati u zavisnosti od osobe koja će ga napisati. Bez obzira na te razlike, uvek će postojati uobičajene kontrolne strukture (tj, elementi) koje će se javljati u bilo čijoj verziji pseudokoda.

KORAK 3: RAZVOJ ALGORITMA - PRIMER

Uobičajene kontrolne strukture su: sequence, condition, repetition, storage i jumping

Uobičajene kontrolne strukture su u nastavku zajedno sa odgovarajućim primerima:

-Lista instrukcija korak po korak (*sequence*):

```
Uveri se da je prekidač uključen
Proveri da li je lampa uključena u struju
Proveri da li je sijalica pregorela
...
```

-Uslovi (*condition*):

```
if lampa nije uključena
    then uključi je
if sijalica je pregorela
    then zameni sijalicu
otherwise kupi novu lampu
```

-Ponavljanje (*repetition*):

```
repeat
    nabavi novu sijalicu
    stavi je u lampu
until lampa radi ili nema više sijalica
repeat 3 times
    isključi lampu
    uključi lampu u drugu utičnicu
    ...
```

-Dodeljivanje vrednosti (*storage*):

```
x <- nova sijalica
    count <- 8
```

-Skakanje (*jumping*):

```
if sijalica radi
    then goto korak 7
```

Iz prethodnih primera možete videti da, za uslovne i ponavljajuće strukture, isečak pseudo koda koji je deo uslova ili koraka ponavljanja je malo uvučen tako da bude jasno da uvučeni kod predstavlja "unurašnje korake" koji pripadaju toj strukturi. Neki programeri koriste vitičaste zagrade da bi naznačili šta je unutar a šta izvan kontrolne strukture, na sledeći način:

```
if(sijalica je pregorela) then{
    zameni sijalicu
}
otherwise{
    kupi novu lampu
}
repeat{
    nabavi novu sijalicu
    stavi je u lampu
}until{lampa radi ili nema više sijalica}
repeat 3 times{
    isključi lampu
    uključi lampu u drugu utičnicu
}
```

Poenta svega je da razumete da postoji puno različitih načina da se napiše pseudokod algoritma.

KORAK 3: RAZVOJ ALGORITMA - PROVERA ALGORITMA

Jedino u slučaju kada ste potpuno sigurni da vaš algoritam nema greške i da rešava postavljeni probleme možete preći na sledeći korak

Bilo da koristite dijagram toka ili pseudokod, potrebno je da testirate vaš algoritam tako što ćete se preslišati dobro i ići korak po korak da bi ste bili sigurni da niste zaboravili neki korak algoritma ili neku specijalnu situaciju. Veoma često ćete pronaći falinku u vašem algoritmu, jer niste uzeli u obzir neku specijalnu situaciju do koje može doći. Jedino u slučaju kada ste potpuno sigurni da vaš algoritam nema greške i da rešava postavljeni probleme možete preći na sledeći korak .

Uzmimo u obzir naš prethodni primer sračunavanja srednje vrednosti skupa od n ocena koje su smeštene u nekom fajlu. Šta mislite, kako treba da izgleda pseudokod? U nastavku je primer kako on može da izgleda u slučaju da imamo primer od n ocena u formi brojeva označenih sa x1 ... xn, koje smo učitali iz fajla:

```
Algoritam: ProsečnaOcena
    postavi sumu ocena na 0.
    učitaj sve ocene x_1 ... x_n iz fajla.
    repeat n times{
        uzmi ocenu x_i
        dodaj x_i na sumu
    }
    izračunaj prosek ocena kao suma/n.
    štampaj prosek ocena.
```

Bilo bi veoma mudro da proverimo kako ovaj algoritam radi sa realnim skupom brojeva. Svaki put kada testiramo naš algoritam upotrebom nepromenljivog skupa ulaznih podataka, mi ustvari izvršavamo jedan slučaj provere algoritma (test case). U cilju provere vašeg algoritam, vi možete da kreirati onoliko slučajeve provere koliko vi želite. U nastavku su neki primeri koje možete da ispobate:
n = 5, x1 = 92, x2 = 37, x3 = 43, x4 = 12, x5 = 71 ... rezultat treba biti 51
n = 3, x1 = 1, x2 = 1, x3 = 1 ..... rezultat treba biti 1
n = 0 ..... rezultat treba biti 0

KORAK 4: PISANJE PROGRAMA

Pisanje programa se često naziva pisanje koda ili implementacija algoritma. Stoga je kod sam po sebi ustvari jedan program.

Sada kada imamo precizan skup koraka neophodnih za rešavanje problema, možemo reći da je najveći deo teškog posla završen. Sledeće što treba je da transformišemo algoritam, definisan u koraku 3, u odgovarajući skup instrukcija koje računar može da razume.

Pisanje programa se često naziva "*pisanje koda*" (writing code) ili implementacija algoritma (implementing an algorithm). Stoga je kod (ili *izvorni kod*, eng. source code) sam po sebi ustvari jedan program.

Bez puno dodatnog pojašnjenja, u nastavku se nalazi program (napisan u jeziku Processing) koji predstavlja implementaciju našeg algoritma za određivanje srednje vrednosti skupa ocena studenata (Slika-6). Primetimo da i pseudokod i kod deluju dosta slično, međutim, kod je teži za čitanje i razumevanje i deluje previše matematički :

Pseudocode	Processing code (i.e., program)

1. set the sum of the grade values to 0.
2. load all grades x_1 ... x_n from file.
3. **repeat n times** {
4. get grade x_i
5. add x_i to the sum
}
6. compute the average to be sum / **n**.
7. print the average.
	`int sum = 0;`
	`byte[] x = loadBytes("numbers");`
	`for (int i=0; i<x.length; i++)`
	`sum = sum + x[i];`
	
	`int avg = sum / x.length;`
	`print(avg);`

Slika 2.6 Pseudo kod i kod primera

Za sada, mi nećemo diskutovati detalje o tome kako je napisan prethodni izvorni kod. To radimo iz razloga što se izvorni kod razlikuje od jednog do drugog programskog jezika. Možda

će vam u početku delovati da je učenje programskog jezika dosta težak zadatak, ali učenje postaje lako kako stičete odgovarajuće praktično iskustvo.

Računar zahteva precizne instrukcije kako bi razumeo šta u stvari želite od njega, tj. šta želite da on uradi. Na primer, ako obrišemo jedan od znakova tačka-zarez (;) iz prethodnog koda, računar će se zbuniti i neće znati šta da uradi jer za njega ovaj znak predstavlja oznaku za kraj jedne instrukcije. Brisanjem jednog od tačka-zarez znaka će se proizvesti situacija zbog koje će vaš program generisati nešto što je u programerskom svetu poznato kao *kompajlerska greška* (compile error).

Kompajliranje (compiling) **je proces konvertovanja programa u skup instrukcija koje računar može da razume i da izvršava.**

Što je veća dužina vašeg programa (što je više linija koda), veće su šanse da se jave kompajlerske greške. Stoga je neophodno da korigujete sve te sitne kompajlerske greške pre nego pređete na sledeći korak u rešavanju programskog problema.

KORAK 5: TESTIRANJE (PROVERA) PROGRAMA:

Što je veća dužina vašeg programa (što je više linija koda) veće su šanse da se jave kompajlerske greške. Njih morate ispraviti da bi ste prešli na sledeći korak

Jednom kada napišete program i kada ste sigurni da možete da ga iskompajlirate, neophodno je da budete sigurni da program rešava problem za čiju namenu je i napravljen i da je rešenje koje program daje tačno.

Izvršavanje programa je proces u kome se računaru naredjuje da odradi iskompajlirane instrukcije.

Kada pokrenete vaš program, u slučaju da je sve u redu, on će proizvesti tačan rezultat i izlaz. Ipak, moguće je da vaš program radi tačno za neki skup ulaznih podataka ali ne za sve skupove podataka. U slučaju da je izlaz (rešenje) vašeg programa netačan, moguće je da niste dobro algoritam pretočili u programski kod. Takođe je moguće da niste osmislili dobar algoritam (u Koraku 3), tako da on ne može da savlada sve situacije koje mogu da se jave. Možda ste neke instrukcije naveli na pogrešnom mestu u nizu instrukcija. Šta god da je nastalo loše sa vašim programom te stvari se uobičajeno nazivaju *bagovi* (kvarovi, ili eng. bugs).

