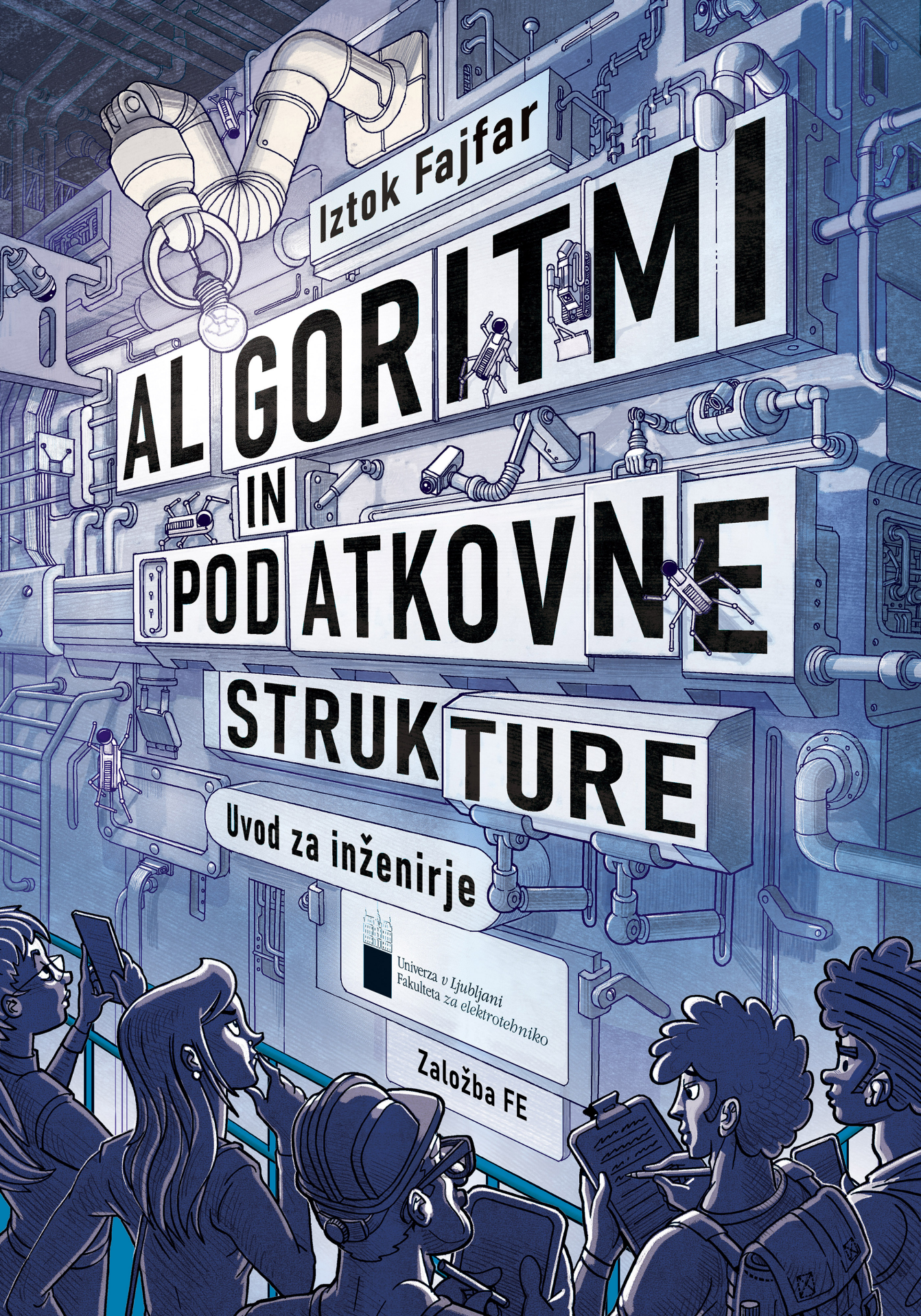

Iztok Fajfar
ALGORITMI
IN
PODATKOVNE
STRUKTURE
Uvod za inženirje
Univerza v Ljubljani
Fakulteta za elektrotehniko
Založba FE

ALGORITMI IN PODATKOVNE STRUKTURE

PRAZNA STRAN

ALGORITMI IN PODATKOVNE STRUKTURE

Uvod za inženirje

Iztok Fajfar

Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

Recenzija: doc. dr. Janez Puhan, doc. dr. Boštjan Slivnik
Jezikovni pregled: Jana Kolarič
Ilustracija in oblikovanje naslovnice: Ciril Horjak, Omar Horjak
Založnik: Založba Fakultete za elektrotehniko, Ljubljana
Izdajatelj: Univerza v Ljubljani, Fakulteta za elektrotehniko, Ljubljana
Urednik: prof. dr. Sašo Tomažič

1. elektronska izdaja
Način dostopa (url): http://fajfar.eu/Algoritmi-in-podatkovne-strukture.pdf

CIP – Kataložni zapis o publikaciji
Narodna in Univerzitetna knjižnica, Ljubljana

COBISS.SI-ID=304902144
ISBN 978-961-243-400-7 (pdf)

Življenju,
vesolju
in sploh vsemu.

PRAZNA STRAN

VSEBINA

PRAZNA STRAN

ZAHVALA

Da kaj nastane, so potrebni ljudje. Ogromno ljudi. Hvala vsem domačim in prijateljem, ki me podpirate pri vsem, kar počnem. Hvala sodelavcem – skupaj smo odlična ekipa in počnemo izjemne stvari. Hvala za vsa vaša konstruktivna nasprotovanja. Hvala brezštevilnim avtorjem knjig, blogov in prispevkov na strokovnih forumih. K vam so me občasno gnali dvomi o tem in onem. Hvala množici neumornih študentov, ki ste ure in ure presedeli v veliki predavalnici Fakultete za elektrotehniko in vaša radovednost niti med odmori ni počivala. Vaša vprašanja in pripombe, nezaupljivi pogledi med predavanji ter dobronamerno kritična elektronska sporočila so gnetli in oblikovali vsebino učbenika, ki ga pravkar berete.

PRAZNA STRAN

NAMESTO UVODA

Ko sem se lotil pisanja tega učbenika, sem želel študentom elektrotehnike (in ostalih tehniških ved) na pregleden način pokazati, kako reševati probleme na algoritemski način in kako pri tem izbrati čim primernejšo zgradbo podatkov. Z drugimi besedami – kako se lotiti programiranja računalnikov. Pri pisanju sem imel pred seboj predvsem dva pomembna cilja: prvič, učbenik mora biti primeren tudi in predvsem za študente, ki se v življenju še nikoli niso srečali z računalniškim programiranjem, in drugič, primeren mora biti tudi kot uvod v kasnejše učenje jezika C. To je še vedno najpomembnejši jezik, ki se ga študentje učijo na Fakulteti za elektrotehniko v Ljubljani, kjer je ta učbenik nastal. Da bi dosegel zastavljena cilja, sem izbral jezik JavaScript. Ta jezik bomo uporabljali kot psevdo jezik – vzeli bomo le njegove drobce, ravno toliko, da se bo dalo algoritme preizkusiti na računalniku. Ne bomo se na primer ukvarjali s podatkovnimi tipi, uporabili bomo zgolj številke. Prezrli bomo večino nepotrebnih elementov jezika, kot sta na primer stavka `switch` in `do...while` ali množica prikladnih objektov, katerih implementacija in uporaba je vezana na konkreten programski jezik. Izognili se bomo tudi znakovnim nizom, ki so zgolj poseben primer nizov števil. Način kodiranja znakovnih nizov ter priročne operacije in funkcije, ki jih konkretni programski jeziki ponujajo za njihovo obdelavo, so zgolj tehnološke posebnosti posameznih jezikov. Zato nas v tem učbeniku ne bodo zanimale.

PRAZNA STRAN

1. POGLAVJE

ALGORITEM IN PROGRAM

Algoritem je navodilo za reševanje določenega problema. Navadno je algoritem zapisan kot seznam korakov, navodil ali ***ukazov***, ki pripeljejo do rešitve problema. Algoritem zapisujemo na različne načine, odvisno od tega, kdo ga bo izvajal. Če algoritem zapisujemo na način, razumljiv človeku, in pri tem ne uporabljamo pravega programskega jezika, potem pravimo, da je algoritem zapisan v tako imenovanem ***psevdo jeziku***. Če pa bo algoritem izvajal računalnik, potem ga moramo zapisati v pravem programskem jeziku. Tako zapisanemu algoritmu pravimo ***program***.

Poglejmo si preprost primer, kjer želimo izračunati potrebno dolžino vzletne steze za potniško letalo. Vzemimo, da imamo podano vzletno hitrost letala $v = 90\ \mathrm{m/s}$, vemo pa tudi, da letalo to hitrost doseže s konstantnim pospeševanjem iz mirovne lege v času $t = 30\ \mathrm{s}$. Ker imamo opravka s konstantnim pospeškom, lahko izračunamo potrebno dolžino vzletne steze po preprosti enačbi:

$$s = \frac{1}{2}\, v\, t \tag{1.1}$$

Z nekaj osnovnega znanja fizike smo nalogo hitro rešili, vendar si bomo zdaj problem računanja dolžine vzletne steze pogledali še z drugačnimi očmi. Naš cilj je, da rešitev problema najprej zapišemo v obliki algoritma, potem pa še kot program v konkretnem jeziku JavaScript. Preden pa se naloge lotimo, se moramo še dogovoriti, v kakšni obliki bomo probleme zapisovali.

Izhajali bomo iz dejstva, da računalniški programi praktično vedno na tak ali drugačen način obdelujejo določene podatke. Zato bomo pri podajanju problemov vedno določili,

kakšni so vhodni podatki (vhod) in kaj pričakujemo kot rezultat (izhod). Problem računanja dolžine vzletne steze lahko zapišemo takole:

Vhod: vzletna hitrost (v), čas pospeševanja (t);
Zahtevani izhod: potrebna dolžina vzletne steze (s);

Zapišimo algoritem, ki reši gornji problem, najprej v psevdo jeziku:

```
Vhod:  v, t;
Izhod:  s;
s ← 1/2 v t;
Sporoči:  s;
```

Puščica v tretji vrstici algoritma predstavlja smer prirejanja vrednosti: spremenljivko na levi nastavimo na vrednost, ki jo izračunamo na desni. Ko se gornji algoritem izvede, hrani ***izhodna*** spremenljivka z imenom s potrebno dolžino vzletne steze, izračunano glede na podatke, ki se nahajajo v ***vhodnih*** spremenljivkah z imenoma v in t.

Program, zapisan v jeziku JavaScript, se ne bo prav dosti razlikoval od gornjega algoritma v psevdo jeziku. Da bomo ob zagonu dobili konkretno rešitev (t.j. dolžino vzletne steze), moramo na začetku vhodnim spremenljivkam prirediti konkretne številske vrednosti (brez enot), na koncu pa moramo poskrbeti, da se bo izračunana vrednost nekje tudi izpisala, da jo bomo lahko videli. Program je videti takole:

```
v = 90;              //nastavi v na 90
t = 30;              //nastavi t na 30
s = 1 / 2 * v * t;   //izračunaj s
console.log(s);      //izpiši s (njegovo vrednost) v konzolo
```

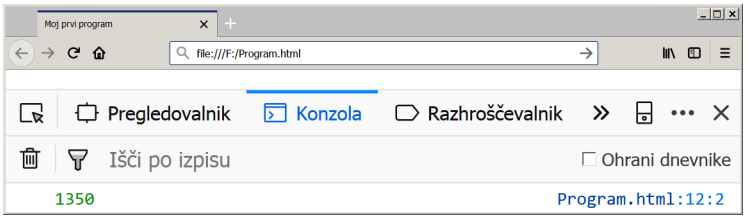
Najprej na desni strani opazimo nekaj vrstic besedila, ki se začenjajo z dvema poševnicama (//). Dve poševnici v jeziku JavaScript označujeta ***opombe***: vse, kar jima sledi do konca trenutne vrstice, računalnik preprosto prezre. Opombe so namenjene človeškemu programerju, da lažje razume programsko kodo. Na levi strani se nahaja dejanski program, ki ga bo računalnik izvedel. V prvih dveh vrsticah kode nastavimo spremenljivki v in t na želene vrednosti. Tretja vrstica izračuna opravljeno pot. Vidimo, da v jeziku JavaScript poševnica (/) predstavlja operator deljenja, medtem ko zvezdica ($\star$) predstavlja množenje. Zelo pomembno je dejstvo, da priredilni operator (=) vedno nastavi spremenljivko na svoji ***levi*** na vrednost, ki se nahaja na njegovi ***desni***[1]. Zadnja vrstica kode poskrbi za to, da se vrednost spremenljivke s, ki smo jo pred tem izračunali, izpiše v konzoli. Podpičja na koncu vrstic sicer niso potrebna za pravilno delovanje programa, vendar jih bomo zaradi doslednosti vedno pisali.

Če želimo gornji program dejansko zagnati, nas čaka še nekaj dela. Ker bomo programe zaganjali v spletnem brskalniku, potrebujemo okrog kode v jeziku JavaScript še nekaj osnovne kode v jeziku HTML. Vendar brez skrbi! Ne bomo se učili še enega jezika. Kodo bomo enostavno prekopirali in prav nič nas ne bo motilo, da je ne razumemo. Če pa bo koga le premagala radovednost, si lahko podrobno razlago med drugim prebere v 1. poglavju učbenika [1], ki ga lahko dobite v fakultetni knjižnici. Takole je videti celotna koda:

[1]To smer bomo v psevdo jeziku posebej poudarjali s puščico, ki smo jo srečali v zadnjem primeru psevdo kode (t.j. $s \leftarrow 1/2\,v\,t$).

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Moj prvi program</title>
  </head>
  <body>
    <script>
      v = 90;              //nastavi v na 90
      t = 30;              //nastavi t na 30
      s = 1 / 2 * v * t;   //izračunaj s
      console.log(s);      //izpiši s v konzolo
    </script>
  </body>
</html>
```

Kodo prekopirajte ali prepišite v običajen urejevalnik besedila in jo shranite kot navadno besedilo ASCII. Datoteko poimenujte s poljubnim imenom, pomembno je le, da se konča s končnico .html. Shranjeno datoteko odprite s spletnim brskalnikom in v brskalniku pritisnite kombinacijo tipk Ctrl+Shift+I, s čimer odprete okno z razvijalskimi orodji. Potem izberite jeziček z imenom *Konzola* (angl. Console). V konzoli boste zagledali številko 1350, kar je izračunana dolžina letališke steze v metrih, ki jo izpiše zadnja vrstica našega programa. Naslednja slika prikazuje stanje v konzoli brskalnika Firefox, kjer vidimo izpisan rezultat 1350 v levem spodnjem kotu:



1.1 Naloge

Za vajo rešite še naslednje naloge:

Naloga 1.1 Kakšna vrednost se bo izpisala v konzoli, ko zaženemo naslednji program? Na vprašanje najprej odgovorite brez uporabe računalnika, potem pa pravilnost svojega odgovora preverite tako, da program zaženete v brskalniku.