Bagovi (kvarovi) su problemi/greške sa vašim programom koje izazivaju da vaš programprestane sa radom ili da vaš program daje netačna rešenja.

Od vas se, stoga, očekuje da ispravite što je više moguće bagova u programu koje tokom provere rada programa možete da detektujete.

Da bi ste efikasno pronašli bagove u vašem programu, neophodno je da testirate vaš program na veliki broj slučajeva (tzv. test suite). Takođe je dobra ideja da drugi ljudi, po mogućstvu laici u programiranju, testiraju vaš program jer oni mogu da propuste program za ulazne podatke ili za slučajeve koji vama možda ne bi pali na pamet.

Proces pronalaženja i otklanjanja grešaka u vašem programu se naziva debagiranje (debugging) i veoma često ovaj proces oduzima puno vremena, u terminima programerskog posla.

Medjutim, ukoliko utrošite dodatno vreme da pažljivo prođete kroz korake od 1 do 3 vašeg procesa za rešavanje problema, moguće je da ćete značajno smanjiti količinu grešaka u vašem programu pa će proces debagiranja vašeg programa postati značajno lakši i kraći.

Slika 2.7 Testiranje

KORAK 6: EVALUACIJA (OCENA) REŠENJA PROBLEMA:

Nakon svega, neophodno je još jednom razmotriti originalni problem i sve detaljno proveriti

Jednom kada vaš program proizvede rezultat koji deluje da je tačan, neophodno je da još jednom razmotrite originalni problem i da proverite, radi sigurnosti, da li je odgovor koji daje vaš programa predstavlja odgovarajuće rešenje postavljenog problema. Veoma često se dešava da vi tek posle svega shvatite da vaš programaski kod ne rešava problem na način kako ste vi to želeli. Stoga možete shvatiti da je neophodno izvršiti još nekoliko koraka.

Na primer, ukoliko je rezultat programa dugačka lista brojeva, a vi ste možda želeli da prikažete samo deo, ili da identifikujete neke elemente na osnovu podataka, onda samo formiranje liste brojeve možda nije dovoljno dobro. Možda je i potrebno prikazati informacije na način koji vam pomaže da vizualizujete ili interpretirate rezultate u odnosu na postavljeni problem. Možda je u tom slučaju grafik ili dijagram bolje rešenje.

Takođe je moguće da, kada pregledate vaše rezultate, shvatite da su vam potrebni dodatni podaci kako bi ste potpuno rešili problem. Ili, eventualno, je neophodno da doterate rezultate kako bi rešenje problema bilo efikasnije (npr, igra radi ali sporo - neophodno je ubrzati je).

Veoma je bitno zapamtiti da će računar uraditi samo ono što ste mu rekli da uradi, a na vama je da interepretirate rezultate na smislen način i da procenite da li on rešava ili ne rešava originalni problem. Možda će biti neophodno izvršiti ponovo neke korake, možda se čak treba vratiti na korak 1 ispočetka, u slučaju da neki podaci nedostaju pa onda ponoviti sve ostale korake do kraja.

Slika 2.8 Slagalica kao primer evaluacije problema

Zaključak

Stigli smo da kraja. Ovo bi bilo tih 6 koraka kojih se treba držati kako bi ste uspeli da rešite problem korišćenjem računara. Tokom ovog kursa, preporuka je da se držite ovog pristupa bez obzira na tip problema na koji naiđete. Dobra je ideja da s vremena na vreme sami vežbate postupke rešavanja problema, jer ćete tako biti sigurni da ste razumeli ceo proces. Na sledećem slajdu su neni praktični primeri koji će vam pomoći da provežbate prva tri koraka u rešavanju problema. Kasnije, kada započnete da pišete programe u Javi, moći ćete da steknete praktično iskustvo kada su u pitanju koraci od Koraka 4 do Koraka 6.

PRIMERI PROSTIH PROBLEMA

Prikazivanje teksta i sabiranje brojeva od 1 do 9

Prikazivanje teksta

Napraviti program koji ispisuje sledeći tekst:

```
a a^2 a^3
1  1  1
2  4  8
3  9  27
4  16 64
```

Slika 2.9 Ovo treba da program prikaže na monitoru

Objašnjenje: Celokupna slika predstavlja kombinaciju štampanja 5 linija teksta sa odgovarajućim rasporedom karaktera, kako bi se formirao odgovarajući prikaz matrice.

Rešenje:

```
PRINT "a a^2  a^3"
for a = 1,4
    PRINT a, a*a, a*a*a;
```

Sabiranje brojeva od 1 do 9

Napraviti program koji sabira brojeve od 1 do 9:

(1+2+3+4+5+6+7+8+9)

Objašnjenje: Ključna reč for označava brojačku petlju. U ovom slučaju vrednost celobrojne promenljive će uzimati vrednosti počev od 1 i u narednih 9 prolaza kroz blok povećavati svoju vrednost za 1 i dodavati je na sumu iz prethodne iteracije. Na kraju poslednje iteracije dobićemo zbir svih prethodnih, što je suma koja je tražena.

Rešenje:

```
/**
  * Suma svih brojeva od 1 do 9. Suma se svaki put povećava
  * za i koji predstavlja iterator kroz petlju. U svakom
  * krugu petlje i je veći za jedan (i++).
  */
sum = 0;
for i = 1, 9
    sum = sum + i
printl "Zbir je = ", sum
```

PRIMER: ODREĐIVANJE MINIMUMA, MAKSIMUMA I SREDNJE VREDNOSTI NIZA BROJEVA

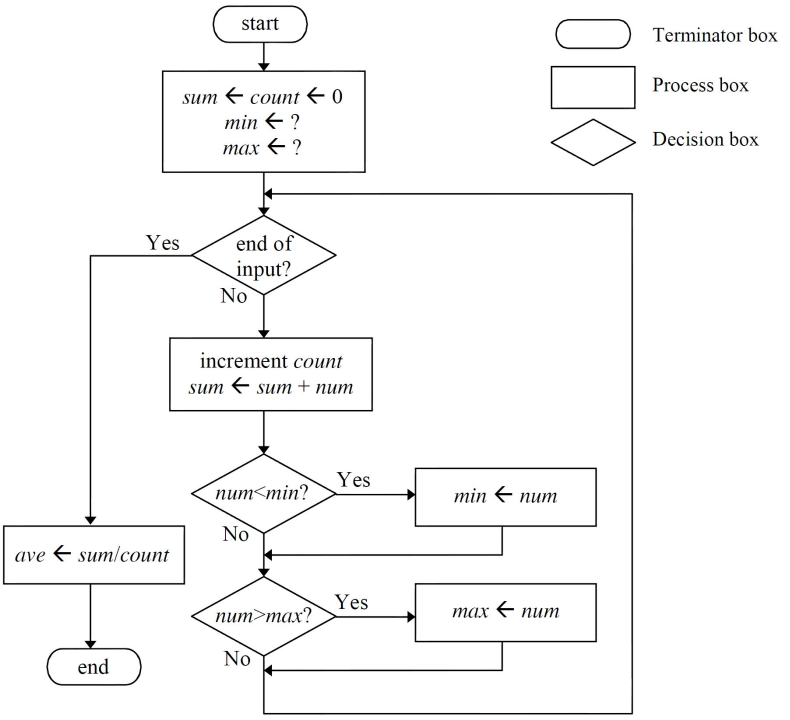
Algoritam za određivanje minimuma, maksimuma i srednje vrednosti liste brojeva

Algoritam 1:

1. Prvo, inicijalizuj *sum* na nulu, min na neki veliki broj, a max na neki mali broj.
2. Onda, unesi brojeve, jedan po jedan.
3. Za svaki uneti broj, dodeli ga promenljivoj *num* i dodaj ga na *sum*.
4. U isto vreme poredi *num* sa *min*, ako je *num* manje od *min*, onda postavi *min* da bude *num*.
5. Slično, poredi *num* sa *max*, ako je *num* veće, onda postavi *max* da bude *num*.
6. Kada su uneti svi brojevi, podeli *sum* sa brojem unetih brojeva i postavi rezultat u *avg*.
7. Kraj.

Algoritam 2:

```
sum <- count <- 0        //count = koliko je brojeva uneto?
min <- ?                 //min = najmanja vrednost
max <- ?                 //max = najveća vrednost
for each uneti broj num
    povećaj count za jedan
    sum <- sum + num
    if num < min then min <- num
    if num > max then max <- num
    avg <- sum/count
```



Slika 2.10 Blok dijagram algoritma za odredjivanje minimuma, maksimuma i srednje vrednosti liste brojeva

PRIMERI ZA SAMOSTALNO VEŽBANJE

Na osnovu svega obrađenog u ovoj sekciji, napisati algoritam za rešavanje sledećih postavljenih problema:

Formulisati model a zatim i razviti algoritam za svaki od problema sa Slike-11. Za svaki od slučajeva, za početak napraviti što prostiji algoritam, a zatim pokušajte da razmišljate o realnim situacijama koje mogu da krenu naopako i u skladu sa tim izvršiti odgovarajuća poboljšanja u vašem algoritmu. Imajte uvek na umu da ne postoji "pravo" rešenje za postavljeni problem. Svaka osoba može da osmisli posebno i jedinstveno rešenje .

a. **Pravljenje sendviča sa džemom ili buterom**

b. **Sastavljanje delića slagalice**

c. **Igranje igre "muzičke stolice"**

d. **Zamena točka na vašem automobilu**

e. **Povratak od škole do kuće**

f. **Pražnjenje kartona Koka Kole iz frižidera**

g. **Kupovina namirnica (od ulaska do izlaska iz radnje)**

Slika 2.11 Primeri praktičnog vežbanja

Poglavlje 3

Apstrakcija upravljanja

UVODNA RAZMATRANJA

Ideja korišćenja najjasnijeg algoritma koji nema u sebi puno nepotrebnih detalja je poznata kao apstrakcija.

Upravo smo obradili osnove kreiranja algoritma u cilju rešavanja postavljenog problema. Međutim, nije baš najjasnije koliko duboko u detalje treba da ide naš algoritam. Na primer, razmotrimo situaciju da smo kod kuće na kuču i da smo žedni. U nastavku je prosto algoritamsko rešenje za naš problem:

```
idi u kuhinju
otvori frižider
izaberi piće
popij piće
```

Međutim, mogli smo da smislimo rešenje koje se bazira na apstraktnijem algoritmu (tj algoritmu sa manje detalja) , kao što sledi:

```
1. uzmi piće
2. popij piće.
```

Ili, možemo da osmislimo algoritam sa mnogo više detalja, koji je naveden u sledećem listingu:

```
ustani sa kauča
idi do kuhinje
otvori frižider
if ima flaša limunade ili soka od jabuke then{
        uzmi flašu
        zatvori frižider
        idi do kredenca
        otvori kredenac
        uzmi čašu
        zatvori kredenac
        sipaj limunadu ili sok od jabuke u čašu
        idi do frižidera
        vrati flašu u frižider
}
otherwise if ima limenka soka then{
        uzmi limenku
        otvori limenku
```

```
}
zatvori frižider
popij
```

Svako od ovih algoritamskih rešenja rešava naš problem. Stoga je pitanje, koji algoritam je najbolji? Odgovor na ovo pitanje nekada nije lak. Po pravilu, želimo da proizvedemo što je prostiji mogući algoritam koji je istovremeno i lak za razumevanje. Dok je bitno da obezbedimo dovoljno detalja da bi bili u mogućnosti da adekvatno opišemo rešenje problema, takođe je bitno ne preopteretiti se sa previše detalja koji će algoritam načiniti pretrpanim i previše komplikovanim.

Stoga, po svojoj prilici, prvi od prikazana tri algoritma će biti dovoljan kao adekvatno rešenje problema. Ova ideja korišćenja najjasnijeg algoritma koji nema u sebi puno nepotrebnih detalja je poznata kao *apstrakcija*.

OSNOVI APSTRAKCIJE

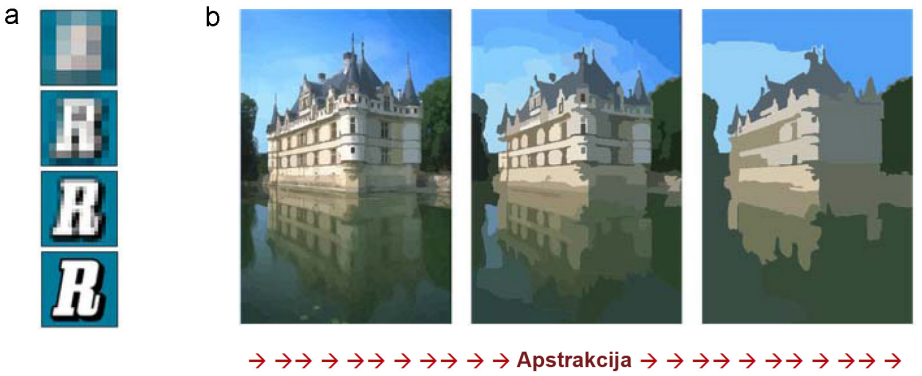
Apstrakcija nam omoguća da se fokusiramo na samo par pojmova u datom trenutku, i da tako shvatimo širu sliku kada je u pitanju rešenje problema

Apstrakcija je bitna jer nam omoguća da se fokusiramo na samo par pojmova u datom trenutku. Ovo nam omogućava da odmah shvatimo "*širu sliku*" kada je u pitanju rešenje problema. Tek nakon toga možemo, u narednom trenutku rešavanja, da "dopunimo" algoritam specifičnim detaljima. Na primer, u prethodnom primeru, iskaz *"uzmi piće iz frižidera"* će biti dovoljan ako taj korak posmatramo kao deo većeg problema, kao što je algoritam koji opisuje uobičajene dnevne aktivnosti osobe u vreme večere. Međutim, ako treba da isprogramiramo isti takav korak u okviru robota, onda će nam trebati mnogo više detalja jer će robot zahtevati znatno precizniji skup operacija pomeranja u cilju izvršavanja takve operacije.

Da bi opisali apstrakciju, analiziraćemo primer slike koja može biti predstavljena pomoću različite rezulocije. Manja rezulucija slike znači manje detalja i veću apstrakciju za razliku od slike veće rezolucije (koja ima više detalja na sebi). Apstrakcija nam omogućava da smanjimo rezoluciju slike u cilju smanjenja neophodnog prostora za skladištenje slike (tj, sakrivanjem detalja) osim ako su ti detalji neophodni (Slika-1a).

Takođe možemo da primetimo ideju o asptrakciji u 3D video igrama, gde se sakrivaju detalji objekata koji su daleko od posmatrača, kada oni trenutno nisu neophodni, sve dok se karakter u igri (robot, čovek, ili humanoid) ne približi dovoljno tim objektima.

To nam, jasno, pomaže da ubrzamo izvršavanje naše igre i smanjimo nered, a istovremeno omogućavamo igraču da se fokusira na važnije objekte scene koji su njemu bliži (Slika-1b):



Slika 3.1.1 a) Smanjenje rezolucije slike smanjuje nivo detalja b) Slika u 3D igri kojoj se smanjuje rezulucija kako se igrač udaljava od objekta

Upravo smo diskutovali o tipu apstrakcije koji omogućava da koraci u našem algoritmu budu ili potpuno apstraktni (tj. na visokom nivou apstrkcije) ili sa puno detalja (tj, nizak nivo apstrkakcije). Ovaj tip apstrakcije se naziva *apstrakcija upravljanja* (constrol apstrakction).