```
a = 10;
b = a / 2;
a = b / 2;
b = a;
a = b;
console.log(a);
```

Naloga 1.2 Zapišite algoritem in program za naslednji problem:

Vhod: polmer kroga (r);
Zahtevani izhod: obseg in ploščina kroga s podanim polmerom;

Pomoč: Namesto številske konstantne vrednosti $3{,}14159$ (π) lahko uporabite vgrajeni izraz jezika JavaScript `Math.PI`. Če boste v programu uporabili številsko konstanto, morate paziti, da decimalno vejico zamenjate s piko (t.j. `3.14159`).

Naloga 1.3 Zapišite algoritem in program za naslednji problem:

Vhod: dve številski vrednosti x in y;
Zahtevani izhod: isti dve številski vrednosti, zamenjani med seboj (x naj ima vrednost, ki jo je imel na začetku y, in obratno);

2. POGLAVJE

KRMILNI STAVKI

Ena najpomembnejših lastnosti računalniških programov je ta, da se lahko na različne vhodne podatke odzivajo na različne načine. Takšno obnašanje lahko dosežemo z uporabo ***krmilnih stavkov***, kot sta na primer pogojni in ponavljalni stavek.

2.1 Pogojni stavek

Dolžino letališke steze lahko izračunamo tudi z drugačnimi vhodnimi podatki. Na primer, če imamo podano vzletno hitrost (v) in pospešek (a) letala, lahko najprej izračunamo čas, ki je potreben, da letalo doseže zahtevano hitrost:

$$t = \frac{v}{a}, \tag{2.1}$$

potem pa lahko dolžino steze spet izračunamo po enačbi (1.1). Združimo obe možnosti (eno, ko je podan čas, in drugo, ko je podan pospešek), kar nas pripelje do naslednjega problema:

Vhod:
vzletna hitrost (v);
bodisi čas pospeševanja (t) bodisi pospešek (a);
Zahtevani izhod: potrebna dolžina vzletne steze (s);

Preden se lotimo pisanja algoritma, se moramo dogovoriti, kako bomo označevali podatke, ki niso podani. Na najnižjem nivoju programiranja ne moremo enostavno preverjati,

ali kakšen podatek obstaja ali ne. Namesto tega dodelimo spremenljivki, katere vrednost naj ne bi obstajala, določeno neveljavno vrednost. Glede na to, da so v našem problemu vse vrednosti pozitivne, se lahko odločimo, da bo vrednost -1 pomenila, da podatek ni podan. Zdaj lahko zapišemo algoritem:

```
Vhod:  v, t, a;
Izhod:  s;
Če je t enak −1:
{
  t ← v/a;
}
s ← 1/2 v t;
Sporoči:  s;
```

Poleg tega, da vsebuje dodatno vhodno spremenljivko (a), se ta algoritem od prejšnjega razlikuje po tem, da smo mu dodali *pogojni stavek*. Slednji povzroči, da se čas (t) izračuna le takrat, ko ni podan (oz. ima vrednost -1). Zapišimo zdaj še ustrezen program v jeziku JavaScript:

```
v = 90;
t = -1;
a = 3;
if (t == -1) {   //Če čas ni podan,
  t = v / a;     //ga izračunaj iz hitrosti in pospeška.
}
s = 1 / 2 * v * t;
console.log(s);
```

Če hočete ta program zagnati v brskalniku, ne smete pozabiti dodati še potrebne kode HTML, program pa vstavite med znački `<script>` in `</script>`.

Pogojni stavek v jeziku JavaScript je stavek `if`, katerega ***sintaksa*** (t.j. pravilo zapisovanja) zahteva, da se pogoj zapiše v par okroglih oklepajev. Kakor hitro je pogoj izpolnjen, se bo izvedla koda, ki je zapisana v paru zavitih oklepajev, ki sledi. Opazimo še nenavaden operator, ki je sestavljen iz dveh enačajev (==). Za razliko od enojnega enačaja (=), ki ***nastavi*** vrednost spremenljivke, dvojni enačaj ***primerja*** dve vrednosti med seboj. Takšno razlikovanje teh dveh operatorjev je potrebno. Če bi za dve različni operaciji uporabljali isti simbol, potem bi potrebovali dodatna pravila, ki bi odločala o tem, kako se bo operator obnašal. To pa bi po nepotrebnem zapletlo zadeve.

Stavek `if` nam omogoča, da z uporabo ustreznega pogoja nadzorujemo, ali se bo določen del kode izvedel ali ne. V svoji razširjeni obliki lahko ta stavek deluje tudi kot kretnica, ki odloča o tem, ali se bo izvršil prvi ali drugi del kode. Poglejmo si primer.

Čeprav je ledišče vode pri okoli nič stopinj Celzija, nas sistem v osebnem avtomobilu opozori na nevarnost poledice že pri treh stopinjah. Vzrok za to je dejstvo, da je lahko temperatura na določenih odsekih cestišča nižja, kot na primer v senci ali na mostu. Izkušnje kažejo, da se zaradi tega pri izmerjenih treh stopinjah Celzija ponekod na cestišču že začne pojavljati poledica. Napišimo algoritem in program za rešitev naslednjega problema:

> ***Vhod:*** izmerjena temperatura (T);
> ***Zahtevani izhod:*** sporočilo o stanju na cesti (t.j. »ni posebnosti« oz. »nevarnost poledice«);

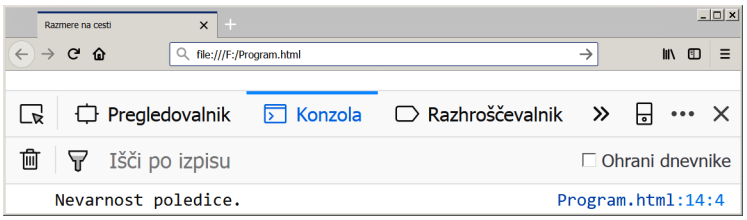
Problem rešimo z naslednjim algoritmom:

```
Vhod:  T;
Izhod:  besedilno sporočilo o stanju na cesti;
Če je T > 3:
{
  Sporoči:  "Na cesti ni posebnosti.";
}
sicer:
{
  Sporoči:  "Nevarnost poledice.";
}
```

Pogojnemu stavku smo dodali del (`sicer`), ki se izvede, kadar pogoj ***ni izpolnjen***. V jeziku JavaScript to dosežemo tako, da stavek `if` razširimo z delom `else`:

```
T = 2.5;
if (T > 3) {   //Če je temperatura večja od treh stopinj:
  console.log("Na cesti ni posebnosti.");
}
else {         //sicer:
  console.log("Nevarnost poledice.");
}
```

Funkcija `console.log` v konzolo izpiše besedilo v navednicah dobesedno tako, kakor je navedeno. Če program zaženemo v brskalniku Firefox, dobimo takšno sliko (sporočilo se izpiše v spodnjem levem vogalu konzole):



Naloga 2.1 Sami poskusite napisati algoritem in program za rešitev naslednjega problema, ki se nanaša na računanje dolžine vzletne steze:

> ***Vhod:*** vzletna hitrost (v), pospešek letala (a), dolžina vzletne steze (s);
> ***Zahtevani izhod:*** sporočilo o tem, ali je steza dovolj dolga (t.j. »Steza je prekratka.« oz. »Steza je dovolj dolga.«);

2.2 Zaporedni pogojni stavki

Nemalokrat se pripeti, da imamo več kot dve možnosti, ki jih moramo upoštevati. Takrat lahko enega za drugim uporabimo več pogojnih stavkov. Poglejmo si primer, kjer želimo zapisati število burgerjev s številko in besedo *burger* v pravilni obliki. Opraviti imamo s štirimi različnimi možnostmi. Namreč, poleg običajnih ednine, dvojine in množine se v slovenščini za količinskimi pridevniki, kot so pet ali mnogo, v imenovalniku uporablja rodilniška oblika (npr. pet burgerjev).

Problem se glasi takole:

Vhod: število burgerjev (n);

Zahtevani izhod: n in beseda *burger* v pravilni obliki (v imenovalniku);

Naslednji algoritem predstavlja možno rešitev problema:

```
Vhod:  n;
Izhod:  n in beseda "burger" v pravilni obliki;
Sporoči:  n;
n ← n%100;          //Opomba: ostanek pri deljenju s 100
Če je n enak 1:
{
  Sporoči:  "burger";
}
sicer, če je n enak 2:
{
  Sporoči:  "burgerja";
}
sicer, če je n enak 3 ali 4:
{
  Sporoči:  "burgerji";
}
sicer:
{
  Sporoči:  "burgerjev";
}
```

Ker se omenjene štiri besedne oblike ponovijo na vsakih 100 (npr. 2, 102, 202, itd. burgerja), moramo najprej izračunati ostanek pri celoštevilskem deljenju s 100. Najpogosteje uporabljen znak za operator, ki vrne ostanek pri celoštevilskem deljenju, je znak za odstotek (%).

Zapišimo še program:

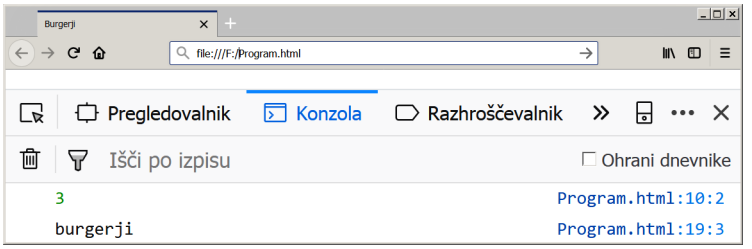
```
n = 3;
console.log(n);
n = n % 100;   //ostanek pri deljenju s 100
if (n == 1) {
  console.log("burger");
}
else if (n == 2) {
  console.log("burgerja");
}
else if (n == 3 || n == 4) {
  console.log("burgerji");
}
else {
  console.log("burgerjev");
}
```

Dve pokončni črti v tretjem stavku `if` predstavljata logični operator ALI. Logičnih operatorjev in izrazov v tem učbeniku ne bomo podrobneje obravnavali. Tu omenimo le, da logični operator ALI med seboj združuje dva pogoja v en sam pogoj, ki je na koncu bodisi ***izpolnjen*** bodisi ***ni izpolnjen***. Pogoj, ki ga sestavljata dva pogoja, združena z operatorjem ALI, je izpolnjen natanko takrat, ko je izpolnjen vsaj eden od pogojev, ki ju ta operator združuje. Pri združevanju pogojev z logičnimi operatorji je pomembno tudi to, da posamezne pogoje pišemo v polni obliki. Ne bi bilo prav, če bi gornji pogoj zapisali

kot `n == 3 || 4`. Na strani 136 v dodatku A boste našli primer, ki dodatno osvetljuje delovanje logičnega operatorja ALI.

Takšno zaporedje stavkov `if...else`, kjer si sledijo `if`, poljubno število kombinacij `else if` in na koncu `else`, se uporablja, kadar moramo pregledati več kot dve možni vrednosti (ali območji vrednosti) kakšne spremenljivke ter se odzvati na največ eno od teh vrednosti. V nekaterih jezikih obstaja celo skovanka `elif`, ki se uporablja kot okrajšava kombinacije `else if`.

Ko gornji program zaženemo v brskalniku, dobimo v konzoli takšen izpis:



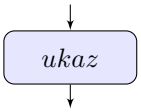
Naloga 2.2 Za vajo razmislite in preizkusite, kaj se zgodi, če iz zadnjega programa odstranimo prvi dve besedi `else`, kar nas pripelje do naslednjega programa:

```
n = 3;
console.log(n);
n = n % 100;   //ostanek pri deljenju s 100
if (n == 1) {
  console.log("burger");
}
if (n == 2) {
  console.log("burgerja");
}
if (n == 3 || n == 4) {
  console.log("burgerji");
}
else {
  console.log("burgerjev");
}
```

Razmislite tudi, s čim moramo zdaj nadomestiti preostalo besedo `else`, da bo program spet deloval tako kot prej. Pomagate si lahko z ***diagramom poteka***, ki ga bomo spoznali v naslednjem razdelku.

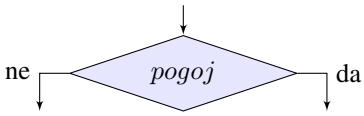
2.3 Diagram poteka

Diagram poteka (angl. flowchart) je grafični način zapisovanja algoritmov, kjer preprosto sledimo puščicam in izvajamo ukaze, ki so zapisani v grafičnih blokih. Izkaže se, da lahko sleherni algoritem sestavimo z uporabo dveh osnovnih blokov: z ***ukaznim*** blokom in z ***odločitvenim*** blokom. Ukazni blok prikazujemo s pravokotnikom, ki ima en vhod in en izhod:



Ko vstopimo v ukazni blok, izvršimo ukaz, ki se nahaja v bloku, in izstopimo iz bloka v smeri izhodne puščice.

Odločitveni blok je v diagramu poteka ponazorjen z rombom, ki ima en vhod in dva izhoda. Preko vhodne puščice vstopimo v romb in preverimo pravilnost pogoja, ki je v njem zapisan. Če je pogoj izpolnjen, izstopimo v smeri puščice, ob kateri piše *da*, sicer izstopimo v smeri puščice, ob kateri piše *ne*:

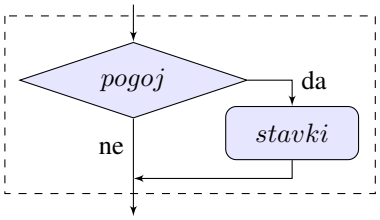


2.4 Pogojna stavka `if` in `if...else`

Spoznali smo že, da je sintaksa pogojnega stavka (angl. conditional statement) `if` sestavljena iz besede `if`, pogoja, ki je zapisan v okroglih oklepajih, in enega ali več stavkov, ki so zapisani v zavitih oklepajih:

```
if ( pogoj ) { stavki }
```

Ta stavek lahko ponazorimo tudi z diagramom poteka:

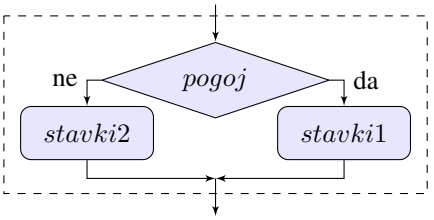


Ena od pomembnih podrobnosti, ki jo opazimo, je ta, da je vsebina ukaznega bloka v diagramu v resnici koda med zavitima oklepajema stavka `if`. Iz diagrama se lepo vidi, da v primeru, ko pogoj ni izpolnjen, stavek enostavno zaobide vso kodo, ki je zapisana v zavitih oklepajih. Okrog diagrama opazimo še črtkan pravokotnik. Ta pravokotnik ponazarja dejstvo, da ima stavek `if` en sam vhod in en sam izhod, ki ga lahko obravnavamo kot samostojni ukazni blok. Tako lahko na primer v ukazni blok v gornjem diagramu poteka vstavimo še en stavek `if`.

Stavek `if...else` ima takšno sintakso:

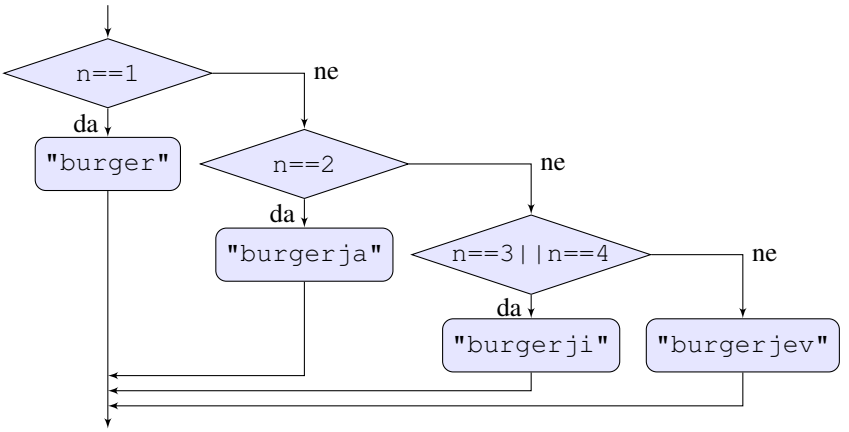
```
if ( pogoj ) { stavki1 } else { stavki2 }
```

Stavek deluje kot kretnica, ki povzroči, da se izvedejo bodisi stavki $stavki1$ bodisi stavki $stavki2$, kar ponazorimo z naslednjim diagramom:

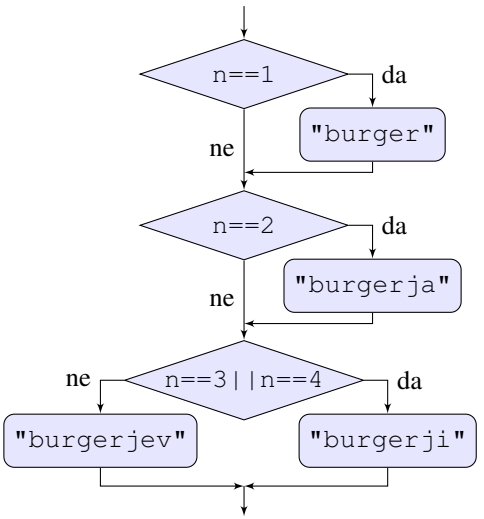


Spet opozorimo na pomembno dejstvo, da ima stavek en sam vhod in en sam izhod, in navzven deluje kot samostojen ukazni blok (črtkan pravokotnik). Tako lahko v vsakega od obeh ukaznih blokov v gornjem diagramu vstavimo tako še en stavek `if` kot tudi stavek `if...else`.

Vrnimo se še nekoliko k našim burgerjem ter narišimo diagram poteka za del programa (od četrte vrstice naprej), ki smo ga zapisali na strani 8:



Tako pa je videti diagram, če iz programa odstranimo prvi dve besedi `else` (program na strani 9):



Iz diagrama se lepo vidi, da program zdaj za vrednosti spremenljivke $n = 1$ in $n = 2$ izpiše poleg pravilne oblike besede (*burger* oz. *burgerja*) še napačno obliko besede (*burgerjev*).

S tem ko smo iz originalnega programa odstranili dve besedi `else`, smo povzročili precejšnjo razliko v strukturi kode. V prvi različici programa imamo opravka s tremi stavki `if...else`, kjer je vsak naslednji stavek vstavljen v enega od ukaznih blokov prejšnjega. Na ta način v vsakem primeru dobimo natanko eno od možnih oblik besede *burger*. V drugi različici programa imamo opravka z dvema stavkoma `if` in enim stavkom `if...else`, ki si sledijo eden za drugim. Tako se nam lahko pripeti, da pridemo tudi do dveh oblik besede *burger*.

2.5 Ponavljalni stavek

Poskušajmo napisati algoritem, ki reši naslednji problem:

Vhod: nenegativno celo število (n);
Zahtevani izhod: faktoriela podanega števila ($n!$);

Faktoriela (ali fakulteta, kot jo tudi poimenujemo) naravnega števila n je v matematiki določena kot produkt pozitivnih celih števil, manjših ali enakih n:

$$n! = 1 \times 2 \times 3 \times \cdots \times n. \tag{2.2}$$

Po dogovoru velja, da je $0! = 1$.

Opazimo, da se v definiciji faktoriele $(n-1)$-krat ***ponovi*** operacija množenja, zato lahko problem rešimo le tako, da v algoritem uvedemo določeno ponavljanje. Če pomislimo natančneje, se poleg množenja ponavlja še ena operacija: vsakokrat moramo od spremenljivke n odšteti konstantno vrednost ena. Glede na to, da bomo končni produkt med izvajanjanjem algoritma gradili postopoma in ga »nalagali« v izhodno spremenljivko, moramo za to spremenljivko izbrati primerno začetno vrednost. Logična izbira za začetno vrednost je ena. Med drugim tudi zato, ker je to tudi končna rešitev problema v primeru, ko je n enak nič ali ena in množenja sploh ne izvajamo. Takole je videti algoritem:

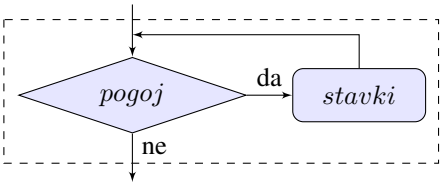
```
Vhod:  nenegativno celo število n;
Izhod:  f (faktoriela vhodne vrednosti);
f ← 1;
Ponavljaj, dokler je n > 1:
{
  f ← f × n;
  n ← n − 1;
}
Sporoči:  f;
```

2.6 Ponavljalni stavek `while`

V jeziku JavaScript ponavljanje dosežemo s stavkom `while`, ki ima podobno sintakso kot stavek `if`:

```
while ( pogoj ) { stavki }
```

Razlika je edino ta, da se stavek `while` znova in znova vrača na preverjanje pogoja, dokler na koncu pogoj ni več izpolnjen. Ponavljalnemu stavku rečemo tudi ***zanka*** (angl. loop), ker pot, ki se vrača nazaj na preverjanje pogoja, v diagramu poteka ustvarja zanko:



Spet vidimo, da stavek navzven deluje kot ukazni blok z enim vhodom in enim izhodom, kar je ponazorjeno s črtkanim pravokotnikom.

Vrnimo se k našemu problemu računanja faktoriele in zapišimo njegovo rešitev v jeziku JavaScript:

```
n = 6;
f = 1;
while (n > 1) {
  f = f * n;
  n = n - 1;
}
console.log(f);
```

Brž ko smo v program uvedli ponavljanje, nas začne skrbeti, ali se bo algoritem tudi ***ustavil*** (angl. terminate). Pravimo, da je algoritem ***ustavljiv***, če se bo za vse veljavne vhodne podatke vedno prej ali slej ustavil. Določanje ustavljivosti je splošno matematično ***neodločljiv*** (angl. undecidable) problem, saj ne obstaja univerzalni postopek, ki bi za poljuben program z gotovostjo potrdil, da se bo na koncu zares ustavil. Obstajajo pa delne rešitve in postopki, in predvsem za enostavnejše algoritme je ustavljivost moč pokazati. Za naš zadnji algoritem lahko na primer hitro ugotovimo, da je ustavljiv. Spremenljivka n se namreč neprestano zmanjšuje in mora prej ali slej postati manjša ali enaka ena, in takrat pogoj za ponavljanje ni več izpolnjen. Potreben pogoj za ustavljivost je, da se vsaj ena od spremenljivk, ki nastopajo v pogoju ponavljalnega stavka, znotraj tega stavka spreminja.

2.6.1 Evklidovo deljenje

Lotimo se še enega, na videz preprostega problema:

Vhod: celoštevilski vrednosti a in b;
Zahtevani izhod: ostanek pri deljenju a/b;

Pri reševanju problema si postavimo še dodatno omejitev, da ne smemo uporabljati vgrajenega operatorja ostanka celoštevilskega deljenja (`%`).

Preden začnemo, si zastavimo pomembno vprašanje, ali je za ***vse veljavne vhodne podatke natanko določen tudi izhod***. Takoj ugotovimo, da gornji problem ni določen za primer, ko je $b = 0$. Kaj pa za negativne vrednosti vhodnih podatkov? Če preizkusimo delovanje operatorja celoštevilskega deljenja v jeziku JavaScript, ugotovimo, da ta operator izračuna ostanek iz absolutnih vrednosti deljenca in delitelja, predznak ostanka pa je enak predznaku deljenca. Tako imata na primer izraza `7%3` in `7%-3` vrednost 1, izraza `-7%3` in `-7%-3` pa -1.

Naloga 2.3 Sami poskusite napisati program (brez uporabe operatorja `%`), ki bo računal ostanek natanko tako kot operator `%` v jeziku JavaScript. Uporabite lahko funkcijo za izračun absolutne vrednosti `Math.abs`.

Marsikdo bo začuden, če bo poskusil izračunati vrednosti izrazov `-7%3` in `7%-3` na primer v programskem jeziku Python. Dobil bo vrednosti 2 oziroma -2. Python pri določanju vrednosti ostanka sledi tako imenovanemu ***Evklidovemu deljenju***, predznak pa določa glede na predznak delitelja.

Spoznali smo, da je naš zadnji problem slabo določen, saj za negativne vrednosti deljenca in/ali delitelja obstaja več načinov računanja ostanka. Da bi problem dokončno določili, se dogovorimo, da bomo ostanek računali po ***Evklidovem izreku celoštevilskega deljenja***, ki pravi:

Za dve celi števili a in b, kjer je $b > 0$, obstajata unikatni celi števili q in r, tako da velja:

$$a = b\,q + r \tag{2.3}$$

in

$$0 \leq r < b. \tag{2.4}$$

Štiri spremenljivke, ki nastopajo v izreku, se imenujejo deljenec (a), delitelj (b), količnik (q) in ostanek (r). Računanje količnika in ostanka iz deljenca in delitelja pa se imenuje ***deljenje*** oziroma natančneje ***Evklidovo deljenje***.

Ker je naš cilj izračunati ostanek pri deljenju, najprej iz enačbe (2.3) izrazimo ostanek:

$$r = a - b\,q. \tag{2.5}$$

Glede na vrednost deljenca (a) bomo računanje ostanka razdelili na dva primera. Vzemimo najprej primer, ko je $a \geq 0$. Zaradi (2.4) mora biti tudi $q \geq 0$. Zdaj lahko poiščemo r tako, da začnemo s $q = 0$ in nato večamo vrednost spremenljivke q za ena toliko časa, dokler ni izpolnjen pogoj (2.4):

$$\begin{array}{ll} r \leftarrow a & (q = 0) \\ r \leftarrow a - b & (q = 1) \\ r \leftarrow a - 2\,b & (q = 2) \\ r \leftarrow a - 3\,b & (q = 3) \\ \ldots & \end{array}$$

Podobno ugotovimo za primer, ko je $a < 0$. Zaradi (2.4) mora biti tudi $q < 0$. Zdaj začnemo s $q = -1$ in manjšamo vrednost spremenljivke q za ena, dokler ni izpolnjen pogoj (2.4):

$$\begin{array}{ll} r \leftarrow a + b & (q = -1) \\ r \leftarrow a + 2\,b & (q = -2) \\ r \leftarrow a + 3\,b & (q = -3) \\ \ldots & \end{array}$$

Na koncu združimo obe možnosti (t.j. za $a \geq 0$ in $a < 0$) in dobimo naslednji algoritem:

Vhod: celi števili a in b, $b > 0$;

```
Izhod:  ostanek pri Evklidovem deljenju (r);
r ← a;
Če je a večji ali enak 0:
{
  Ponavljaj, dokler je r < 0 ali r ≥ b:
  {
    r ← r − b;
  }
}
sicer:
{
  Ponavljaj, dokler je r < 0 ali r ≥ b:
  {
    r ← r + b;
  }
}
Sporoči:  r;
```

V primeru, ko je $a < 0$, se bo drugi ponavljalni stavek vedno izvršil vsaj enkrat. Namreč, če je $a < 0$, potem je na začetku nujno tudi $r < 0$, kar pomeni, da je pogoj za ponavljanje ob prvem vstopu v to zanko vedno izpolnjen. Ta ugotovitev je popolnoma v skladu s pred tem zapisanim algoritmom za $a < 0$, ki se začne z $r \leftarrow a + b$ ($q = -1$).

Opazimo tudi, da je v prvem ponavljalnem stavku odveč pogoj $r < 0$, saj ta nikoli ne bo izpolnjen. Ostanek r dobi namreč pred tem nenegativno vrednost (saj je $a \geq 0$), potem pa ga zmanjšujemo za b (vendar le, če je $r \geq b$). Iz podobnih razlogov je odveč pogoj $r \geq b$ v drugem ponavljalnem stavku.

In še program v jeziku JavaScript:

```
a = -7; //deljenec
b = 3;  //delitelj
r = a;  //ostanek
if (a >= 0) {
  while (r < 0 || r >= b) {
    r = r - b;
  }
}
else {
  while (r < 0 || r >= b) {
    r = r + b;
  }
}
console.log(r);
```

Opazimo lahko, da sta stavka `while`, ki se pojavljata v gornjem programu, precej podobna. Razlikujeta se le v predznaku pred spremenljivko `b`, ki je odvisen od predznaka deljenca. Vesten programer se bo skušal izogibati takšnemu podvajanju kode, ki predstavlja nepotrebno dodatno tveganje za programske napake. Poskusimo združiti oba stavka `while` v enega samega, tako da uvedemo pomožno spremenljivko `predznak`, ki bo hranila predznak deljenca. Program poskusite napisati sami, če pa vam ne bo uspelo, si oglejte možno rešitev:

```
a = -7; //deljenec
b = 3;  //delitelj
r = a;  //ostanek
predznak = 1;
```

```
if (a < 0) {
  predznak = -1;
}
while (r < 0 || r >= b) {
  r = r - predznak * b;
}
console.log(r);
```

Morda se zdi nenavadno, da je ostanek celoštevilskega deljenja $7/3$ enak 1, medtem ko je ostanek pri deljenju $-7/3$ enak 2. Pojav ima popolnoma praktično uporabo in ga lahko pojasnimo.

Recimo, da je ura dve in nas zanima, koliko bo ura čez 21 ur. Ker se ura obrne na vsakih 12 ur, gre pri tem v resnici za računanje po ***modulu***[1] 12 (ostanek pri deljenju z 12): na vsakih 12 ur pridemo v isto točko. Izračunamo $(2 + 21) \bmod 12$ in dobimo 11, kar pomeni, da bo čez 21 ur ura 11. Podobno lahko izračunamo, koliko je bila ura pred tremi urami: $(2 - 3) \bmod 12$. Tudi tokrat dobimo odgovor, da je bila ura 11.

2.6.2 Upravljanje s števkami

Naslednji primer prikazuje, kako je mogoče v določenem desetiškem številu upravljati s posameznimi števkami (oz. ciframi). Naloga se glasi takole:

> ***Vhod:*** celoštevilska vrednost n;
> ***Zahtevani izhod:*** vrednost, ki jo dobimo, če v n obrnemo vrstni red števk;

Vzemimo za primer, da imamo na vhodu število 1265. Program nam bo vrnil število 5621. Nalogo lahko rešimo tako, da števke eno za drugo jemljemo s konca vhodne vrednosti (n) in jih dodajamo na konec izhodne vrednosti ($nIzhod$). Dobimo naslednje zaporedje stanj:

```
n:       1265
nIzhod:     0

n:        126
nIzhod:     5

n:         12
nIzhod:    56

n:          1
nIzhod:   562

n:          0
nIzhod:  5621
```

Opazimo, da se pri tem, ko prestavljamo števke iz enega števila v drugo, števili pomikata v desno oziroma levo. Takšno pomikanje lahko dosežemo z deljenjem oziroma množenjem z deset. Prestavljanje števke iz enega števila v drugo pa dosežemo z odštevanjem oziroma prištevanjem ostanka pri deljenju z deset. Takole je videti dokončan algoritem:

[1] V matematiki govorimo o modularni aritmetiki, kadar se celoštevilske vrednosti »ovijajo«, kakor hitro dosežejo določeno mejno vrednost. Primer takšne modularne aritmetike je računanje z urami. Operatorju, ki računa ostanek celoštevilskega deljenja, zato včasih pravimo tudi ***modulo***. Operator, ki računa ostanek pri deljenju celih števil, se v nekaterih jezikih označuje z znakom %, v drugih jezikih pa z besedico *mod*.

```
Vhod:  celo število n;
Izhod:  vhodno število z obrnjenim vrstnim redom števk (nIzhod)
nIzhod ← 0;
Ponavljaj, dokler je n različen od nič:
{
  nIzhod ← nIzhod × 10;          //Opomba: pomik v levo
  nIzhod ← nIzhod + (n%10);      //Opomba: dodajanje zadnje števke k izhodu
  n ← n − (n%10);                //Opomba: odvzemanje zadnje števke od vhoda
  n ← n/10;                      //Opomba: pomik v desno
}
Sporoči:  nIzhod;
```

Vrstni red štirih vrstic kode znotraj ponavljalnega stavka je pomemben in ga ne moremo spreminjati. Razmislite, zakaj.

Tako pa je videti dokončan program v jeziku JavaScript:

```
n = 12345;
nIzhod = 0;
while (n != 0) {
  nIzhod = nIzhod * 10;
  nIzhod = nIzhod + (n % 10);
  n = n - (n % 10);
  n = n / 10;
}
console.log(nIzhod);
```

V programu smo uporabili primerjalni operator, ki preverja, ali sta njegova leva in desna stran različni (`!=`). Stavek `while` se bo torej izvajal, dokler bo `n` različen od nič.

Ker operator ostanka (`%`) v jeziku JavaScript računa ostanek iz absolutnih vrednosti, predznak pa določi glede na predznak deljenca, deluje gornji program tudi za negativne vrednosti.

Po krajšem razmisleku in preizkusu lahko ugotovimo, da se pri vrednostih, ki se končajo z eno ali več ničlami, te ničle pri obračanju izgubijo. Na primer, za `n = 341000` dobimo na izhodu število 143. Vendar to ni težava, saj je vrednost brez vodilnih ničel matematično popolnoma enaka vrednosti z vodilnimi ničlami (t.j. $000143 = 143$).

Naloga 2.4 Za vajo napišite program za rešitev naslednjega problema, kjer morate prav tako iz podane številke izbirati posamezne števke:

Vhod: nenegativno celo število n;
Zahtevani izhod: sporočilo o tem, ali je n Armstrongovo število;

V razvedrilni matematiki je Armstrongovo število tako število, ki je vsota vseh svojih števk, potenciranih s številom teh števk. Na primer, 371 je Armstrongovo število, kajti $3^3 + 7^3 + 1^3 = 371$. Posamezne števke smo potencirali s tri, ker število 371 vsebuje tri števke.

Z napisanim programom preizkusite, katero od naslednjih števil je Armstrongovo število: 5, 153, 276, 407, 1634, 2971 in 8208.

Pomoč: Za potenciranje lahko uporabite funkcijo `Math.pow`. Funkcija sprejme dva parametra in prvega potencira z drugim. Na primer, `Math.pow(2, 5)` vrne vrednost potence 2^5.

2.7 Naloge

Preden nadaljujemo z novim poglavjem, je tu še nekaj nalog, ki jih poskusite rešiti sami:

Naloga 2.5 Zapišite problem iskanja največjega med tremi realnimi števili. Nato narišite diagram poteka in zapišite algoritem za rešitev problema ter napišite pripadajoč program v jeziku JavaScript.

Naloga 2.6 Ali je naslednji problem podan tako, da je za vsak veljaven vhod natanko definiran tudi izhod?

> ***Vhod:*** dolžine treh stranic trikotnika a, b in c ($a > 0$, $b > 0$, $c > 0$);
> ***Zahtevani izhod:*** ploščina trikotnika s podanimi dolžinami stranic;

Popravite definicijo ter napišite algoritem in program za rešitev problema.

Pomoč: Ploščino trikotnika lahko iz dolžin treh stranic izračunamo po tako imenovani Heronovi enačbi:

$$p = \sqrt{s(s-a)(s-b)(s-c)}, \tag{2.6}$$

pri čemer je s polovični obseg trikotnika:

$$s = \frac{a+b+c}{2}. \tag{2.7}$$

Kvadratni koren v jeziku JavaScript izračunate s funkcijo `Math.sqrt`.

Naloga 2.7 Narišite diagram poteka in zapišite algoritem ter sestavite program za naslednji problem:

> ***Vhod:*** poljubna letnica iz gregorijanskega koledarja;
> ***Zahtevani izhod:*** sporočilo o tem, ali je leto prestopno (t.j. »Leto je prestopno.« oz. »Leto ni prestopno.«);

Pomoč: Leto je prestopno, če je deljivo s 4, vendar ni deljivo s 100. Leto, ki je deljivo s 400, je spet prestopno.

Dodatek: Spremenite in dopolnite program tako, da bo rešil naslednji problem:

> ***Vhod:*** dve letnici iz gregorijanskega koledarja $letn1$ in $letn2$ ($letn1 < letn2$);
> ***Zahtevani izhod:*** seznam vseh prestopnih let z letnicami med vključno $letn1$ in $letn2$. Če je seznam prazen, naj program izpiše »V tem obdobju ni prestopnih let.«;

Naloga 2.8 Narišite diagram poteka, zapišite algoritem ter sestavite program za rešitev naslednjega problema:

Vhod: naravno število $n > 1$;
Zahtevani izhod: sporočilo o tem, ali je n praštevilo;

Po definiciji je ***praštevilo*** (angl. prime number) vsako naravno število $n > 1$, ki ima natanko dva pozitivna delitelja: število ena in samo sebe. Prvih deset praštevil je: $2, 3, 5, 7, 11, 13, 17, 19, 23$ in 29.

Pomoč: Nalogo rešite tako, da poskušate vhodno število n deliti z vsakim od naravnih števil med vključno 2 in $n-1$. Pri tem štejte, kolikokrat se je deljenje izšlo brez ostanka. Če se je vsaj eno od takšnih deljenj izšlo brez ostanka, potem n ni praštevilo.

Zanimivost: Trenutno (januar 2020) največje znano praštevilo (odkrito l. 2018) je Mersennovo število $2^{82\,589\,933} - 1$, ki ima, če ga zapišemo v desetiškem zapisu, $24\,862\,048$ mest[a]. Število je ***Mersennovo število***, če je njegova vrednost za ena manjša od potence števila dve, zaradi česar je takšno število v dvojiškem zapisu sestavljeno iz samih enic. Veliko največjih praštevil je Mersennovih števil – od decembra 2018 je kar osem največjih znanih praštevil Mersennovih števil. Obstaja mednarodni projekt (GIMPS, Great Internet Mersenne Prime Search), v katerem sodelujejo prostovoljci z vsega sveta v iskanju Mersennovih praštevil. Pri tem uporabljajo posebno prosto dostopno programsko orodje. Trenutno ponuja GIMPS nagrado v višini $3\,000$ ameriških dolarjev za odkritje novega Mersennovega praštevila velikosti do sto milijonov desetiških mest z uporabo njihovega programskega orodja. Poleg tega ponuja EFF (Electronic Frontier Foundation) nagrado v višini $150\,000$ ameriških dolarjev za prvo praštevilo z vsaj sto milijoni desetiških mest in nagrado v višini $250\,000$ ameriških dolarjev za prvo praštevilo z vsaj milijardo desetiških mest.

[a] Če bi hoteli takšno številko natisniti, bi potrebovali knjigo s približno $8\,000$ (!) stranmi.

Naloga 2.9 Narišite diagram poteka, zapišite algoritem ter sestavite program za rešitev naslednjega problema:

Vhod: naravno število n;
Zahtevani izhod: seznam prafaktorjev podanega števila n;

Prafaktor ali ***praštevilski delitelj*** naravnega števila je vsak faktor, ki je praštevilo in da z ostalimi prafaktorji enoličen zmnožek, ki je število samo. Na primer, število 44044 je zmnožek svojih prafaktorjev 2, 2, 7, 11, 11 in 13.

Pomoč: Za definicijo praštevila glej prejšnjo nalogo.

Naloga 2.10 Francoski matematik François Viète je prvi v Evropi odkril neskončno zaporedje, ki izračuna število π. Število izračunamo kot naslednji neskončni zmnožek:

$$\frac{\pi}{2} = \frac{2}{\sqrt{2}} \times \frac{2}{\sqrt{2+\sqrt{2}}} \times \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \times \cdots \tag{2.8}$$

Napišite program za rešitev naslednjega problema:

Vhod: število upoštevanih členov za izračun števila π po enačbi (2.8);
Zahtevani izhod: izračunano število π;

Pomoč: Za računanje kvadratnega korena uporabite funkcijo `Math.sqrt`.

Naloga 2.11 Narišite diagram poteka, zapišite algoritem ter sestavite program za rešitev naslednjega problema:

Vhod: dve naravni števili m in n;
Zahtevani izhod: največji skupni delitelj naravnih števil m in n;

Največji skupni delitelj dveh števil je največji od vseh deliteljev, ki so številoma skupni.

Naloga 2.12 Napišite program za rešitev naslednjega problema:

Vhod: nenegativno celo število n;
Zahtevani izhod: vrednost in zaporedna številka prvega člena Fibonaccijevega zaporedja, ki je večji od n;

Fibonaccijevo zaporedje je zaporedje števil, ki se začne s številoma 0 in 1, vsako naslednje število pa je vsota njegovih dveh predhodnikov. Prvih deset števil Fibonaccijevega zaporedja je tako: 0, 1, 1, 2, 3, 5, 8, 13, 21 in 34.

Naloga 2.13 Napišite program, ki reši naslednji problem:

Vhod: naravno število n;
Zahtevani izhod: tabelirana poštevanka števil do n, pri čemer naj se izpišejo le diagonala in vrednosti pod njo;

Na primer, za $n = 5$ naj se v konzoli izpiše takšna tabela:

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

Opomba: Ker funkcija `console.log` sleherni podatek izpiše v novo vrstico konzole, bodo gornje številke v konzoli v resnici ena pod drugo, in sicer v takšnem vrstnem redu: $1, 2, 4, 3, 6, 9, 4, 8, \cdots$.

Naloga 2.14 Napišite algoritem in program za rešitev naslednjega problema:

Vhod: naravno število n;

Zahtevani izhod: vsa naravna števila, po vrsti od 1 do n. Namesto vseh števil, ki so deljiva s tri, naj se izpiše beseda *bim*. Namesto števil, ki so deljiva s pet, naj se izpiše beseda *bam*. Namesto števil, ki so deljiva tako s tri kot tudi s pet, pa naj se izpiše beseda *bimbam*;

Na primer, za $n = 16$ naj se izpiše naslednje zaporedje:

```
1 2 bim 4 bam bim 7 8 bim bam 11 bim 13 14 bimbam 16
```

Zanimivost: Različni viri navajajo, da se ta naloga včasih uporablja pri preverjanju kandidatov za delovno mesto programerja. Presenetljivo, marsikdo ima z nalogo težave.

PRAZNA STRAN

3. POGLAVJE

TABELA (PODATKOVNA STRUKTURA)

Mnogo problemov, ki jih rešujemo z računalniškimi programi, zahteva večje količine podatkov. V takih primerih je za hranjenje posameznih vrednosti nepraktično ali celo neizvedljivo uporabljati običajne (t.j. skalarne) spremenljivke. Na pomoč pokličemo ***tabele*** (angl. array). Enorazsežni tabeli bomo včasih rekli tudi ***vektor***, dvorazsežni pa ***matrika***. V psevdo kodi bomo tabele zapisovali s krepkim tiskom, posamezne ***elemente*** tabele pa z običajnim tiskom in dodanim indeksom, ki bo označeval zaporedno številko elementa v tabeli. Elementi enorazsežnostnih tabel bodo imeli en, elementi dvorazsežnostnih tabel pa dva indeksa (t.j. zaporedno številko vrstice in stolpca elementa). Kadar bomo izpisovali celo tabelo s posameznimi elementi, bomo elemente postavljali v par oglatih oklepajev in jih ločevali z vejicami. Na primer, enorazsežnostno tabelo $\boldsymbol{t}$ z n elementi bomo zapisovali takole:

$$\boldsymbol{t} = [t_0, t_1, \cdots, t_{n-1}]. \tag{3.1}$$

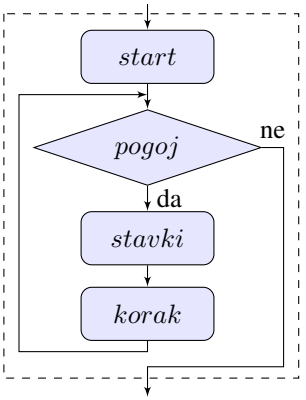
Ker se v večini programskih jezikov štetje elementov začne z nič, bomo tudi mi prvi element označili z indeksom 0, drugega z 1 in tako naprej do n-tega elementa, ki bo imel indeks $n - 1$.

3.1 Ponavljalni stavek `for`

Za delo s tabelami pogosto uporabljamo ponavljalni stavek `for`, ki se v jeziku JavaScript zapiše takole:

```
for ( start ; pogoj ; korak ) { stavki }
```

Poleg obveznih oklepajev se v stavku pojavita še dve podpičji, ki sta tudi obvezni. Delovanje stavka prikazuje naslednji diagram poteka:



Vidimo, da se ta stavek ne razlikuje prav dosti od stavka `while`. Ker se stavek `for` uporablja navadno takrat, kadar imamo opravka s kakršnimkoli štetjem, vsebuje med parom okroglih oklepajev dva dodatna izraza: *start* in *korak*. Čeprav lahko popolnoma enako delovanje dosežemo s stavkom `while`, pa ta dva dodatna izraza pripomoreta k preglednejšemu nadzoru nad štetjem: *start* nam pove, kje začnemo šteti, *pogoj* nam pove, kje končamo, *korak* pa določa korak štetja. Na primer, naslednji program izpiše vsa soda števila od dve do vključno deset:

```
for (i = 2; i <= 10; i = i + 2) {
  console.log(i);
}
```

Operator <=, ki je videti kot puščica v levo, je v resnici primerjalni operator $\leq$. Operator preverja, ali je vrednost na njegovi levi manjša ali enaka vrednosti na njegovi desni.

Bodite pozorni na vrstni red izvajanja operacij v stavku `for`: stavki v paru zavitih oklepajev se izvedejo ***pred*** zadnjim izrazom v paru okroglih oklepajev.

Naloga 3.1 Za vajo rešite z uporabo stavka `for` nalogo 2.8 na strani 18 ter nalogo 2.14 na strani 20.

3.2 Enorazsežnostna tabela

Za ilustracijo uporabe tabele rešimo najprej naslednji problem:

> ***Vhod:*** tabela $\boldsymbol{x}$ z n realnimi elementi, $\boldsymbol{x} = [x_0, x_1, \cdots, x_{n-1}], n > 0$;
> ***Zahtevani izhod:*** srednja vrednost podanih elementov ($\overline{x}$) in seznam (množica) elementov, ki se od srednje vrednosti ne razlikujejo za več kot 10% (t.j. iščemo množico $\{x_i : 0{,}9\,\overline{x} \leq x_i \leq 1{,}1\,\overline{x}\}$);

Očitno je, da bomo morali problem rešiti v dveh korakih:

```
Izračunaj srednjo vrednost (x_sr);
Izpiši elemente tabele, ki se od srednje vrednosti
  ne razlikujejo za več kot 10%;
```

Srednjo vrednost spremenljivke x smo v algoritmu označili z x_{sr}, ker v jeziku JavaScript ne moremo uporabiti prečne črte. Seveda ne moremo uporabiti niti podpisanega besedila, zato bomo srednjo vrednost v končnem programu označili z `xSr`.

Prvi korak gornjega algoritma zahteva, da seštejemo vse elemente tabele in jih delimo z n. V drugem koraku za vsak element tabele posebej preverimo, ali leži v zahtevanem območju okrog srednje vrednosti. Če leži v tem območju, ga izpišemo. Takole je videti dokončan algoritem:

```
Vhod:  tabela x z n realnimi elementi;
Izhod:  srednja vrednost podanih elementov ter seznam
  elementov, ki se ne razlikujejo za več kot 10% od
  srednje vrednosti;
x_sr ← 0;
 //Opomba: Izračunaj in izpiši srednjo vrednost:
Ponovi za vsak i od 0 do n − 1 s korakom 1:
{
  x_sr ← x_sr + x_i;
}
x_sr ← x_sr/n;
Sporoči:  x_sr;
 //Opomba: Izpiši elemente, ki ustrezajo pogoju:
Ponovi za vsak i od 0 do n − 1 s korakom 1:
{
  Če velja 0.9 x_sr ≤ x_i ≤ 1.1 x_sr:
  {
    Sporoči:  x_i;
  }
}
```

Prevedimo zdaj ta algoritem v jezik JavaScript:

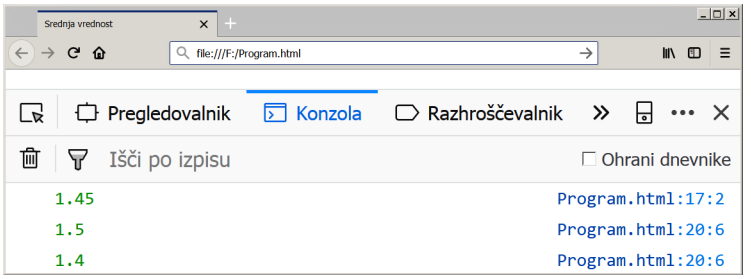
```
x = [3.0, -1.8, 1.5, 5.1, -0.5, 1.4];
n = 6;
xSr = 0;
for (i = 0; i < n; i = i + 1) {
  xSr = xSr + x[i];
}
xSr = xSr / n;
console.log(xSr);
for (i = 0; i < n; i = i + 1) {
  if (0.9 * xSr <= x[i] && x[i] <= 1.1 * xSr) {
    console.log(x[i]);
  }
}
```

Prva stvar, ki jo v programu opazimo, je način zapisovanja indeksa (oziroma ***zaporedne številke***) elementa v tabeli. V jeziku JavaScript se indeks zapisuje v par oglatih oklepajev, postavljenih za imenom tabele.

V gornjem programu opazimo tudi operator `&&`, ki smo ga uporabili v pogoju stavka `if`. Operator predstavlja logični operator IN, ki tako kot logični operator ALI združuje dva pogoja v en sam pogoj. Dobljeni pogoj je izpolnjen samo takrat, kadar sta izpolnjena

oba od pogojev, ki ju operator IN združuje. Na strani 136 v dodatku A boste našli primer, ki dodatno osvetljuje delovanje logičnega operatorja IN.

Ko program zaženemo, dobimo naslednji izpis:



Srednja vrednost elementov iz tabele x je enaka $1{,}45$, od nje pa se za največ 10% razlikujeta le elementa $x_2 = 1{,}5$ in $x_5 = 1{,}4$.

3.3 Pretvorba številskih sistemov

Vsako vrednost je mogoče zapisati v različnih številskih sistemih. Nam najbližji je desetiški zapis, kjer posamezne števke predstavljajo koeficiente pred ustreznimi potencami ***osnove*** deset. Na primer, vrednost 6481 lahko zapišemo kot naslednjo vsoto:

$$6481 = 6 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 1 \times 10^0.$$

Kadar zapisujemo števila v sistemu, ki ni desetiški, zapišemo osnovo sistema kot podpisano številko na koncu zapisane vrednosti. Na primer, število 1101_2 je zapisano v dvojiškem sistemu, ker ima na koncu podpisano dvojko. Njegovo vrednost lahko prav tako izračunamo kot vsoto števk (oziroma ***bitov***[1]), pomnoženih z ustreznimi potencami osnove, ki pa je tokrat enaka dve:

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13.$$

Pri pretvarjanju vrednosti iz desetiškega v poljuben drug zapis nam bo v pomoč naslednje dejstvo: Množenje in deljenje z vrednostjo, ki je enaka osnovi sistema, pomika posamezne števke števila v levo oziroma desno. Pri deljenju z osnovo je vsakokratni ostanek enak števki, ki se nahaja na zadnjem mestu deljenca. Ker je vrednost ostanka odvisna le od vrednosti deljenca in delitelja, ne pa tudi od številskega sistema, lahko poljubno število pretvorimo v poljuben številski sistem zgolj z deljenjem z ustrezno osnovo. Na primer, če želimo število 13 pretvoriti v dvojiški zapis, bomo to število štirikrat delili z dve in po vrsti dobili ostanke 1, 0, 1 in 1. Natanko iste ostanke bi dobili, če bi z dve delili število 1101_2 (ki predstavlja desetiško vrednost 13).

Napišimo zdaj program za rešitev naslednjega problema:

> ***Vhod:*** celo število x med 0 in 255 (v desetiškem zapisu);
> ***Zahtevani izhod:*** število x, zapisano v dvojiškem zapisu (v obliki tabele ***b*** z osmimi elementi);

[1] V računalništvu pravimo števkam v dvojiškem zapisu biti.

Število 255 predstavlja največjo desetiško vrednost, ki jo lahko zapišemo z osmimi biti, zato smo za vhodno vrednost x tudi izbrali takšno omejitev. Ker se ostanki pri deljenju pojavljajo v obratnem vrstnem redu, kot jih potrebujemo, bomo ostanke najprej zapisali v tabelo. To bomo storili v obrnjenem vrstnem redu. Vsebino tabele bomo na koncu izpisali v konzoli. Takole je videti algoritem:

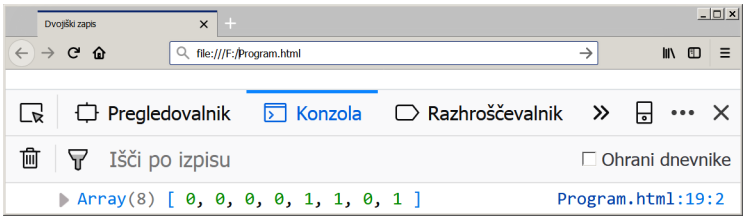
```
Vhod:  celo število x med 0 in 255;
Izhod:  tabela b = [b_0, b_1, ··· , b_7] z osmimi biti števila x;
i ← 7;  //Opomba: Indeks nastavimo na zadnji element tabele.
Ponavljaj, dokler je x večji od nič:
{
  b_i ← x%2;    //Opomba: Ostanek pri deljenju vstavimo v tabelo na i-to mesto.
  i ← i − 1;    //Opomba: Indeks pomikamo proti začetku tabele.
  x ← x − (x%2);
  x ← x/2;
}
Sporoči:  b;
```

V jeziku JavaScript program zapišemo takole:

```
x = 13;
b = [0, 0, 0, 0, 0, 0, 0, 0];
i = 7;
while (x > 0) {
  b[i] = x % 2;
  i = i - 1;
  x =  x - (x % 2);
  x =  x / 2;
}
console.log(b);  //Funkcija console.log "zna" izpisati celo tabelo.
```

V tabelo `b` smo na začetku vpisali osem ničel. To je potrebno za vse primere, ko je `x` manjši od 128. Takrat namreč en ali več vodilnih elementov tabele v gornjem programu ne bi dobilo nobene določene vrednosti, medtem ko mora biti njihova vrednost enaka nič.

Ko program zaženemo, dobimo naslednjo sliko:



Po pričakovanju dobimo v konzoli dvojiško vrednost 1101_2. Če bi poskusili zagnati program z vhodno vrednostjo $x = 255$, bi v konzoli dobili izpis 11111111_2, kar je najvišja vrednost, ki jo lahko zapišemo z osmimi biti.

Za vajo razmislite, kakšno vrednost ima spremenljivka `i`, ko se gornji program konča. Ugotovitev preizkusite tako, da vrednost spremenljivke izpišete v konzoli.

Naloga 3.2 V računalništvu za zapisovanje vrednosti, kjer so pomembni posamezni biti, namesto dvojiškega pogosto uporabljamo šestnajstiški zapis. Ker lahko v šestnajstiškem zapisu zapišemo štiri bite z enim samim simbolom, predstavlja šestnajstiški zapis bolj strnjeno, a še vedno pregledno obliko zapisovanja posameznih bitov.

Sami napišite program za pretvarjanje števil v šestnajstiški zapis. Naloga se glasi takole:

> ***Vhod:*** celo število x med 0 in 65 536 (v desetiškem zapisu);
> ***Zahtevani izhod:*** število x, zapisano v šestnajstiškem zapisu (v obliki tabele $\boldsymbol{h}$ s štirimi elementi);

Za zapisovanje vrednosti v šestnajstiškem zapisu potrebujemo 16 različnih števk. Poleg desetiških števk od 0 do 9 uporabimo še črke od A do F, ki po vrsti predstavljajo vrednosti koeficientov od 10 do 15.

Tako kot v dvojiški zapis (glej problem na strani 26) boste število pretvarjali tudi v šestnajstiški zapis z deljenjem z osnovo, ki pa je tokrat 16. Ker ostanki, ki pri tem nastanejo, še niso šestnajstiške števke (namesto števk od A do F boste dobili ostanke od 10 do 15), boste morali dobljene ostanke še pretvoriti v števke. To lahko storite na primer z odčitavanjem iz tabele šestnajstiških števk: Če imate tabelo $\boldsymbol{cifre} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}]$, potem:

$k \leftarrow x\%16$; //Opomba: Ostanek pri deljenju je zaporedna številka števke iz tabele $\boldsymbol{cifre}$.
$h_i \leftarrow cifre_k$; //Opomba: k-to števko iz tabele $\boldsymbol{cifre}$ vstavimo na i-to mesto izhodne tabele.

Opomba: V jeziku JavaScript za podatke ne moremo uporabljati črk, ker jih ne bi mogli ločiti od običajnih spremenljivk. Zato moramo črke, ki predstavljajo konstantne vrednosti, postaviti v enojne navednice:

```
cifre = [0, 1, ..., 9, 'A', 'B', 'C', 'D', 'E', 'F' ];
```

3.4 Kontrolna vsota

Kontrolna vsota (angl. checksum) je podatek, ki ga z namenom odkrivanja napak dodamo k večjemu bloku podatkov, kadar te podatke prenašamo ali shranjujemo. Na primer, zadnja števka 13-mestne enotne matične številke občana (EMŠO) predstavlja kontrolno vsoto[2]. Kontrolna vsota je določena tako, da mora biti vsota vseh 13 števk EMŠO, pomnoženih z določenimi koeficienti, deljiva z 11. Če vsota ni deljiva z 11, potem vemo, da je pri prepisovanju ali prenosu številke prišlo do napake.

Algoritem za določanje kontrolne vsote iz prvih 12 števk EMŠO je takšen:

```
Vhod:  tabela emso, ki vsebuje 12 števk EMŠO
       brez kontrolne vsote;
Izhod:  kontrolna vsota za podani EMŠO (kontrola);
kontrola ← 0;
kontrola ← kontrola + 7 × emso_0;
```

[2] V prvih 12 števkah EMŠO so zapisani: datum rojstva, območje rojstnega kraja, spol in zaporedna številka osebe.

$kontrola \leftarrow kontrola + 6 \times emso_1$;
$kontrola \leftarrow kontrola + 5 \times emso_2$;
$kontrola \leftarrow kontrola + 4 \times emso_3$;
$kontrola \leftarrow kontrola + 3 \times emso_4$;
$kontrola \leftarrow kontrola + 2 \times emso_5$;
$kontrola \leftarrow kontrola + 7 \times emso_6$;
$kontrola \leftarrow kontrola + 6 \times emso_7$;
$kontrola \leftarrow kontrola + 5 \times emso_8$;
$kontrola \leftarrow kontrola + 4 \times emso_9$;
$kontrola \leftarrow kontrola + 3 \times emso_{10}$;
$kontrola \leftarrow kontrola + 2 \times emso_{11}$;
$kontrola \leftarrow kontrola \% 11$;
Če je $kontrola > 0$:
{
  $kontrola \leftarrow 11 - kontrola$;
}
Sporoči: $kontrola$;

Če je ostanek pri deljenju enak ena, potem algoritem vrne dvomestno številko in kontrolne vsote ni mogoče določiti. V takem primeru upravljalec, ki določa EMŠO, poveča zaporedno številko osebe in ponovi postopek določanja kontrolne vsote.

V gornjem algoritmu opazimo določeno ponavljanje, ki ga lahko izvedemo na več načinov. Koeficienti, s katerimi množimo posamezne števke EMŠO, se manjšajo od sedem do dve. To lahko dosežemo enostavno tako, da vsakokrat zmanjšamo koeficient za ena in takoj nato preverimo njegovo vrednost. Ko postane koeficient manjši od dve, ga nastavimo nazaj na sedem:

```
emso = [2, 5, 0, 5, 9, 9, 7, 5, 0, 0, 1, 5];
k = 7;
kontrola = 0;
for (i = 0; i < 12; i = i + 1) {
  kontrola = kontrola + k * emso[i];
  k = k - 1;
  if (k < 2) {
    k = 7;
  }
}
kontrola = kontrola % 11;
if (kontrola > 0) {
  kontrola = 11 - kontrola;
}
console.log(kontrola);
```

Algoritem lahko nekoliko poenostavimo, če koeficiente zapišemo v tabelo:

```
emso = [2, 5, 0, 5, 9, 9, 7, 5, 0, 0, 1, 5];
k = [7, 6, 5, 4, 3, 2, 7, 6, 5, 4, 3, 2];
kontrola = 0;
for (i = 0; i < 12; i = i + 1) {
  kontrola = kontrola + k[i] * emso[i];
}
kontrola = kontrola % 11;
if (kontrola > 0) {
  kontrola = 11 - kontrola;
}
console.log(kontrola);
```

Pomemben del načrtovanja algoritmov je tudi načrtovanje podatkov. Na zadnjem primeru smo videli, da lahko z ustrezno izbranimi podatki precej poenostavimo algoritem. Za hranjenje koeficientov smo porabili sicer nekoliko več pomnilnika, vendar tudi koda, ki smo jo lahko na ta način izpustili, zaseda prostor v pomnilniku. Z minimalno matematično akrobacijo pa se lahko znebimo tudi pomožne tabele `k`:

```
emso = [2, 5, 0, 5, 9, 9, 7, 5, 0, 0, 1, 5];
kontrola = 0;
for (i = 0; i < 6; i = i + 1) {
  kontrola = kontrola + (7 - i) * (emso[i] + emso[i + 6]);
}
kontrola = kontrola % 11;
if (kontrola > 0) {
  kontrola = 11 - kontrola;
}
console.log(kontrola);
```

V tem primeru je algoritem verjetno najučinkovitejši kar zadeva hitrost in porabo pomnilnika, je pa za spoznanje zagonetnejši od prejšnje različice. Opozoriti velja, da je preglednost in čitljivost kode včasih pomembnejši faktor od hitrosti in porabe pomnilnika, saj vpliva na zanesljivost in čas, potreben za vzdrževanje kode. O pomembnih načrtovalskih kriterijih, kot so čitljivost, poraba pomnilnika in časovna zahtevnost, bomo več govorili kasneje.

Naloga 3.3 Mednarodna številka bančnega računa oziroma IBAN (angl. International Bank Account Number) je mednarodni standard za prepoznavanje bančnih računov preko državnih meja. Številka IBAN za Slovenijo se začne s kodo SI56, ki ji sledi 15 desetiških števk, od katerih predstavljata zadnji dve kontrolno vsoto. Obstaja splošen algoritem za iskanje napak v številki IBAN:

> preveri dolžino številke IBAN (19 znakov za Slovenijo);
> prestavi prve štiri znake na konec;
> zamenjaj črke z dvomestnimi številkami (A z 10, B z 11 ... Y s 34, Z s 35);
> izračunaj ostanek pri deljenju števila, ki smo ga dobili v prejšnjem koraku, s 97;
> ostanek mora biti enak ena, sicer IBAN vsebuje napako.

Na primer, številko SI56 1920 0123 4567 892 bi preverili takole:

> dolžina: 19 znakov;
> prestavi prve štiri znake na konec: 192001234567892SI56;
> zamenjaj črke s številkami: 19200123456789228 1856;
> ostanek pri deljenju s 97: $19200123456789228 1856 \,\%\, 97 = 1$.

Ker so prvi štirje znaki za Slovenijo vedno enaki, uporabite v programu namesto kode SI56 pretvorjeno kodo 281856. Takole se glasi problem:

> ***Vhod:*** številka IBAN zapisana kot tabela $[281856, ddddddddddddd]$ (vsak znak d predstavlja eno desetiško številko);
> ***Zahtevani izhod:*** opozorilo v primeru napačno vpisane številke IBAN;

Opomba: V programu boste imeli opravka s celimi števili, katerih vrednost presega največjo celoštevilsko vrednost, ki jo podpira jezik JavaScript (t.j. $2^{53} - 1$). Uporabite lahko vgrajeni objekt `BigInt`, ki omogoča računanje s poljubno velikimi celimi števili. Njegova uporaba je enostavna: dovolj je, da dodate črko `n` na konec vsake velike konstantne celoštevilske vrednosti oziroma vsake celoštevilske vrednosti, ki jo boste pri računanju uporabili skupaj z velikimi celimi števili. Na primer:

```
IBAN = [281856n, 192001234567892n];
```

Nalogo skušajte rešiti sami. Če vam ne bo uspelo, si lahko ogledate rešitev na strani 137.

3.5 Čuvaj

V računalniškem programiranju je ***čuvaj*** (angl. sentinel) posebna vrednost, ki sodeluje pri ustavitvenem pogoju v ponavljalnem stavku ali pri rekurzivnih algoritmih. Čuvaj je oblika tako imenovane ***znotrajpasovne*** (angl. in-band) informacije, ki omogoča, da algoritem zazna konec niza podatkov. Uporablja se, kadar ne obstaja tako imenovana ***zunajpasovna*** (angl. out-of-band) informacija o koncu podatkov, kot je na primer velikost niza podatkov.

Na primer, kadarkoli smo uporabljali tabelo, smo za pravilno delovanje algoritma poleg same tabele potrebovali še poseben (zunajpasoven) podatek o njeni velikosti. Ta je bil podan z ločeno spremenljivko ali celo konstanto. Takšno ločevanje podatkov lahko privede do napak, ki jih je težko odkriti. Lahko se na primer zgodi, da neki tabeli bodisi odvzamemo bodisi dodamo element, pozabimo pa popraviti vrednost (ločene) spremenljivke, ki hrani velikost te tabele. Kadar je tabel veliko, lahko pomotoma povežemo spremenljivko, ki hrani velikost tabele, z napačno tabelo.

V določenih primerih je namesto posebne spremenljivke, ki hrani velikost tabele, smiselno uporabiti čuvaja. Na primer, če imamo tabelo naravnih števil, lahko za čuvaja uporabimo ničlo ali katerokoli negativno vrednost. Ker te vrednosti v kontekstu naravnih števil niso veljavni podatki, lahko s katero od njih označimo konec tabele:

```
tabela = [96, 2, 10, 88, 42, 3, 0];  //Vrednost 0 predstavlja čuvaja.
```

Gornjo tabelo lahko zdaj uporabimo v programu takole:

```
for (i = 0; tabela[i] != 0; i = i + 1) {
  console.log(tabela[i] * 2);
}
```

Program bo izpisal vse elemente tabele, pomnožene z dve. Izpisal pa ne bo čuvaja, s katerim zgolj preverjamo, ali smo dosegli konec tabele.

S spretno uporabo čuvaja je včasih mogoče poenostaviti algoritem. Kot primer si poglejmo naslednji problem iskanja elementa v tabeli:

Vhod:
tabela $\boldsymbol{t}$ z n celoštevilskimi elementi;
celoštevilska vrednost a;

Zahtevani izhod:
zaporedna številka prvega elementa tabele $\boldsymbol{t}$, ki je enak vrednosti spremenljivke a; -1, če takega elementa ni;

Nalogo rešimo preprosto tako, da vsak element iz tabele $\boldsymbol{t}$ po vrsti primerjamo s spremenljivko a. Če in ko najdemo prvi tak element, si zapomnimo njegovo zaporedno številko. Naslednji algoritem predstavlja eno od možnih rešitev:

```
Vhod:  tabela t z n celoštevilskimi elementi;
       celoštevilska vrednost a;
Izhod:  indeks prvega elementa t, ki je enak a;
        −1, če takega elementa ni;
indeks ← −1;
Ponovi za vsak i od 0 do n − 1 s korakom 1:
{
  Če je t_i enak a in indeks enak −1:   //Opomba: Če odstranimo drugi del pogoja (za
  {                                     //besedo in), vrne algoritem indeks zadnjega
    indeks ← i;                         //elementa, ki je enak spremenljivki a.
  }
}
Sporoči:  indeks;
```

Program v jeziku JavaScript je videti takole:

```
t = [9, -4, 7, -4, 6, 1];
n = 6;
a = -4;
indeks = -1;
for (i = 0; i < n; i = i + 1) {
  if (t[i] == a && indeks == -1) {
    indeks = i;
  }
}
console.log(indeks);
```

Drugi del pogoja v stavku `if` (t.j. `indeks == -1`) je potreben zato, da se po tem, ko smo našli prvi element, ki je enak spremenljivki `a`, indeks ne spreminja več. Če ta pogoj odstranimo, bo program vrnil indeks zadnjega, in ne več prvega najdenega elementa.

Program lahko naredimo učinkovitejši (čeprav mogoče nekoliko manj pregleden), če namesto drugega dela pogoja v stavku `if` vrednost spremenljivke `i` nastavimo na `n`, takoj ko smo našli iskani element. S tem predčasno prekinemo[3] izvajanje stavka `for`:

```
t = [9, -4, 7, -4, 6, 1];
n = 6;
a = -4;
indeks = -1;
for (i = 0; i < n; i = i + 1) {
  if (t[i] == a) {
    indeks = i;
    i = n;
  }
}
```

[3] V večini programskih jezikov – kakor tudi v jeziku JavaScript – lahko za predčasno prekinitev izvajanja ponavljalnega stavka uporabimo stavek `break`, ki pa ga v tem učbeniku ne bomo obravnavali.

```
console.log(indeks);
```

V gornjem programu moramo ob vsakem obhodu zanke (t.j. v vsaki ***iteraciji***) izvesti vsaj dva preizkusa: ali smo našli element (t.j. `t[i] == a`) ter ali še nismo na koncu tabele (t.j. `i < n`).

Z uporabo čuvaja lahko opravimo oba preizkusa z enim samim pogojnim izrazom. Za čuvaja uporabimo kar vrednost, ki jo iščemo. Če je iskana vrednost že v tabeli, potem iskanje prekinemo takoj, ko jo najdemo. Če iskane vrednosti ni v tabeli, potem se bo iskanje prekinilo najkasneje na koncu tabele, kamor postavimo čuvaja. Zamisel prikazuje naslednji algoritem:

```
Vhod:  tabela t z n celoštevilskimi elementi;
       celoštevilska vrednost a;
Izhod:  indeks prvega elementa t, ki je enak a;
        −1, če takega elementa ni;
Dodaj a na konec t;    //Opomba: Na konec tabele postavimo čuvaja.
i ← 0;
Ponavljaj, dokler je t_i različen od a:
{
  i ← i + 1;
}
indeks ← −1;
Če je i < n:    //Opomba: Čuvaj je element z zaporedno številko n. Če je
{               //i manjši od n, smo našli pravi element.
  indeks ← i;
}
Sporoči:  indeks;
```

In še program v jeziku JavaScript:

```
t = [9, -4, 7, -4, 6, 1];
n = 6;
a = -4;
t[n] = a;
i = 0;
while (t[i] != a) {
  i = i + 1;
}
indeks = -1;
if (i < n) {
  indeks = i;
}
console.log(indeks);
```

Poglejmo si še en primer, kjer čuvaj poenostavi algoritem. Naloga se glasi takole:

> ***Vhod:*** dve tabeli naravnih števil t in u, zaključeni s čuvajem z vrednostjo nič;
> ***Zahtevani izhod:*** sporočilo o tem, ali sta tabeli enaki (t.j. »Tabeli sta enaki.« oz. »Tabeli sta različni.«);

Nalogo rešimo tako, da primerjamo po dva in dva (istoležna) elementa obeh tabel. Začnemo pri prvem elementu in nadaljujemo, dokler bodisi ne najdemo dveh različnih elementov bodisi ne naletimo na enega od obeh čuvajev:

```
Vhod:  tabeli naravnih števil t in u, zaključeni s čuvajem (0);
Izhod:  sporočilo o tem, ali sta tabeli enaki;
Ponovi za vsak i od 0 do konca vsaj ene od tabel
in dokler sta t_i in u_i enaka, s korakom 1:
{
}
Če sta t_i in u_i enaka:
{
  Sporoči:  "Tabeli sta enaki.";
}
sicer:
{
  Sporoči:  "Tabeli sta različni.";
}
```

Opazimo, da med zavitima oklepajema ponavljalnega stavka ni ničesar. V resnici nam tudi ničesar ni treba storiti, saj je vse delo opravljeno že pred tem – dovolj je, da pridemo do dveh različnih elementov ali do konca vsaj ene od obeh tabel.

Napišimo zdaj še program. Pogoj `do konca vsaj ene od tabel` zapišemo v stavku `for` kot `t[i] != 0 && u[i] != 0`. Na ta način se zanka ustavi, brž ko dosežemo prvega od obeh čuvajev (ali oba, če sta tabeli enaki). Zanka se mora ustaviti tudi, če naletimo na dva elementa, ki sta različna (t.j. mora se ponavljati, dokler sta elementa enaka). Slednje dosežemo z uporabo pogoja `t[i] == u[i]`. Takole je videti dokončan program:

```
t = [9, 4, 6, 1, 0];
u = [9, 4, 6, 1, 0];
for (i = 0; t[i] == u[i] && t[i] != 0 && u[i] != 0; i = i + 1) {}
if (t[i] == u[i]) {
  console.log("Tabeli sta enaki.");
}
else {
  console.log("Tabeli sta različni.");
}
```

Gornja zanka `for` se bo ustavila, brž ko ne bo izpolnjen vsaj eden od treh pogojev, ki jih povezujeta logična operatorja IN. Imamo več možnosti: če se zanka ustavi, ker ni več izpolnjen pogoj `t[i] == u[i]`, potem sta tabeli različni. Če pa se zanka ustavi zato, ker ni izpolnjen kateri od pogojev `t[i] != 0` ali `u[i] != 0` (t.j. naleteli smo na vsaj enega čuvaja), potem sta lahko tabeli enaki ali različni. Vendar če sta tabeli različni, potem zanko ustavi že prvi pogoj (t.j. `t[i] == u[i]`). Zato nas zanima le še situacija, ko sta tabeli enaki. To pa pomeni, da je dovolj, če preverjamo le enega od obeh čuvajev. Stavek `for` v gornjem programu lahko zato poenostavimo takole:

```
for (i = 0; t[i] == u[i] && t[i] != 0; i = i + 1) {}
```

Stavek `if...else`, ki sledi, znova primerja oba elementa, da ugotovi, na kakšen način se je predhodna zanka ustavila. Če sta elementa enaka, potem smo prišli do čuvajev in sta potemtakem enaki tudi tabeli. Če pa sta elementa različna, sta zagotovo različni tudi tabeli.

Namesto s čuvajem bi seveda lahko označili velikost vsake tabele s posebno spremenljivko. To pa bi nekoliko zapletlo algoritem, saj v primeru vsaj ene prazne tabele[4] ne bi mogli med seboj primerjati vrednosti elementov. Algoritem bi tako razpadel na dva dela: en del bi moral še vedno med seboj primerjati vrednosti posameznih elementov, drugi del pa bi moral primerjati velikosti tabel, če bi bila vsaj ena od tabel prazna.

De Morganova zakona Povezovanje dveh ali več logičnih trditev z operatorjema IN in ALI v eno samo logično trditev je lahko begajoče. Na primer, izjavo *konec vsaj ene od tabel*, ki smo jo uporabili v zadnjem algoritmu, lahko nadomestimo z enakovredno logično izjavo *konec prve tabele* ALI *konec druge tabele*. Ta druga izjava povezuje dve neodvisni izjavi v eno samo z operatorjem ALI, mi pa smo v dejanskem programu uporabili logični operator IN (`t[i] != 0 && u[i] != 0`). Zakaj? Spomnimo se, da se ob izpolnjenem pogoju zanka ponavlja, ko pa pogoj ni več izpolnjen, se zanka konča. Zato moramo za ponavljanje zanke zadnjo izjavo v resnici zanikati: NI (*konec prve tabele* ALI *konec druge tabele*). Pomembno je, da se zanikanje v zadnji izjavi nanaša na celotno izjavo (t.j. zanikali smo izjavo *konec prve tabele* ALI *konec druge tabele*), in ne na vsako od obeh izjav posebej. Dve pravili za zanikanje takšnih sestavljenih izjav je že sredi 19. stoletja zapisal britanski matematik Augustus De Morgan:

$$\neg(p \wedge q) \iff (\neg p) \vee (\neg q) \tag{3.2}$$
$$\neg(p \vee q) \iff (\neg p) \wedge (\neg q). \tag{3.3}$$

Tu sta p in q dve neodvisni logični trditvi. Operator $\neg$ predstavlja logično negacijo, operatorja $\wedge$ in $\vee$ pa logično konjunkcijo (IN) in disjunkcijo (ALI). Operator $\iff$ pomeni, da sta leva in desna stran popolnoma enakovredni. Če povzamemo, de Morganova zakona pravita, da sestavljeno izjavo zanikamo tako, da zanikamo vsako od obeh izjav posebej, poleg tega pa zamenjamo konjunkcijo z disjunkcijo ali obratno.

Poglejmo, kako to deluje v praksi. Izjavo *konec prve tabele* ALI *konec druge tabele* zapišemo takole:

```
t[i] == 0 || u[i] == 0
```

To izjavo moramo za uporabo v stavku `for` zanikati. V skladu s pravilom (3.3) zanikamo celotno izjavo tako, da zanikamo vsako od obeh izjav posebej, operator ALI (`||`) pa zamenjamo z operatorjem IN (`&&`):

```
t[i] != 0 && u[i] != 0
```

3.6 Dvorazsežnostna tabela

Dvorazsežnostno tabelo z n vrsticami in m stolpci predstavimo kot tabelo n enorazsežnostnih tabel z m elementi:

$$\boldsymbol{t} = [\boldsymbol{t}_0, \boldsymbol{t}_1, \cdots, \boldsymbol{t}_{n-1}], \tag{3.4}$$

[4]Prazna tabela ne vsebuje nobenega elementa. Kadar pa uporabimo čuvaja, vsebuje tudi prazna tabela en element, saj še vedno vsebuje čuvaja.

kjer je $\boldsymbol{t}_i$ i-ta vrstica tabele $\boldsymbol{t}$:

$$\boldsymbol{t}_i = [t_{i,0}, t_{i,1}, \cdots, t_{i,m-1}], i = 0, 1, \cdots, n-1. \tag{3.5}$$

Element tabele $\boldsymbol{t}$ v i-ti vrstici in j-tem stolpcu označimo s $t_{i,j}$, oziroma v programskem jeziku JavaScript s `t[i][j]`.

Za boljšo predstavo o načinu dela z dvorazsežnostnimi tabelami si poglejmo preprost primer, ki sešteje vse elemente takšne tabele:

> ***Vhod:*** dvorazsežnostna tabela realnih števil $\boldsymbol{t}$ z n vrsticami in m stolpci;
> ***Zahtevani izhod:*** vsota vseh elementov vhodne tabele;

Problem rešimo z naslednjim algoritmom:

```
Vhod: dvorazsežnostna tabela t, število vrstic (n) in stolpcev (m);
Izhod: vsota vseh elementov vhodne tabele (vsota);
vsota ← 0;
Ponovi za vsak i od 0 do n − 1 s korakom 1:        //Opomba: za vsako vrstico
{
  Ponovi za vsak j od 0 do m − 1 s korakom 1:      //Opomba: za vsak stolpec
  {
    vsota ← vsota + t_{i,j};
  }
}
Sporoči: vsota;
```

Opazimo, da potrebujemo za rešitev problema dva ponavljalna stavka, in sicer enega znotraj drugega: v vsaki vrstici moramo obiskati vse stolpce. Povedano drugače, vrstica kode $vsota \leftarrow vsota + t_{i,j}$ se bo ponovila natanko enkrat za vsako možno kombinacijo indeksov i in j.

Iz gornjega algoritma hitro pridemo do naslednjega programa:

```
t = [[1.6, -2], [7, -2], [5.1, 0.4]];
n = 3;
m = 2;
vsota = 0;
for (i = 0; i < n; i = i + 1) {
  for (j = 0; j < m; j = j + 1) {
    vsota = vsota + t[i][j];
  }
}
console.log(vsota);
```

Ko program zaženemo, dobimo pričakovano vsoto 10,1.

Naloga 3.4 V zadnjem primeru smo uporabili pravokotno tabelo, v kateri so bile vse vrstice enako dolge. Čeprav je to najbolj običajno, pa ni nujno, da je dvorazsežnostna tabela pravokotne oblike. Na primer, z uporabo čuvaja lahko dosežemo, da imajo vrstice znotraj ene in iste tabele poljubne dolžine. Za vajo napišite program, ki reši naslednji problem:

> ***Vhod:*** dvorazsežnostna tabela pozitivnih števil $\boldsymbol{t}$ z ničlami na koncu vsake vrstice ter samó ničlo v zadnji vrstici (ničle predstavljajo čuvaje);

Zahtevani izhod: vsota vseh elementov vhodne tabele;

Na primer, za vhodno tabelo

$$\begin{aligned} \boldsymbol{t} = [&[1{,}3,\ 0{,}1,\ 4{,}6,\ 3{,}8,\ 0],\\ &[0{,}3,\ 1{,}1,\ 0],\\ &[1{,}5,\ 2{,}5,\ 0{,}4,\ 0],\\ &[0]] \end{aligned}$$

naj program vrne vsoto $15{,}6$.

Nalogo poskusite rešiti sami, če pa boste imeli težave, si pomagajte z rešitvijo na strani 138.

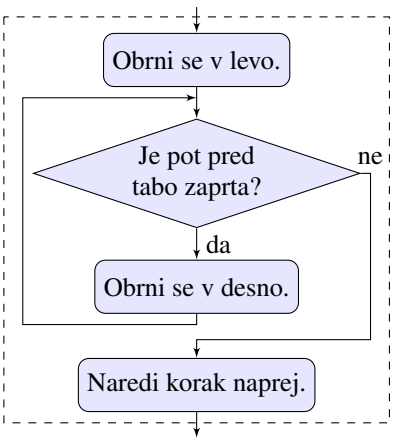
Opomba: Ne pozabite, da morate pri realnih konstantah v programu namesto decimalnih vejic uporabiti decimalne pike.

Kot zadnji primer v tem poglavju bomo razvili algoritem, ki najde pot skozi blodnjak. Algoritem je sila preprost:

ko vstopiš v blodnjak, se dotakni stene z levo (ali desno) roko;
potem pojdi skozi blodnjak, ne da bi roko kdajkoli odmaknil od stene.

Če začnemo pri kateremkoli vhodu v blodnjak, potem bomo s tem algoritmom zagotovo našli pot skozenj, če ta le obstaja. Če pot ne obstaja, potem se bomo prej ali slej spet znašli na vhodu.

Algoritem za iskanje poti skozi blodnjak je sicer preprost, še vedno pa ostaja vprašanje, kako ga zapisati na način, da ga bo razumel računalnik. Navodilo, da se moramo z levo roko ves čas dotikati stene, lahko izrazimo tudi drugače. Naslednji diagram prikazuje navodilo, kako napraviti en korak skozi blodnjak:



Iz algoritma vidimo, da vedno zavijemo v levo, če se to le da. Če to ni mogoče, potem se obračamo v desno, dokler pot pred nami ni prosta. Na ta način nam levice nikoli ni treba odmakniti od stene razen v primeru, ko se nam proti levi odpre pot (takoj zatem ko naredimo korak naprej). Takrat pa se še vedno lahko dotikamo vogala levo za nami, okrog katerega bomo zavili v naslednjem koraku.

Gornji algoritem ponavljamo, dokler ne pridemo iz blodnjaka. Zapišimo zdaj celoten algoritem v psevdo jeziku:

```
Vhod:  blodnjak in začetni položaj (pol) ter smer;
Izhod:  blodnjak z vpisano potjo;
Ponavljaj, dokler si v blodnjaku:
{
  Obrni se v levo (spremeni smer);
  Ponavljaj, dokler je pot pred tabo zaprta:
  {
    Obrni se v desno (spremeni smer);
  }
  Označi trenutni položaj v blodnjaku;
  Naredi korak naprej (spremeni pol);
}
```

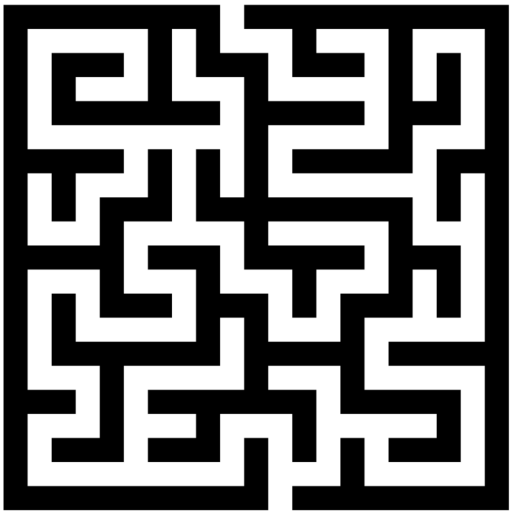
Vsi podatki, s katerimi smo delali doslej, so bile številke ali črke, ki smo jih zapisovali na običajen način (črke smo morali postaviti v enojne navednice, da se v programu ločijo od imen spremenljivk). Blodnjak je v resnici tudi podatek, vendar oblika njegovega zapisa ni več tako očitna. Za zapis blodnjaka si bomo pomagali s splošnimi načeli ***kodiranja***, ki jih uporabljamo, kadar želimo zapisati kakšno informacijo, ki sama po sebi ni številske narave[5]. V računalništvu in informatiki predstavlja kodiranje zbirko pravil, na podlagi katerih lahko kakršnokoli informacijo (npr. besedo, zvok, sliko, gib ipd.) zapišemo na način, primeren za prenos, shranjevanje ali obdelavo.

Naš blodnjak bomo zakodirali kot dvorazsežnostno tabelo z dvema različnima vrednostma elementov. Odločimo se, da enka predstavlja steno, ničla pa prosto pot. Po blodnjaku se bomo pomikali bodisi vodoravno bodisi navpično in vsak korak nas bo peljal z enega prostega elementa (z vrednostjo nič) na enega od njegovih štirih neposrednih sosedov, ki pa mora biti prav tako prost (imeti vrednost nič). Okrog in okrog blodnjaka postavimo še čuvaje, ki jih predstavimo z elementi z vrednostjo dve:

$$
\begin{aligned}
\boldsymbol{blodnjak} = [&[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2],\\
&[2,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2],\\
&[2,1,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,1,2],\\
&[2,1,0,1,1,1,0,1,0,1,1,1,0,1,1,1,0,1,0,1,0,1,0,1,0,1,2],\\
&[2,1,0,1,0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,1,0,1,2],\\
&[2,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,0,1,0,1,2],\\
&[2,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,1,2],\\
&[2,1,1,1,1,1,1,0,1,0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,1,1,2],\\
&[2,1,0,1,0,0,0,1,0,1,0,1,0,1,0,0,0,0,0,0,0,1,0,1,0,1,2],\\
&[2,1,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0,1,0,1,2],\\
&[2,1,0,0,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,1,2],\\
&[2,1,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,2],\\
&[2,1,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,2],\\
&[2,1,0,1,0,1,1,1,0,1,1,1,0,1,0,1,1,1,0,1,0,1,0,1,0,1,2],\\
&[2,1,0,1,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,2],\\
&[2,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,2],\\
&[2,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,1,2],\\
&[2,1,1,1,1,1,0,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,0,1,2],\\
&[2,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,1,0,1,0,1,2],\\
&[2,1,0,1,1,1,0,1,1,1,0,1,0,1,0,1,0,1,1,1,0,1,0,1,0,1,2],\\
&[2,1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,2],\\
&[2,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,2],\\
&[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]]
\end{aligned}
$$

[5] V digitalnem računalniku v resnici kodiramo tudi številke, in sicer z nizi ničel in enk. Na nivoju, ki ga obravnavamo v tem učbeniku, pa je številka najbolj elementaren način zapisovanja podatkov.

Za boljšo predstavo si blodnjak še narišimo. Vsak element z vrednostjo ena prikažemo s črnim, vsak element z vrednostjo nič pa z belim kvadratkom:



Poleg blodnjaka potrebujemo še podatek o našem položaju v blodnjaku ($\boldsymbol{pol}$) in smeri iskanja ($\boldsymbol{smer}$). Oba podatka bomo zakodirali v obliki vektorjev z dvema elementoma, saj gre za položaj in smer na dvorazsežnostni ravnini. Za položaj lahko vzamemo kar indeks vrstice in stolpca, kjer se trenutno nahajamo. Smer pa zapišemo kot spremembo položaja (indeksa vrstice ali stolpca), ki jo opravimo z enim korakom, ko se premaknemo na sosednji element blodnjaka. S tako kodiranimi podatki izvedemo premik po blodnjaku za en korak takole:

$[pol_0, pol_1] \leftarrow [pol_0, pol_1] + [smer_0, smer_1]$ //Opomba: Prvemu elementu položaja prištejemo
//prvi element smeri in drugemu elementu položaja prištejemo
//drugi element smeri.

Ker se bomo pomikali le vodoravno ali navpično, bo en element vektorja $\boldsymbol{smer}$ vedno enak nič, drugi pa ena oziroma -1. V blodnjak bomo vstopili pri zgornjem vhodu, ki se nahaja v vrstici z indeksom ena in stolpcu z indeksom deset, zato moramo nastaviti začetni položaj na $\boldsymbol{pol} = [1, 10]$. Ker smo v tem primeru ob vstopu v blodnjak obrnjeni navzdol, nastavimo začetno smer na $\boldsymbol{smer} = [1, 0]$. Takšna smer poveča vrstico položaja za ena, stolpec položaja pa ostane nespremenjen.

Glede na vsakokratno smer gibanja ima lahko vektor $\boldsymbol{smer}$ katerokoli od naslednjih štirih vrednosti:

$\boldsymbol{smer} = [-1, 0]$ //Opomba: gor
$\boldsymbol{smer} = [0, -1]$ //Opomba: levo
$\boldsymbol{smer} = [1, 0]$ //Opomba: dol
$\boldsymbol{smer} = [0, 1]$ //Opomba: desno

Zdaj moramo ugotoviti samo še to, kako se bomo v blodnjaku zasukali v levo (nasproti urinemu kazalcu) oziroma desno (v smeri urinega kazalca). V ta namen lahko vektor smeri množimo z matriko vrtenja, vendar se bomo v našem primeru temu izognili. Ker imamo

opravka le z obrati za $90°$, lahko iz gornjega zaporedja štirih možnih vrednosti smeri hitro izluščimo naslednja dva algoritma:

```
Zasuk v levo (nasproti urinemu kazalcu):
```
$[smer_0, smer_1] \leftarrow [-smer_1, smer_0]$ //Opomba: Drugi element množimo z -1 in //zamenjamo vrednosti elementov.