POTPROGRAMI

Potprogram je blok koda koji obavlja specifični podzadatak

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

3.1 Apstrakcija na više nivoa

PREMEŠTANJE DETALJA U POSEBNE PODALGORITME

Uvek je moguce kreirati visoko apstraktan algoritam, sakrivajući detalje u podalgoritme

Često, iako je neophodno imati puno detalja, i dalje je moguće kreirati veoma apstraktan algoritam. Pritom obezbeđujemo da specifičniji detalji budu opisani u nekom podalgoritmu ili podprogramu. Kao rezultat, imamo mogućnost da opišemo algoritam na više nivoa apstrakcije. U našem primeru "*gašenja žeđi*" smo mogli da koristimo detaljniji algoritam, pri čemu bi sakrili najdetaljniji deo algoritma u neki pod-algoritam, kao što je prikazano u sledećem kodu :

```
AlgoritamX: UgasiŽeđ
    ustani sa kauča
    idi do kuhinje
    otvori frižider
```

```
    izvrši PodAlgoritam1
    zatvori frižider
    popij
PodAlgoritam1: UzmiPiće
    if ima flaša limunade ili soka od jabuke then{
        uzmi flašu
        zatvori frižider
        idi do kredenca
        otvori kredenac
        uzmi čašu
        zatvori kredenac
        sipaj limunadu ili sok od jabuke u čašu
        idi do frižidera
        otvori frižider
        vrati flašu u frižider
    }
    otherwise if ima limenka soka then{
        uzmi limenku
        otvori limenku
    }
```

Vidimo da je glavni algoritam sada prilično prost i bez puno detalja. Naravno, primetimo da drugi algoritam, *PodAlgoritam1*, nije nezavistan i ne može samostalno da reši problem. Ovaj podalgoritam, *PodAlgoritam1*, pretpostavlja da osoba stoji ispred friždera i da su vrata friždera otvorena. Mi možemo da zaobiđemo ova ograničenja, na primer, premeštanjem Koraka 3 i 4 iz glavnog algoritma *AlgoritamX* u podalgoritam *PodAlgoritam1*. Primetite sledeće izmene:

```
AlgoritamX: UgasiŽeđ
    ustani sa kauča
    izvrši PodAlgoritam1
    popij
PodAlgoritam1: UzmiPiće
    idi do kuhinje
    otvori frižider
    if ima flaša limunade ili soka od jabuke then{
        uzmi flašu
        zatvori frižider
        idi do kredenca
        otvori kredenac
        uzmi čašu
        zatvori kredenac
        sipaj limunadu ili sok od jabuke u čašu
        idi do frižidera
        vrati flašu u frižider
    }
    otherwise if ima limenka soka then{
        uzmi limenku
        otvori limenku
    }
    zatvori frižider
```

KREIRANJE DODATNIH NIVOA APSTRAKCIJE

Stvari možemo na ovaj način načiniti još apstraktnijim koliko god puta osetimo da je takav postupak neophodan

Nastavljamo sa prethodnim primerom. Vidimo da smo se sada pomerili na još veći nivo apstrakcije sakrivanjem detalja unutar algoritma *PodAlgoritam1*. Mogli smo i da izvršimo još jedan nivo apstrakcije tako što bi dodatno sakrili neke detalja u okviru algoritma *Algoritam1*:

```
AlgoritamX: UgasiŽeđ
    ustani sa kauča
    idi do kuhinje
    izvrši PodAlgoritam1
    popij
PodAlgoritam1: UzmiPiće
    otvori frižider
    if ima flaša limunade ili soka od jabuke then{
        izvrši PodAlgoritam2
    }
    otherwise if ima limenka soka then{
        uzmi limenku
        otvori limenku
    }
    zatvori frižider
```

```
PodAlgoritam2: SipajPiće
    uzmi flašu
    zatvori frižider
    idi do kredenca
    otvori kredenac
    uzmi čašu
    zatvori kredenac
    sipaj limunadu ili sok od jabuke u čašu
    idi do frižidera
    vrati flašu u frižider
```

Primetite kako je podalgoritam *PodAlgoritam1* postao apstraktniji i lakši za čitanje. Stvari možemo na ovaj način načiniti još apstraktnijim koliko god da puta osetimo da je takav postupak neophodan. Ali, postavlja se pitanje dokle, tj. kada da odlučimo da stanemo sa pomeranjem nivoa apstrakcije? Pa, ne postoji tačno definisano pravilo. Mada, kada pod-algoritam deluje da je dovoljno mali da bi bio razumljiv, ili kada je usklađen sa dobro definisanom grupom akcija iz realnog života, onda je verovatno dobra ideja da ne cepkamo više naš pod-algoritam na neke dodatne podalgoritme.

ZAŠTO JE KORISNO KREIRATI POD-ALGORITME?

Uvek postoji mogućnost da deo vašeg algoritma bude korišćen od strane nekog drugog algoritma, naročito u situacijama kad je neke korake neophodno ponoviti više puta

Treba imati na umu da uvek postoji mogućnost da deo vašeg algoritma može biti korišćen od strane nekog drugog algoritma. Na primer, šta ako želimo da imamo algoritam koji će postaviti čašu na kuhinjski sto, u toku pripreme stola jer nam dolaze gosti na večeru ? U takvoj situaciji ćete uvideti da je neophodno da izvršite iste korake ponovo, kao što je prikazano u linijama od 3 do 6 podalgoritma *PodAlgoritam2*.

Stoga je možda dobra ideja napisati podalgoritam *PodAlgoritam3* koji ima sledeću strukturu:

```
PodAlgoritam3: UzmiČašu
    idi do kredenca
    otvori kredenac
    uzmi čašu
    zatvori kredenac
```

Nakon toga, naravno, možemo da koristimo ovaj podalgoritam *PodAlgoritam3* u okviru podalgoritma *PodAlgoritam2*, kao i u okviru našeg novog algoritma koji posprema sto na sledeći način:

```
PodAlgoritam2: SipajPiće
    uzmi flašu
    zatvori frižider
    izvrši PodAlgoritam3
    sipaj limunadu ili sok od jabuke u čašu
    idi do frižidera
    vrati flašu u frižider
```

```
AlgoritamY: PostaviStoZa4Osobe
    idi do kuhinje
    repeat 4 times{
        izvrši PodAlgoritam3
        postavi čašu na sto
        izvrši podAlgoritam4  //Sličan algoritam za uzimanje tanjira iz kredenca
        postavi tanjir na sto
        izvrši PodAlgoritam5
        postavi nož i viljušku na sto   //Sličan algoritam za uzimanje noža i
viljuške iz fioke
    }
    vrati se na kauč
```

Nadamo se da ste sada uvideli koliko praktična i moćna apstrakcija može da bude u cilju kreiranja prostijeg, čitkijeg i, što je najbitnije, razumljivijeg algoritma za onog ko pokušava da ga izuči.

U oblasti računarskih nauka, ovi podalgoritmi su poznati pod drugim i dobro poznatim nazivima. Uglavnom se podalgoritmi nazivaju *moduli*, *funkcije* ili *procedure*.

3.2 Moduli, funkcije i procedure

UVODNA RAZMATRANJA

Podalgoritmi se u oblasti računarskih nauka uglavnom nazivaju moduli, funkcije ili procedure.

Podalgoritmi se u oblasti računarskih nauka uglavnom nazivaju moduli, funkcije ili procedure.

Naime, nije dobra ideja numerisati redom podalgoritme brojevima redom od 1, itd... već je umesto toga pogodnije podalgoritmima dati imena koja imaju neko značenje. Usvojeno je da pri imenovanju funkcija i procedura treba da koristite karaktere tipa slova, brojeve i donju crtu (t.j _) ali nikako praznine niti znakove interpunkcije. Takođe, prvi karakter u nazivu bi najpoželjnije trebalo da bude malo slovo. Ukoliko se u nazivu koristi više reči onda sve reči osim početne treba da počinju velikim slovom. Poslednje, mi često koristimo zagrade (tj. *()*) nakon naziva funkcije ili procedure da bi naznačili da se radi o podalgoritmu. Ovo ćemo videti kasnije. Takođe, trebate izabrati logične nazive ali koji nisu previše dugački. U nastavku su neki logični nazivi koje možemo da koristimo kod našig podalogoritama:

- *PodAlgoritam1*: *izaberiPiće()* ili *izaberiPićeIzFrižidera()*
- *PodAlgoritam2*: *sipajPiće()* ili *sipajPićeIzTetrapaka()*
- *PodAlgoritam3*: *uzmiČašu()* ili *uzmiČašuIzKredenca()*
- *PodAlgoritam4*: *uzmiTanjir()* ili *uzmiTanjirIzKredenca()*
- *PodAlgoritam5*: *uzmiPribor()* ili *uzmiViljuškuINož()* ili *getUtensilsFromDrawer()*

Primetimo da algoritam postaje znatno čitkiji i jasniji:

```
AlgoritamX: UgasiŽeđ
    ustani sa kauča
    idi do kuhinje
    izaberiPiće()
    popij
```

i

```
AlgoritamY: PostaviStoZa4Osobe
    idi do kuhinje
    repeat 4 times{
        uzmiČašu()
        postavi čašu na sto
        uzmiTanjir()
        postavi tanjir na sto
        uzmiPribor()
        postavi nož i viljušku na sto
```

```
    }
    vrati se na kauč
```

Primetimo u *AlgoritmuY* da uzimamo čašu, tanjir i pribor u linijama 3,4 i 7, redom. Zatim u linijama 4,6, i 8 postavljamo ove stvari na sto. Takođe, u liniji 3 *AlgoritmaX*, nakon što izvršimo funkciju *izaberiPiće()*, mi zatim i popijemo to piće u liniji 4.

Dakle, svaki od ovih podalgoritama uzima a zatim se i vraća (engl. return) sa nekim tipom objekta (tj, piće, čaša, tanjir ili pribor). Kada podalgoritam kao rezultat vraća neki tip objekta ili vrednost, takav podalgoritam nazivamo *funkcija*.

PRIMER KORIŠĆENJA FUNKCIJA I PROCEDURA

Ukoliko pod-algoritam ne vraća pozivaocu nikavku vrednost onda se takav pod-algoritam naziva procedura.

Ukoliko pod-algoritam ne vraća pozivaocu nikavku vrednost onda se takav pod-algoritam naziva procedura.

Primer porcedure bi bio algoritam koji vrši kretanje osobe do kuhinje (eventualno da ga nazovemo *idiDoKuhinje()*) i koji bi mogao biti korišćen u liniji 2 *AlgoritmaX* ili liniji 1 *AlgoritmaY*. Jasno je da takav podalgoritam nije funkcija jer iako on izvršava neke aktivnosti, on ne vraća pozivaocu objekat niti neku konkretnu vrednost. On jednostavno dovodi osobu do kuhinje (ili robota) i ne postoji vrednost koju bi on kao informaciju dao pozivaocu.

Kada rešavamo probleme, često je neophodno da problem izdelimo na korake kojima možemo lako da upravljamo, kao što je prikazano u prethodnim primerima. Iste stvari radimo i u realnom životu kako bi što lakše rešili neke svoje životne probleme.

Razmotrimo na primer slučaj rešavanja slagalice (Slika-1). Ukoliko slagalica ima puno delića (npr, 1000), bio bi za nas veoma veliki zadatak da rešimo tako nešto. Neko čak ne bi ni pristao na takav zadatak, dok bi drugi pokušali da zadatak podele u podzadatke, koji su prostiji i lakši za rešavanje.

Uobičajena osoba bi pokušala da slagalicu reši na sledeći način:

```
Algoritam: RešiSlagalicu
    prospi sve delove po stolu
    obrni sve delove tako da se vidi strana sa slikom
    razdvoj ivične delove od unutrašnjih
    reši ivične delove
    repeat until nema više lako prepoznatljivih delova{
        izaberi delove koji formiraju jedan lako prepoznatljiv deo slike
        spoji te izabrane delove
        pokušaj da uklopiš dobijeni deo slike uz ivice ili uz ostale delove slike
    }
    repeat until nema neuklopljenih delova slagalice{
        uzmi deo i pokušaj da ga uklopiš negde
        ako nije jasno gde može da se uklopi, vrati deo na gomilu
    }
```

Primetimo kako je naizgled težak problem podeljen na dobro definisane etape (ili korake). Rešavanjem prvo oboda slagalice, osoba dobija bolju predstavu o veličini cele slike što mu daje bolji osećaj o tome kako će se ostali delovi slagalice uklapaju međusobno.

Slika 3.2.1 Problem slagalice

PRIMER SLAGALICE

Naizgled težak problem može biti podeljen na dobro definisane etape (ili korake). To će sigurno dovesti do lakšeg rešavanja problema

Koraci od 5 do 8, iz prethodnog primera, mogu biti pretvoreni u specifičnije korake ako imamo više informacija o slagalici. Na primer, razmotrimo kako bi rešili Sliku-2:

Slika 3.2.2 Primer slagalice sa više podelemenata

Možda bi rešenje bilo upravo sledeće:

```
Algoritam: KonkretnaSlika
    prospi sve delove po stolu
    obrni sve delove tako da se vidi strana sa slikom
    razdvoj ivične delove od unutrašnjih
    reši ivične delove
    reši štence
    reši znak
    reši deo sa travom
    reši ogradu
```

```
    reši seno
    reši štalu
```

Primetimo kako naš algoritam postaje specifičniji jer imamo sliku onogo što pokušavamo da napravimo. Naravno, redosled u kome ćemo rešiti odgovarajuće delove slike je potpuno nebitan, ali da li li vidite kako podela problema na prostrije, manje procedure rešavanje čini znatno lakšim?

Programiranje na računaru treba biti urađeno na isti način. Uvek podelite vaš problem na manje delove koje znate da rešite. Nastavite podelu na prostije celine sve dok ne dobijete problem koga znate da rešite.

Ova strategija razbijanja problema na manje delove je često poznata kao strategija pod nazivom podeli i osvoji (divide and conquer) i predstavlja osnovni pricip za rešavanje problema.

Ova strategija nije uopste teška. kao što je spomenuto,mi je svakodnevno primenjujemo u realnom životu. Čak i deca znaju da je koriste.

PRIMER: EFIKASNOST ALGORITAMA

Glavna karakteristika algoritma je da bude što brži i da radi posao što efikasnije moguće

Razmotrimo ponovo algoritam za postavljanje stola za 4 osobe.

```
AlgoritamY: PostaviStoZa4Osobe
    idi do kuhinje
    repeat 4 times{
        uzmiČašu()
        postavi čašu na sto
        uzmiTanjir()
        postavi tanjir na sto
        uzmiPribor()
        postavi nož i viljušku na sto
    }
    vrati se na kauč
```

Algoritam koristi funkciju *uzmiČašu()*:

```
UzmiČašu()
    idi do kredenca
    otvori kredenac
    uzmi čašu
    zatvori kredenac
```

Šta je problem kod ovog algoritma? Svaki put se uzima po jedna čaša iz kredenca. Gubi se vreme na odlazak i otvaranje kredenca 4 puta. Moguće je unaprediti algoritam tako da se odlazak do kredenca vrši samo jednom. Umesto uzimanja jedne čaše, kada se dođe do kredenca uzimaćemo sve 4 čaše.

Koristi se funkcija koja uzima 4 čaše odjednom:

```
UzmiČaše()
    idi do kredenca
    otvori kredenac
    repeat 4 times{
        uzmi čašu
    }
    zatvori kredenac
```

Slično treba i izmeniti funkcije *uzmiTanjir()* i *uzmiPribor()* tako da se odjednom uzima za sve 4 osobe. Sada imamo novi kompletan algoritam za postavljanje stola za 4 osobe.

```
AlgoritamY2: EfikasnoPostavljanjeStolaZa4Osobe
    idi do kuhinje
    uzmiČaše()
    postavi čaše na sto
    uzmiTanjire()
    postavi tanjire na sto
    uzmiPribor()
    postavi noževe i viljuške na sto
    vrati se na kauč
```

Kada uporedimo ova dva algoritma, u prvom se za svaku osobu ide 3 puta do kredenca, prvi put se uzima čaša, zatim tanjir i na kraju pribor. Kako imamo 4 osobe, to je ukupno 4x3 tj. **12 puta**. Kod drugog algoritma se ide samo **3 puta** do kredenca, jednom se uzmu sve čaše, jednom svi tanjiri i na kraju svi pribori. Na osnovu toga se kaže da je drugi algoritam efikasniji od prvog.

UPOTREBA PARAMETARA FUNKCIJE

Primer upotrebe parametara u algoritmima

U prethodnom primeru se razmatra postavljanje stola za 4 osobe. Kako napisati algoritam za postavljanje stola za 8 osoba? A kako za 6 ili za 10? Da li je potrebnu uvek pisati novi algoritam?