```
Zasuk v desno (v smeri urinega kazalca):
```
$[smer_0, smer_1] \leftarrow [smer_1, -smer_0]$ //Opomba: Prvi element množimo z -1 in //zamenjamo vrednosti elementov.

Prvi od gornjih dveh zapisov pomeni, da dobi prvi element tabele $\boldsymbol{smer}$ vrednost, ki jo je imel prej njen drugi element, pomnožen z -1, drugi element tabele $\boldsymbol{smer}$ pa dobi vrednost, ki jo je imel prej njen prvi element. podobno razumemo drugi zapis: Prvi element tabele $\boldsymbol{smer}$ dobi vrednost, ki jo je imel prej njen drugi element, drugi element tabele $\boldsymbol{smer}$ pa dobi vrednost, ki jo je imel prej njen prvi element, pomnožen z -1.

Sestavimo zdaj vse skupaj v naslednji program:

```
blodnjak = [
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
[2, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2],
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
];
pol = [1, 10];
smer = [1, 0];
while (blodnjak[pol[0]][pol[1]] != 2) {
  //Obrat v levo:
  zacasno = smer[0];
  smer[0] = -smer[1];
  smer[1] = zacasno;
  while (blodnjak[pol[0] + smer[0]][pol[1] + smer[1]] == 1) {
    //Obrat v desno:
    zacasno = -smer[0];
    smer[0] = smer[1];
    smer[1] = zacasno;
  }
```

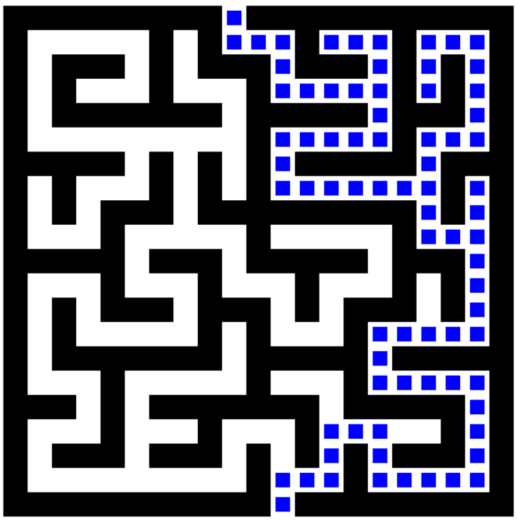
```
    //Vpišemo položaj v blodnjak:
    blodnjak[pol[0]][pol[1]] = 3;
    //Korak naprej:
    pol[0] = pol[0] + smer[0];
    pol[1] = pol[1] + smer[1];
  }
```

V pogojih obeh stavkov `while` smo elementa vektorja `pol` uporabili kot indeksa tabele `blodnjak`. Na primer, če je `pol = [1, 10]`, potem se pogoj v prvem stavku `while` bere kot `blodnjak[1][10] != 2`. Hkrati lahko tu vidimo, kako smo uporabili čuvaja za enostavno preverjanje, ali smo še vedno v blodnjaku. Brez čuvajev bi bil pogoj v tem prvem stavku `while` precej bolj zapleten, saj bi morali preveriti obe meji (gornjo in spodnjo) tako za številko stolpca kot tudi vrstice. Ker z iskanjem začnemo na robu blodnjaka, bi bilo treba ugotoviti še to, ali smo na robu zato, ker roba sploh še nismo zapustili (pred prvim korakom iskanja), ali pa smo do roba prišli in moramo algoritem ustaviti.

Naj poudarimo še, da v pogoju drugega stavka `while` v resnici ne izvedemo premika[6]. Izraza `pol[0] + smer[0]` in `pol[1] + smer[1]` zgolj vrneta številki vrstice in stolpca, kamor ***bi se premaknili***, če bi premik izvedli. Dejanski premik se izvede v čisto zadnjih dveh vrsticah gornje kode.

Trenutni položaj v blodnjaku označimo z vrednostjo tri (to vrednost vpišemo v blodnjak, tik preden se premaknemo naprej), zato bo na koncu veriga trojk v blodnjaku predstavljala najdeno pot.

Ko program zaženemo, le-ta najde pot, ki je grafično prikazana na naslednji sliki:



Iz slike se lepo vidi, da je program preiskal tudi vse slepe odcepe, ki se na poti odprejo proti levi. Slepih odcepov, ki se odprejo proti desni, ni preiskal.

V dodatku C boste našli kodo, ki vam omogoča, da blodnjak in najdeno pot v brskalniku prikažete grafično. Za vajo lahko preizkusite, kaj se zgodi, če bi pot iskali v desno namesto v levo.

[6]Edini operator, ki lahko dejansko spremeni vrednost spremenljivke, je priredilni operator (=).

3.7 Naloge

Sledi še nekaj nalog za samostojno reševanje:

Naloga 3.5 Napišite program, ki reši naslednji problem:

> ***Vhod:*** tabela $\boldsymbol{t}$ z n realnimi elementi;
> ***Zahtevani izhod:*** razlika med največjim in najmanjšim elementom vhodne tabele;

Na primer, za vhodno tabelo $\boldsymbol{t} = [-6,\ 33,\ 12{,}1,\ -7{,}9]$ naj program izpiše vrednost $40{,}9$ (t.j. $33 - (-7{,}9)$).

Naloga 3.6 Napišite program, ki reši naslednji problem:

> ***Vhod:*** tabela $\boldsymbol{t}$ z n celoštevilskimi elementi;
> ***Zahtevani izhod:*** ista tabela z obrnjenim vrstnim redom elementov;

Na primer, program naj vhodno tabelo $\boldsymbol{t} = [8, 14, -4, 2, 77]$ pretvori v tabelo $\boldsymbol{t} = [77, 2, -4, 14, 8]$.

Naloga 3.7 Napišite program, ki reši naslednji problem:

> ***Vhod:*** tabela naravnih števil $\boldsymbol{t}$, kjer predstavlja prvi element v tabeli število vseh elementov tabele brez prvega elementa;
> ***Zahtevani izhod:*** tabeli $\boldsymbol{lihi}$ in $\boldsymbol{sodi}$, v katerih so (poleg podatka o velikosti tabele) shranjeni vsi lihi oziroma sodi elementi tabele $\boldsymbol{t}$;

Na primer, program naj iz tabele $\boldsymbol{t} = [5, 2, 3, 55, 12, 13]$ ustvari tabeli $\boldsymbol{lihi} = [3, 3, 55, 13]$ in $\boldsymbol{sodi} = [2, 2, 12]$. Podobno naj iz tabele $\boldsymbol{t} = [4, 9, 11, 43, 1]$ ustvari tabeli $\boldsymbol{lihi} = [4, 9, 11, 43, 1]$ in $\boldsymbol{sodi} = [0]$.

Naloga 3.8 Napišite program, ki reši naslednji problem:

> ***Vhod:*** tabela $\boldsymbol{t}$ z n elementi ter dva indeksa i_{start} in i_{stop}, $0 \leq i_{\text{start}} < i_{\text{stop}} \leq n$;
> ***Zahtevani izhod:*** tabela $\boldsymbol{u}$, ki vsebuje vse elemente tabele $\boldsymbol{t}$ z zaporednimi številkami med vključno i_{start} in $i_{\text{stop}} - 1$;

Na primer, za vhodne podatke $\boldsymbol{t} = [9, 0, 3, 5, 1, 8, 7]$, $i_{\text{start}} = 1$ in $i_{\text{stop}} = 4$ naj program vrne tabelo $\boldsymbol{u} = [0, 3, 5]$.

Pomoč: Program začnite s prazno tabelo ($\boldsymbol{u} = [\,]$) in postopoma dodajajte elemente ($u_i = t_j$).

Naloga 3.9 Ta naloga je razširitev Naloge 2.8 na strani 18, kjer je bilo treba preveriti, ali je n praštevilo. Zdaj napišite program za naslednjo nalogo:

Vhod: naravno število $m > 1$;
Zahtevani izhod: tabela vseh praštevil, ki so manjša ali enaka m;

Program poskusite napisati tako, da bo za rešitev potrebno ***čim manj deljenj*** (t.j. računanj ostankov pri deljenju). Pri tem upoštevajte naslednjo ugotovitev: če n ni deljiv z nobenim od ***praštevil***, ki so manjša od n, potem je n praštevilo. Namreč, če obstaja delitelj števila n (na primer k), ki ni praštevilo, potem mora biti n deljiv tudi z vsemi prafaktorji števila k. Zato je dovolj, da preverite deljivost le s prafaktorji števila k, ne pa tudi s k.

Poleg tega je dovolj, da preverite deljivost le s števili, ki so ***manjša ali enaka*** $\sqrt{n}$. Namreč, če obstaja delitelj števila n, ki je večji od $\sqrt{n}$, potem mora nujno obstajati tudi delitelj, ki je manjši od $\sqrt{n}$. Torej, če ne najdete delitelja, ki bi bil manjši ali enak $\sqrt{n}$, potem delitelj ne obstaja.

Število deljenj lahko bistveno zmanjšate tudi s tem, da prekinete preverjanje deljivosti števila n takoj, ko odkrijete prvega delitelja.

Naloga 3.10 Napišite program, ki reši naslednji problem:

Vhod: tabela $\boldsymbol{t}$ z n elementi;
Zahtevani izhod: ista tabela, spremenjena tako, da je vsakemu elementu prišteta vsota vseh elementov pred njim;

Na primer, program naj vhodno tabelo $\boldsymbol{t} = [4, 1, 6, -2, 1, 3, 4, -5]$ pretvori v tabelo $\boldsymbol{t} = [4, 5, 11, 9, 10, 13, 17, 12]$.

Naloga 3.11 Napišite program, ki reši naslednji problem:

Vhod: vektorja $\boldsymbol{x}$ in $\boldsymbol{y}$ z n elementi;
Zahtevani izhod: skalarni produkt (angl. dot product) vektorjev $\boldsymbol{x}$ in $\boldsymbol{y}$;

Skalarni produkt dveh vektorjev $\boldsymbol{x}$ in $\boldsymbol{y}$ z n elementi določa naslednja enačba:

$$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=0}^{n-1} x_i \, y_i = x_0 \, y_0 + x_1 \, y_1 + \cdots + x_{n-1} \, y_{n-1}. \tag{3.6}$$

Naloga 3.12 Napišite program, ki reši naslednji problem:

Vhod: nenegativno celo število n;
Zahtevani izhod: enorazsežnostna tabela $\boldsymbol{p}$, ki predstavlja n-to vrstico Pascalovega trikotnika;

V matematiki je Pascalov trikotnik trikotna tabela binomskih koeficientov, ki nastopajo pri razvoju n-te potence dvočlenika $(a + b)$ (t.i. binomski izrek). Na primer, potenco $(a + b)^4$ po binomskem izreku razvijemo takole:

$$(a + b)^4 = 1\,a^4 + 4\,a^3\,b + 6\,a^2\,b^2 + 4\,a\,b^3 + 1\,b^4. \tag{3.7}$$

Koeficiente 1, 4, 6, 4 in 1 najdemo v vrstici $n = 4$ Pascalovega trikotnika:

```
(n = 0)              1
(n = 1)            1   1
(n = 2)          1   2   1
(n = 3)        1   3   3   1
(n = 4)      1   4   6   4   1
(n = 5)    1   5  10  10   5   1
```

Pomoč: Vsak notranji element Pascalovega trikotnika dobimo kot vsoto dveh elementov v vrstici neposredno nad njim. Program začnite s tabelo z enim samim elementom z vrednostjo ena ($\boldsymbol{p} = [1]$). Vsako naslednjo vrstico izračunajte tako, da seštevate po dva in dva elementa in desnega od obeh elementov nadomestite z dobljeno vsoto. To počnite od desne proti levi. V zadnjem koraku računanja vsake nove vrstice dodajte na konec tabele $\boldsymbol{p}$ nov element z vrednostjo ena.

Naloga 3.13 Napišite program, ki reši naslednji problem:

> ***Vhod:*** dve tabeli naravnih števil $\boldsymbol{t}$ in $\boldsymbol{u}$, zaključeni s čuvajem z vrednostjo nič;
> ***Zahtevani izhod:*** zaporedna številka prvega elementa tabele $\boldsymbol{t}$, pri katerem se začne niz elementov, ki je enak nizu elementov v tabeli $\boldsymbol{u}$ oziroma enak -1, če takega niza ni;

Na primer, z vhodnima tabelama $\boldsymbol{t} = [1, 5, 8, 1, 4, 2, 0]$ in $\boldsymbol{u} = [1, 4, 0]$ naj program vrne vrednost tri, kajti niz $[1, 4]$ se v tabeli $\boldsymbol{t}$ začne na mestu tri.

Naloga 3.14 Napišite program, ki reši naslednji problem:

> ***Vhod:*** kvadratna tabela $\boldsymbol{k}$ velikosti $n \times n$ elementov;
> ***Zahtevani izhod:*** sporočilo o tem, ali predstavlja $\boldsymbol{k}$ latinski kvadrat;

Latinski kvadrat je dvorazsežnostna tabela z $n \times n$ elementi. Tabela je zapolnjena z n različnimi simboli, od katerih se vsak od njih v vsaki vrstici ali stolpcu pojavi natanko enkrat. Primer latinskega kvadrata velikosti 4×4 je:

$$\begin{aligned}\boldsymbol{k} = [&[2, 1, 4, 3],\\ &[1, 4, 3, 2],\\ &[3, 2, 1, 4],\\ &[4, 3, 2, 1]]\,.\end{aligned}$$

Zanimivost: Obstaja postopek, ki temelji na nizih ortogonalnih latinskih kvadratov, s katerim je moč popravljati napake pri prenosu podatkov, kadar komunikacijo moti

več različnih vrst šuma. Latinski kvadrati se uporabljajo tudi na različnih področjih kombinatorike in načrtovanja eksperimentov. Priljubljena matematična uganka sudoku je prav tako primer latinskega kvadrata.

Naloga 3.15 Pri prenosu dvojiških podatkov se za kontrolno vsoto pogosto uporablja en sam bit: če je število bitov v podatku liho, potem je vrednost kontrolnega bita ena, sicer je njegova vrednost nič. Takšna kontrola je preveč občutljiva na napake (če se pri prenosu pokvari sodo število bitov, napake ne bomo zaznali). Zato vsake toliko časa dodamo še en kontrolni podatek, ki vsebuje vrstico bitov, ki hranijo podatek o sodosti oziroma lihosti bitov v »stolpcih«. Odločimo se, da bomo prenašali pakete treh podatkov s po sedmimi biti. Z dodanimi kontrolnimi biti za vrstice in stolpce dobimo dvorazsežnostno tabelo s štirimi vrsticami po osem elementov:

$$\begin{aligned}\boldsymbol{prenos} = [&[0,0,1,1,0,1,0,1],\\ &[1,0,1,1,0,1,0,0],\\ &[0,1,0,1,0,0,1,1],\\ &[1,1,0,1,0,0,1,0]]\,.\end{aligned}$$

Napišite program, ki reši naslednji problem:

Vhod: kvadratna tabela $\boldsymbol{prenos}$ velikosti 4×8 elementov;
Zahtevani izhod: sporočilo o tem, ali tabela vsebuje napako;

Opomba: Tabela zagotovo vsebuje napako, če vsebuje katerikoli stolpec ali katerakoli vrstica liho število enic.

PRAZNA STRAN

4. POGLAVJE

PODPROGRAMI

Resnični programi so navadno precej daljši in bolj zapleteni od programov, ki smo jih srečali doslej. Za gradnjo takšnih programov je zelo pomembno, da lahko določene delne rešitve zapišemo kot ločene kose programov, iz katerih nato sestavljamo bolj in bolj kompleksne sisteme. Takšnim ločenim kosom programov rečemo ***podprogrami*** (ali tudi funkcije, procedure oz. rutine). Podprogrami so v resnici programi, kakršne smo pisali doslej, le da moramo podprogramu dodeliti ime. Natančno je določen tudi način, na katerega v podprogram vnesemo vhodne podatke, kakor tudi način, s katerim na koncu iz njega dobimo izhodne podatke.

Poglejmo si primer varnega računanja kvadratnega korena. V množici realnih števil je kvadratni koren določen le za nenegativne vrednosti. Kakšen rezultat dobimo, če računamo kvadratni koren negativnega števila, je odvisno od programskega jezika. Različni konkretni računalniški jeziki bodisi opozorijo na napako bodisi vrnejo določene posebne vrednosti, kakršna je na primer NaN (angl. Not a Number).

Napišimo podprogram, ki vrne za kvadratni koren negativnega števila vrednost -1. Takole je videti ***definicija*** podprograma, zapisana v psevdo jeziku:

```
Podprogram varenKvadratniKoren:
{
  Vhod:  realno število n;
  Če je n ≥ 0:
  {
    Izhod:  √n;
  }
```