Možemo da koristimo isti algoritam, ali ako dodamo neke nove informacije napisanim funkcijama. Funkciji *uzmiČaše()* možemo da prosledimo broj čaša koji treba da se uzme. Taj broj se naziva parametar funkcije.

```
UzmiČaše(n)
    idi do kredenca
    otvori kredenac
    repeat n times{
        uzmi čašu
    }
    zatvori kredenac
```

Zatim kada pišemo algoritam, tamo gde koristimo funkciju, navodimo **konkretan broj** koji nam treba. Ako postavljamo sto za 8 osoba, algoritam je sledeći:

```
AlgoritamY3: EfikasnoPostavljanjeStolaZa8Osoba
    idi do kuhinje
    uzmiČaše(8)
    postavi čaše na sto
    uzmiTanjire(8)
    postavi tanjire na sto
    uzmiPribor(8)
    postavi noževe i viljuške na sto
    vrati se na kauč
```

Sada možemo da napišemo dve verzije algoritma za postavljanje stola za određeni broj osoba. Taj broj može da bude bilo koji ceo broj i tada se parametar obično piše kao slovo **n.**

```
Algoritam A:
    idi do kuhinje
    repeat n times{
        uzmiČašu()
        postavi čašu na sto
        uzmiTanjir()
        postavi tanjir na sto
        uzmiPribor()
        postavi nož i viljušku na sto
    }
    vrati se na kauč
Algoritam B:
    idi do kuhinje
    uzmiČaše(n)
    postavi čaše na sto
    uzmiTanjire(n)
    postavi tanjire na sto
    uzmiPribor(n)
    postavi noževe i viljuške na sto
    vrati se na kauč
```

Ako ponovo razmotrimo razlike ovih algoritama, zaključujemo da kod prvog moramo **3*n** puta da odemo do kredenca, dok drugog moramo samo **3** puta. Primećujemo da broj odlazaka do kredenca kod drugog algoritma **ne zavisi od broja n.** Pogledajmo na primeru za prvi algoritam: za 4 osobe to je 12 puta, za 8 osoba to je 24, za 10 osoba to je 30 puta, itd. Kod drugog algoritma broj odlazaka do kredenca je uvek 3. Zaključujemo da je drugi algoritam efikasniji od prvog.

PRIMERI UPOTREBE FUNKCIJA I PROCEDURA

Navodimo primere upotrebe funkcija i procedura

Primer 1. Procedura koja prikazuje deset zvezdica na ekranu:

```
displayTenStars()
    FOR numStars = 1 to 10 DO
        Display "*"
    ENDFOR
    Display ""
EXIT
```

Primer 2. Procedura koja računa i prikazuje srednju vrednost

```
displayAverage(count, total)
    IF count > 0 THEN
        average = total / count
        Display average
    ELSE
        Display "Error, nothing to average"
    ENDIF
EXIT
```

Primer 3. Funkcija za učitavanje rednog broja meseca u godini

```
inputMonth()
    Input monthNumber
    WHILE monthNumber < 1 OR monthNumber > 12 DO
        Display "Error invalid month number"
        Input monthNumber
    ENDWHILE
RETURN with monthNumber
```

Primer 4. Funkcija koja proverava da li je broj paran Kao rezultat vraća da li je to tačno ili ne.

```
oddNumber(num)
    oddNumber = false
    IF num modulus 2 = 0 THEN
        oddNumber = true
    ENDIF
RETURN with oddNumber
```

Primer 5. Funkcija koja računa kub broja i vraća rezultat pozivaocu funkcije

```
cube(number)
    cube = number * number * number
RETURN with cube
```

Primer 6. Funkcija koja određuje maksimum dva broja i rezultat vraća pozivaocu.

```
maximumNumber (number1, number2)
    IF number1 >= number2 THEN
        maximumNumber = number1
    ELSE
        maximumNumber = number2
    ENDIF
RETURN with maximumNumber
```

ZADACI ZA INDIVIDUALNU VEŽBU

Na osnovu svega obrađenog uraditi samostalno sledeće zadatke

1. Zadatak: Napisati pod-algoritam koji prevodi funte u kilograme (1 funta = 400 g). Zatim iskorititi podalgoritam i prevesti 145 funti.

2. Zadatak: Napisati podalgoritam koji pretvara vreme zapisano u formatu (min, sek) u sate. Napisati podalgoritam koji pretvara km u milje, ako se zna da je 1 milja 1,6 kilometara. Ukoliko atletičar za 45 minuta i 30 sekundi predje 14 kilometara, napisati algoritam za program koji računa njegovu brzinu u jedinicama milja/sat.

3. Zadatak: Pretpostaviti da atletičar trči 24 milja za vreme od 1 sata, 40 minuta i 35 sekundi. Napisati pod-algoritam za program koji računa prosečnu brzinu u kilometrima po satu. (1 milja = 1.6 km)

4. Zadatak: Napisati pod-algoritam koji dimenzije papira zadat u centimetrima prevodi u inče. (1 inč = 2.54cm). Zatim iskoristiti pod-algoritam i prevesti A4 format iz centimetara u inče (A4 ima dimenzije 21.0cm x 29.7cm)

Poglavlje 4

Tehnike rešavanja problema

POSTAVLJANJE PITANJA

Postoje neka uobičajena pitanja koja možete da postavite kada je u pitanju proces programiranja

Vi rešavate probleme svaki dan, često bez svesti o procesu kroz koji prolazite. Na časovima nastave, iz većine predmeta, obično dobijete sve informacije koje su vam neophodne da bi rešili problem: jasna postavka problema, neophodan ulaz, i traženi izlaz. Za razliku od časa u školi, problemi koje rešavate u realnom životu nisu baš tako jednostavni. Veoma često je neophodno da sami definišete problem a zatim da odlučite sa kojim tipom informacije treba da radite i kakav rezultat treba da dobijete.

Nakon što ste razumeli i dobro izanalizirali problem, morate da osmislite rešenje za njega, tj — algoritam. U prethodnom delu teksta smo definisali algoritam kao proceduru korak-po-korak za rešavanje problema u konačnom vremenu korišćenjem konačnog skupa podataka.

Mi često pratimo date smernice, igramo igru, spajamo lego kockice, ili uzimamo lek. Kada je u pitanju faza rešavanja problema u procesu programiranja, ideja je da se kreira algoritam a ne da se prate njegove instrukcije. To znači da morate da budete svesni strategije koju koristite da bi ste rešili problem jer jedino tako možete da je primenite na programerski problem.

Kada vam usmeno postave zadatak, obično postavljate pitanja—Kada? Zašto? Gde?—dok u potpunosti ne budete sigurni da razumete šta tačno treba da uradite. Ukoliko su vaše instrukcije zapisane, možete pitanja da dopišete na marginama stranice, podvučete najbitnije reči ili rečenice, ili da na bilo koji drugi način naznačite da deo postavke zadatka nije potpuno jasan. Moguće je da se odgovori na vaša pitanja upravo nalaze u prvom sledećem paragrafu (obično je to slučaj) ili možda morate dodatno da prodiskutujete sa osobom koja vam je postavila zadatka i dodatno razjasnite stvari.

Postoje neka uobičajena pitanja koja možete da postavite kada je u pitanju proces programiranja:

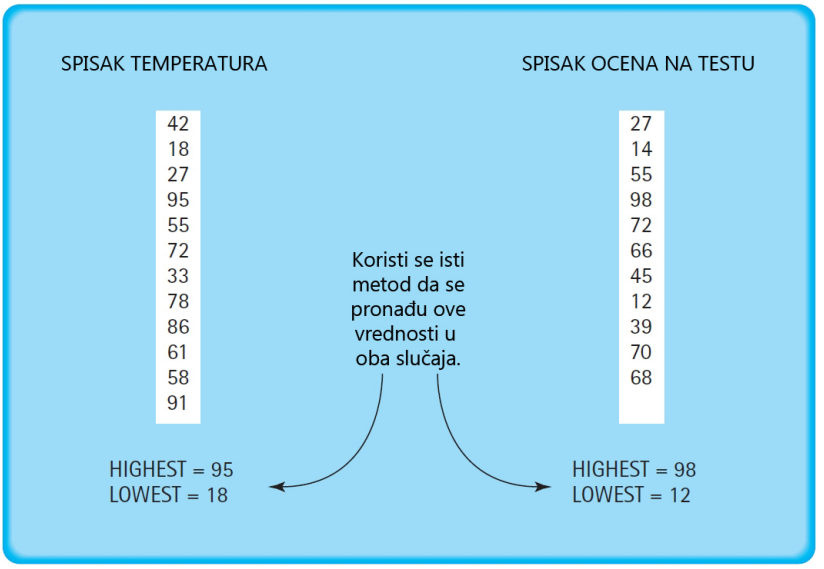
- Sa čim treba da radim—tj, koji su moji podaci?
- Kako ti podaci izgledaju (kakvog su oblika)?
- Koje operacija treba da izvršim nad podacima?
- Koliko različitih podataka mi je na raspolaganju?
- Kako da znam da sam završio sa obradom svih podataka, da nije neki podatak promakao?
- Kako izlaz mog programa treba da izgleda?
- Da li postoje neki specijalni uslovi koji mogu da dovedu do greške?

ISKORISTITE STVARI KOJE SU VAM VEĆ POZNATE

Ako ste rešavali isti ili sličan problem ranije, onda iskoristite već definisano rešenje

Nikad ne pokušavajte da otkrijete toplu vodu, ili osmislite točak – nešto što već postoji. Ako rešenje postoji onda ga iskoristite. Ako ste rešavali isti ili sličan problem ranije, onda iskoristite već definisano rešenje. Ljudi su veoma talentovani kada je u pitanju prepoznavanje sličnih stvari i situacija. Kao što je poznato, mi ne moramo da učimo kako da odemo do radnje da kupimo mleko, zatim jaja, i na kraju neku čokoladicu. Na osnovu iskustva, znamo da se svaki odlazak do radnje svodi na isto, samo kupujemo različite stvari.

U programiranju, odgovarajući problemi se javljaju iznova i iznova samo u drugačijijm oblicima. Dobar programer će uvek moći da prepozna podzadatak koji je rešavao, ili rešavala, ranije i to će uključiti u rešenje. Na primer, određivanje najviše i najniže dnevne temperature je gotovo identičan proces određivanju najviše i najniže ocene na testu. U oba slučaja nas zanimaju najveće i najmanje vrednosti u nekom skupu brojeva, Slika-1.



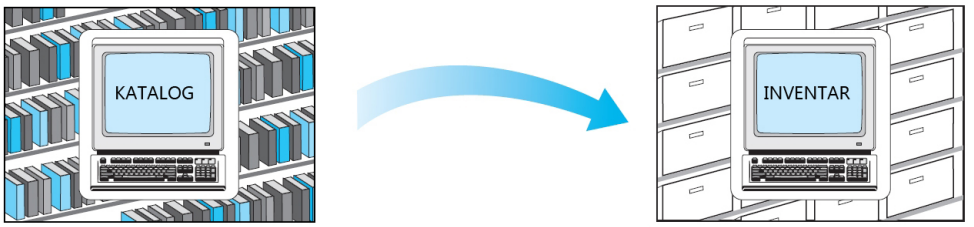
Slika 4.1 Potražiti stvari koje su vam već poznate

REŠITI PO ANALOGIJI

Rešavanje problema postaje lakše ako se sećate kako ste rešili neki drugi sličan problem

Obično problemi podsećaju na neki sa kojim ste se već suočili rainije. Možete uvideti da je rešavanje problema lakše ako se sećate kako ste rešili neki drugi problem. Drugim rečima, napravite paralelu između ta dva problema. Kako radite na novom problemu možete da naiđete na stvari koje su različite u odnosu na stari problm, ali su to uglavnom samo sitni detalji koje možete veoma lako da savladate.

Kada pokušavate da izaberete algoritam za rešavanje problema, nemojte se ograničiti na kompjuterski-orijentisana rešenja. Obavezno se vratite korak unazad i pokušajte da uhvatite širu sliku problema. Nemojte da se uplašite ako vaš problem nije u idealnoj analogiji sa nekim drugim problemom— glavni razlog startovanja od analogije je da otkrijete mesto odakle treba da započnete projektovanje vašeg algoritma (pogledati Sliku-2). Imajte uvek na umu da su najbolji programeri oni ljudi koji imaju ogromno iskustvo u rešavanju bilo koje vrste problema.



Kataloški sistem biblioteke može dati uvid u to kako treba da se organizuju delovi inventara.

Slika 4.2 Analogija sa postojećim rešenjem.

CILJ OPRAVDAVA SREDSTVA (MEANS-ENDS)

Uopštena strategija ove analize je da se definišu medjutačke na putu, i da se onda analiziraju sredstva za dolazak iz jedne tačke u drugu

Obično su početno i krajnje stanje rešenja poznate stvari; problem je definisati skup akcija koje trebaju biti korišćene da bi se stiglo iz jednog stanja u drugo.

Pretpostavimo da želimo da putujemo od Beograda do Pariza. Poznato je naše početno stanje (u Beogradu smo) i konačno stanje (Pariz). Problem je ustvari kako doći od jednog mesta do drugog. U ovom primeru imamo puno izbora. Možemo da letimo, pešačimo, stopiramo, itd. Metod koji ćemo da izaberemo zavisi od raznih okolnosti. Ukoliko smo u žurbi verovatno ćemo leteti. Jednom kad smo usko definisali skup akcija, možemo da počnemo da radimo na detaljima. Često može da pomogne ako definišemo međustanja do kojih ćemo lakše stići nego do konačnog odredišta. Pretpostavimo da imamo kao opciju jedan veoma jeftin let od Beograda do Ženeve. Tamo se možda odlučimo da put podelimo na dva dela: Od Beograda do Ženeve, i od Ženeve do Pariza. Naš međucilj je da stignemo od Beograda do Ženeve (Slika-3).

Start: Beograd **Cilj**: Pariz	**Sredstva:** let, pešačenje, auto stop, bicikl, automobil, autobus
Start: Beograd **Cilj**: Pariz	**Razmotrena sredstva**: let do Minhena a onda do Pariza; let do Frankfurta a onda do Pariza; let do Ženeve a onda do Pariza
Start: Beograd **Posredni cilj**: Ženeva **Cilj**: Pariz	**Sredstva do posrednog cilja**: jeftini let, pešačenje, auto stop, bicikl, automobil, autobus
Rešenje: Uzeti jeftini let do Ženeve, a zatim uhvatiti let iz Ženeve do Pariza	

Slika 4.3 Means-ends analysis

Uopštena strategija ove analize je da se definišu međutačke na putu, i da se onda analiziraju sredstva za dolazak iz jedne tačke u drugu. Ovaj proces se veoma lako prevodi u jedan računarski program.

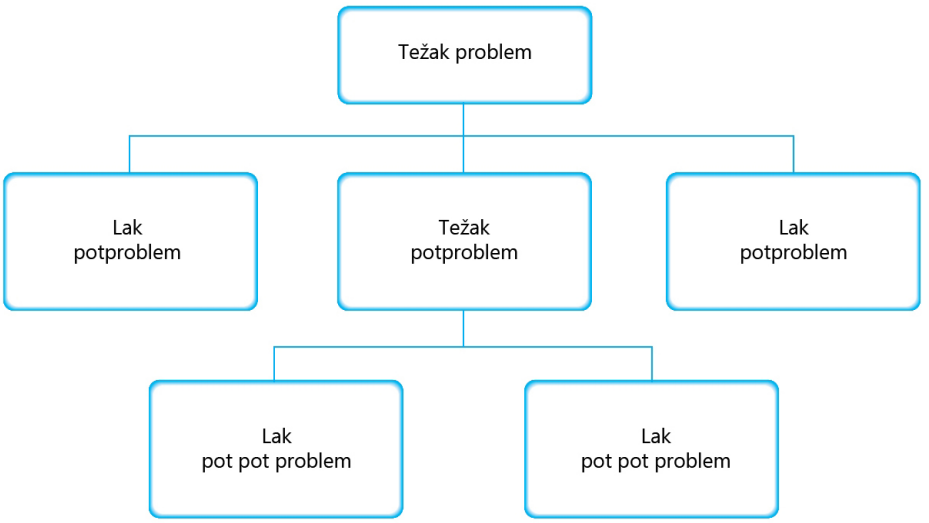
Proces programiranja započinjemo definisanjem onoga što će biti ulaz i definisanjem kako izlaz treba da izgleda. Zatim treba da razmotrimo akcije koje računar treba da izvrši i da onda konačno izaberemo sekvencu akcija koja će uspeti da ulazne podatke preobrazi u željene rezultate.

PODELI I OSVOJI

Mi uvek delimo veliki problem na manje delove koji se mogu pojedinačno rešiti

Veoma često pokušavamo da ogromne probleme podelimo na manje celine koje su lakše za rešavanje. Na primer, čišćenje cele kuće možda deluje kao pretežak zadatak; ali, ako idmo sobu po sobu (ne prelazimo na sledeću dok ne završimo trenutnu) onda zadatak postaje dosta jednostavniji. Isti princip je primenljiv i na programiranje.

Mi uvek delimo veliki problem na manje delove koji se mogu pojedinačno rešiti (Slika-4), i to će vas sigurno pratiti tokom celog vašeg programerskog poziva.



Slika 4.4 Podeli i osvoji

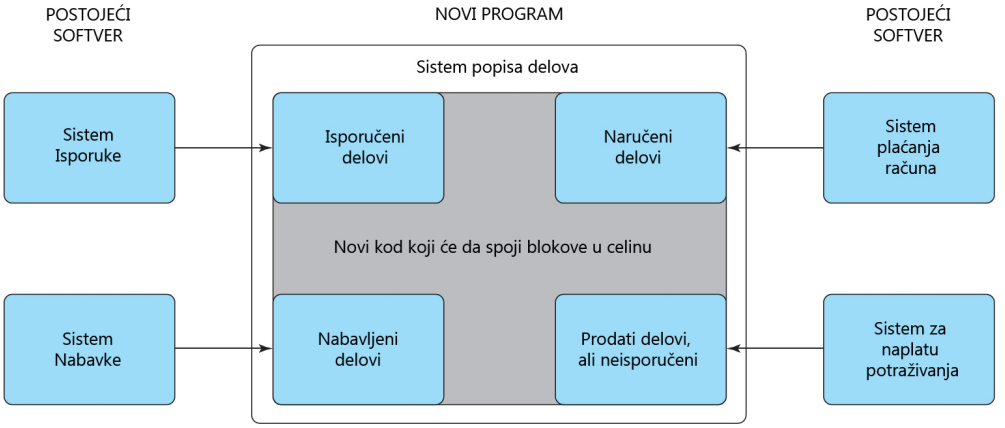
PRISTUP SKLAPANJA KOCKICA (BUILDING-BLOCK)

Analiziranjem velikog problema shvatamo da on može biti podeljen na manje probleme za koje rešenje već postoji

Jedan od načina da napadnemo veliki problem je da proverimo da li rešenje za bilo koji delić tog problema već postoji. Može biti moguće spojiti ove deliće zajedno i rešiti ceo problem. Ova strategija je kombinacija *„potrage za sličnim stvarima“* i strategije*„podeli i osvoji“*.

Vi analizirate veliki problem a zatim shvatate da on može biti podeljen na manje probleme za koje rešenje već postoji. Rešavanje velikog problema se svodi na sklapanje postojećih rešenja za njegove manje podprobleme, kao u slučaju kada slažete cigle i tako pravite zid (Slika-5).

Kao što ćete i videti u nastavku ovog kursa ova tehnika rešavanja problema je osnova metodologije koja se naziva objektno-orijentisano projektovanje (object-oriented design).



Slika 4.5 Pristup sklapanja kockica (building-block approach)

MENTALNA BLOKADA: STRAH OD SAMOG PROBLEMA

Napišite svojim rečima na papiru postavku problema tako da budete sigurni da ste ga razumeli

Piscima je dobro poznato kako izgleda početak pisanja, kada kreću od praznog papira i ne znaju odakle da počnu. Programeri se suočavaju sa sličnom preprekom kada počinju da rade na velikim problemu. Problem sa kojim se suočavaju često deluje previše težak za njih, Slika-6.

Bez obzira na sve, zapamtite da će uvek postojati mesto odakle možete da započnete rešavanje bilo kog problema: Stoga napišite svojim rečima na papiru postavku problema tako da budete sigurni da ste ga razumeli. Nakon što ste svojim rečima „prepričali“ postavku vašeg problema, tj parafrazirali problem, moći ćete lakše da se fokusirate na svaki od poddelova problema umesto da pokušavate da savladate ceo problem odjednom. Ovaj proces vam daje jasniju sliku o ukupnom problemu. On vam pomaže da uočite delove problema koji deluju slično, ili su analogni, sa problemom koga ste već rešavali. Takođe vam omogućava da izolujete zonu problema koja je nejasna i za koju vam trebaju dodatne informacije.

Kako zapisujete problem na papiru, težnja je da grupišete zajedno stvari u manje i razumljivije komade podataka i informacija, jer je takav način pogodan za preimenu strategije podeli i osvoji (divide and conquer).

Slika 4.6 Mentalna blokada.

Najveći broj slučajeva mentalnih blokada je rezultat nerazumevanja problema. Pokušavanje preformulisanja problema svojim rečima je dobar način da se usredsredite na poddelove problema, jedan u datom trenutku, i da razumete šta je sve neophodno da bi problem rešili.

UOBIČAJENA PROGRAMERSKA PRAKSA

Kada god bi rešanje pojedinačnih podređenja dupliralo broj koraka, razmislite o tome da spojite rešavanje umesto da rešavate potprobleme jedan za drugim

Spajanje rešenja

Jedan od načina da se iskombinuju postojeća rešenja je da se ona spajaju redom, počevši od odgovarajuće osnove. Na primer, da bi odredili srednju vrednost liste brojava, moramo prvo da sumiramo brojeve i da izbrojimo koliko ih ima.

Ako već imamo izračunatu sumu, i ako imamo već sračunat njihov broj, onda možemo da kombinujemo ova rešenja. Ali, ako prvo izvršimo sumiranje, a zatim brojanje morali bi da pročitamo listu dva puta.

Stoga, možemo da smanjimo broj koraka (uštedimo na vremenu) ako spojimo oba rešenja: pročitaj vrednost, zatim ga dodaj na sumu (sum), a zatim uvećaj brojač (count) za jedan pre nego što pročitaš sledeću vrednost.

Zato, kada god bi rešanje pojedinačnih podređenja dupliralo broj koraka, razmislite o tome da spojite rešavanje umesto da rešavate potprobleme jedan za drugim.

Algoritamsko rešavanje problema

Osmišljavanje algoritma za rešavanje problema nije tako jednostavan posao ("prosto kao pasulj"). Naime, to je uglavnom proces koji se bazira na metodu *probe i greške* (trial and error) koji zahteva nekoliko pokušaja i nekoliko podešavanja. U realnosti, uglavnom testiramo program nakon svakog pokušaja da proverimo da li je to rešilo problem. Ako jeste onda super. Ako nije onda pokušaj ponovo. Mi obično koristimo kombinaciju tehnika koje smo do sad opisali da bi rešili netrivijalne probleme.

Zapamtite da računar može da odradi samo određene stvari. Stoga je vaša najveća briga kako da od računara obezbedite da on transformiše, manipuliše, sračunava ili procesira ulazne podatke tako da se proizvede željeni izlaz. Ako dobro znate koje su dozvoljene instrukcije i tipovi podataka u programskom jeziku koga koristite, onda sigurno nećete projektovati algoritam koji je težak za kodiranje, ili koga je nemoguće iskodirati.

KORISNI TRIKOVI ZA REŠAVANJE BILO KOG PROGRAMERSKOG PROBLEMA

Svaki programer, bilo početnik ili ekspert, pred sobom ima problem koji pokušava efikasno da reši. I jedan i drugi se suočavaju sa istim izazovima na tom putu

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Zakljucak

ZAKLJUČAK

Na osnovu svega obrađenog možemo zaključiti sledeće:

Algoritam je opis za rešavanje nekog problema. To je konačna i precizno definisana procedura (niz dobro definisanih pravila) kojom se ulazne vrednosti transformišu u izlazne.

Osnovne osobine algoritama su diskretnost, rezultativnost, determinisanost i masovnost. Algoritmi se obično predstavljaju na dva načina: grafički i pomoću pseudokoda.

Rešavanje problema je sekvencijalni (korak po korak) proces analiziranja informacija koje se odnose na datu situaciju i generisanje odgovarajućih odziva.

Ideja korišćenja najjasnijeg algoritma koji nema u sebi puno nepotrebnih detalja je poznata kao apstrakcija. Apstrakcija nam omoguća da se fokusiramo na samo par pojmova u datom trenutku, i da tako shvatimo širu sliku kada je u pitanju rešenje problema.

Uvek je moguce kreirati visoko apstraktan algoritam, sakrivajuci detalje u podalgoritme. Stvari možemo na ovaj način načiniti još apstraktnijim koliko god puta osetimo da je takav postupak neophodan.

Uvek postoji mogućnost da deo vašeg algoritma bude korišćen od strane nekog drugog algoritma, naročito u situacijama kad je neke korake neophodno ponoviti više puta. Podalgoritmi se u oblasti računarskih nauka uglavnom nazivaju moduli, funkcije ili procedure.

Ukoliko pod-algoritam ne vraća pozivaocu nikavku vrednost onda se takav pod-algoritam naziva procedura. Naizgled težak problem može biti podeljen na dobro definisane etape (ili korake). To će sigurno dovesti do lakšeg rešavanja problema