```
    sicer:
    {
      Izhod:  −1;
    }
  }
```

Gornja definicija ni nič posebnega. Za besedo `Podprogram` je zapisano ime podprograma (varenKvadratniKoren). Celotna koda podprograma je zapisana v paru zavitih oklepajev, ki sledi. Koda se začne s seznamom vhodnih podatkov, v našem primeru je to eno samo realno število n. Izhod iz podprograma je možen na več mestih (v našem primeru imamo dva izhoda). Paziti moramo le, da vsaka možna pot skozi podprogram na koncu vrne določeno vrednost.

Definicija, ki smo jo pravkar spoznali, je zgolj zapis, ki določa, kaj naj podprogram naredi. Če želimo, da se bo podprogram dejansko izvedel, ga moramo ***klicati***. Podprogram kličemo z imenom, kateremu v par okroglih oklepajev dodamo dejanske vrednosti vhodnih podatkov, ki jim navadno rečemo ***parametri*** ali tudi ***argumenti*** podprograma:

```
Klic:  x ← varenKvadratniKoren(n ← 16);
Sporoči:  x;
```

Iz gornje psevdo kode se lepo vidi, kaj vse se ob klicu podprograma zgodi. Glede na smer puščic (ki predstavljajo smer pretoka podatkov) vidimo, da se zgodba prične na desni in teče proti levi. Najprej vhodno vrednost 16 (t.i. ***dejanski*** parameter) kopiramo v vhodno spremenljivko n (t.i. ***formalni*** parameter). Potem se izvede podprogram z imenom varenKvadratniKoren. Karkoli podprogram vrne (bodisi $\sqrt{n}$ bodisi -1), na koncu kopiramo v spremenljivko x, katere vrednost se v naslednji vrstici izpiše.

V jeziku JavaScript se podprogrami imenujejo funkcije, zato se njihova definicija začne z rezervirano besedo `function`. Vse formalne parametre (t.j. vhodne spremenljivke) funkcije je treba zapisati v par okroglih oklepajev, ki sledi imenu funkcije. Prenos podatka (t.j. izhod) iz funkcije se izvede s posebnim stavkom `return`. Takole je videti definicija funkcije `varenKvadratniKoren`, zapisana v JavaScriptu:

```
//Definicija funkcije:
function varenKvadratniKoren(n) {
  if (n >= 0) {
    return Math.sqrt(n);
  }
  else {
    return -1;
  }
}
```

Pri klicu funkcije v JavaScriptu ne navajamo imen formalnih parametrov, vnesemo zgolj vrednosti dejanskih parametrov:

```
x = varenKvadratniKoren(16);  //klic funkcije
console.log(x);               //izpiše 4
x = varenKvadratniKoren(-7);  //klic funkcije
console.log(x);               //izpiše -1
```

4.1 Lokalne spremenljivke

Pri pisanju podprogramov poleg vhodnih spremenljivk mnogokrat potrebujemo še dodatne (pomožne) spremenljivke. Poskrbeti moramo, da območje delovanja teh spremenljivk omejimo na podprogram. V nasprotnem primeru se nam lahko v program prikradejo napake, ki jih je včasih zelo težko odkriti.

Poglejmo si primer definicije podprograma, ki izračuna faktorielo podanega parametra (program za računanje faktoriele smo zapisali že na strani 12):

```
Podprogram faktoriela:
{
  Vhod:  nenegativno celo število n;
  Lokalno:  f;
  f ← 1;
  Ponavljaj, dokler je n > 1:
  {
    f ← f × n;
    n ← n − 1;
  }
  Izhod:  f;
}
```

V definiciji podprograma smo takoj za seznamom vhodnih parametrov dodali še seznam ***lokalnih spremenljivk***. Vsak od obeh seznamov vsebuje sicer le po eno spremenljivko, vendar lahko uporabimo toliko spremenljivk, kolikor jih potrebujemo. V primeru, da je spremenljivk več, med njihova imena pišemo vejice.

Gornji podprogram lahko preizkusimo z naslednjim algoritmom, ki podprogram kliče 11-krat, in sicer z vrednostmi dejanskega parametra od nič do deset:

```
Ponovi za vsak i od 0 do 10 s korakom 1:
{
  Klic:  x ← faktoriela(n ← i);
  Sporoči:  x;
}
```

Takole je videti definicija funkcije `faktoriela` v jeziku JavaScript:

```
function faktoriela(n) {
  var f;
  f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Iz definicije vidimo, da ***lokalno spremenljivko*** v funkciji ***napovemo*** oziroma ***deklariramo*** s pomočjo besede `var`, ki je angleška kratica za spremenljivko (angl. variable). Delovanje funkcije preizkusimo na naslednji način:

```
for (i = 0; i <= 10; i = i + 1) {
  x = faktoriela(i);
```

```
  console.log(x);
}
```

Program v konzoli prikaže naslednji pričakovani izpis:

```
1
1
2
6
24
120
720
5040
40320
362880
3628800
```

Odstranimo zdaj iz definicije funkcije `faktoriela` deklaracijo lokalne spremenljivke (t.j. odstranimo vrstico `var f;`). Ko program znova zaženemo, dobimo enak izpis kot prej. Zakaj je potem deklaracija potrebna?

Program deluje pravilno samo zato, ker spremenljivke z imenom `f` nismo uporabljali nikjer drugje v programu. Ko pa program raste, se povečuje tudi verjetnost, da bomo še kje drugje uporabili spremenljivko z imenom `f`. Takrat že lahko pričakujemo težave. Zamenjajmo na primer spremenljivko `i` v zadnjem stavku `for` s spremenljivko `f`:

```
for (f = 0; f <= 10; f = f + 1) {
  x = faktoriela(f);
  console.log(x);
}
```

Zdaj dobimo takšen izpis (iz funkcije `faktoriela` je treba seveda odstraniti deklaracijo `var f;`):

```
1
2
6
5040
```

Opazimo, da program »preskakuje« določene vrednosti. To se zgodi zato, ker podprogram `faktoriela` nima svoje lokalne spremenljivke `f`, temveč uporablja isto spremenljivko kot zanka `for`, ki ta podprogram kliče. Spremenljivki, ki se nahaja zunaj kateregakoli podprograma, pravimo ***globalna*** spremenljivka, ker je taka spremenljivka dostopna povsod razen v podprogramih, ki imajo svojo lokalno spremenljivko z istim imenom. Brž ko v definicijo podprograma `faktoriela` vrnemo deklaracijo lokalne spremenljivke `f`, dobimo spet pravilen izpis.

Ugotovili smo, da je ključnega pomena, da pomožne (lokalne) spremenljivke, ki jih uporabljamo v podprogramu, deklariramo z uporabo rezervirane besedice `var`. Na ta način bodo ostale te spremenljivke skrite znotraj podprograma. Kaj pa parametri podprograma? Na srečo tudi vsi formalni parametri podprograma avtomatično postanejo njegove lokalne spremenljivke. To lahko potrdimo, če zaženemo naslednji program:

```
for (n = 0; n <= 10; n = n + 1) {
  x = faktoriela(n);
  console.log(x);
}
```

Ime spremenljivke, ki smo ga tokrat uporabili v zanki (t.j. `n`), je enako imenu formalnega parametra v podprogramu `faktoriela`. Vendar deluje program vseeno pravilno, saj je formalni parameter `n` lokalna spremenljivka podprograma, ki iz gornjega stavka `for` ni dostopna.

4.2 Iskanje napak

V resničnem življenju pri razvoju programske opreme razvijalci daleč največ časa posvečajo preizkušanju kode in iskanju napak. Napaki v programski kodi v računalniškem žargonu pravimo ***hrošč*** (angl. bug), iskanju teh napak pa ***razhroščevanje*** (angl. debugging). V tem razdelku si bomo ogledali dva pogosto uporabljena pristopa pri iskanju napak: ***sled programa*** (angl. program trace) in ***razhroščevanje z razhroščevalnikom***. Razhroščevalnik (angl. debugger) je poseben dodatek, ki ga vsebuje večina razvojnih okolij, med njimi tudi orodja za razvijalce v mnogih brskalnikih.

4.2.1 Sled programa

Sled programa (angl. program trace) pomeni zapisovanje pomembnih dogodkov, ki se vrstijo med izvajanjem programa. Iz takšnega zapisa lahko kasneje v miru proučimo, kako se je program izvajal, in sklepamo na nepravilnosti v njegovem delovanju. Najpogosteje nas zanima, kakšne so vrednosti določenih spremenljivk, kako se med izvajanjem programa te vrednosti spreminjajo ter kateri deli programa se sploh izvajajo. Iz tega, ali se je vrednost določene spremenljivke spremenila ali ne (in kako se je spremenila), lahko že zelo dobro sklepamo, kje bi lahko bila napaka v programu.

Za primer opremimo eno od prejšnjih zank `for` (tisto, ki vsebuje spremenljivko `f`) z nekoliko drugačnim izpisovanjem v konzolo:

```
for (f = 0; f <= 10; f = f + 1) {
  console.log("Pred klicem podprograma:", f); //console.log omogoča
                                              //pisanje več sporočil hkrati,
                                              //le ločiti jih moramo z vejico.
  x = faktoriela(f);
  console.log("Po klicu podprograma:", f);
}
```

Ko program zaženemo, dobimo v konzoli naslednji izpis:

```
Pred klicem podprograma:  0
Po klicu podprograma:  0
Pred klicem podprograma:  1
Po klicu podprograma:  1
Pred klicem podprograma:  2
Po klicu podprograma:  2
...
Pred klicem podprograma:  10
Po klicu podprograma:  10
```

Ta niz zapisov predstavlja sled programa. Iz sledi vidimo, da je vrednost spremenljivke `f` po klicu podprograma vedno enaka, kot je bila pred klicem. To smo tudi pričakovali, saj podprogram `faktoriela` ne sme (in tudi ne more) spremeniti vrednosti spremenljivke, ki jo podamo kot dejanski parameter.

Če zdaj iz podprograma `faktoriela` odstranimo deklaracijo lokalne spremenljivke (t.j. vrstico `var f;`), potem dobimo popolnoma drugačno sled:

```
Pred klicem podprograma:  0
Po klicu podprograma:  1
Pred klicem podprograma:  2
Po klicu podprograma:  2
Pred klicem podprograma:  3
Po klicu podprograma:  6
Pred klicem podprograma:  7
Po klicu podprograma:  5040
```

Ta sled nas takoj opozori na napako v podprogramu, ki očitno spreminja vrednost globalne spremenljivke `f`. Vrednost te spremenljivke je zdaj po klicu podprograma skoraj vedno drugačna kot pred njim.

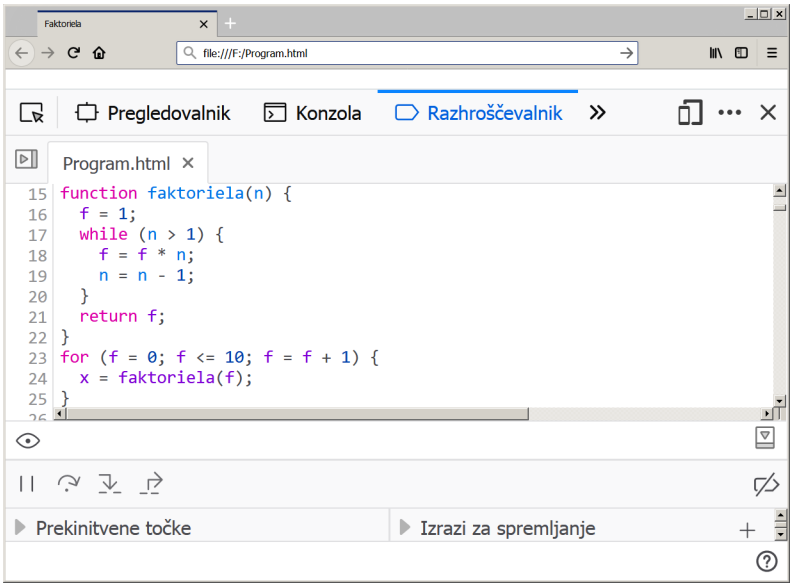
Naloga 4.1 Za vajo zapišite sled, ki jo v naslednjem programu pusti ***lokalna*** spremenljivka `x`. Upoštevajte vse dogodke, kjer se tej spremenljivki priredi vrednost.

```
function test(x) {
  x = x * 2;
  return x;
}

x = 0;
while (x < 10) {
  x = test(x + 1);
}
```

Opomba: Nalogo poskusite rešiti najprej brez uporabe računalnika, potem pa si pomagajte še s funkcijo `console.log`.

4.2.2 Razhroščevalnik

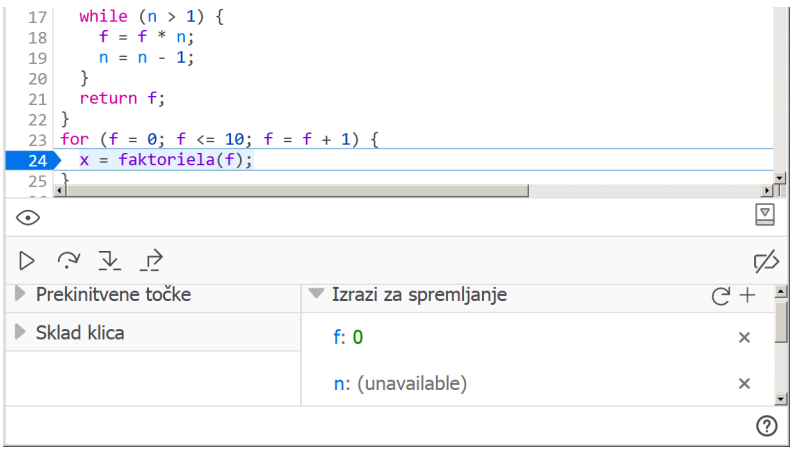
V resnici tudi z razhroščevalnikom opazujemo sled programa. Ena od prednosti razhroščevalnika je v tem, da lahko z njegovo pomočjo izvajamo program korak za korakom ter poleg vrednosti spremenljivk opazujemo tudi vrstni red, v katerem se koda izvaja. Naložimo naš zadnji program (različico brez deklaracije lokalne spremenljivke) v brskalnik Firefox, odprimo okno z razvijalskimi orodji (Ctrl+Shift+I) in kliknimo na jeziček Razhroščevalnik. V spodnjem delu okna zagledamo sporočilo Ctrl P za iskanje virov. Pritisnemo kombinacijo tipk Ctrl+P in vpišemo ime datoteke, ki vsebuje program, ki ga želimo razhroščevati. Dobimo stanje, ki ga prikazuje naslednja slika:



V razhroščevalniku bomo uporabljali naslednje funkcije:

- prekinitvena točka (angl. breakpoint),
- izrazi za spremljanje (angl. watch),
- prestopi (angl. step over),
- vstopi (angl. step into) in
- izstopi (angl. step out).

Prekinitvena točka je mesto v programu, kjer želimo, da se njegovo izvajanje začasno prekine. Nastavimo jo preprosto tako, da kliknemo na številko pred vrstico kode, v kateri želimo izvajanje prekiniti. Če program zdaj znova naložimo v brskalnik (s pritiskom na F5 ali Ctrl+R oziroma s klikom na simbol ↻ levo zgoraj v oknu brskalnika), se bo njegovo izvajanje prekinilo ob prekinitveni točki. Zagledamo naslednjo sliko:

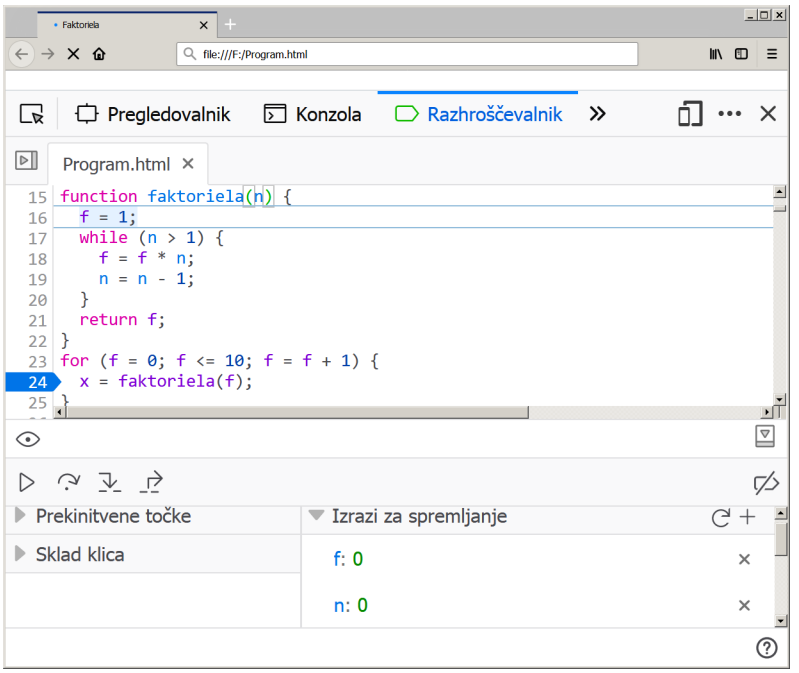


Modra puščica preko številke 24 označuje prekinitveno točko, svetlo modro obarvana vrstica pa označuje kodo, ki se bo izvedla v naslednjem koraku. Zdaj ko program miruje,

si lahko ogledamo trenutne vrednosti njegovih spremenljivk. Zanimata nas spremenljivki `f` in `n`, ki ju dodamo k izrazom za spremljanje: za vsako od spremenljivk kliknemo na znak + na desni ter vpišemo njeno ime. Spodaj desno na gornji sliki lahko vidimo, da ima spremenljivka `f` vrednost nič, za spremenljivko `n` pa piše, da ni na voljo (angl. unavailable). To je razumljivo, saj je `n` vhodni parameter podprograma `faktoriela`, zaradi česar je `n` dostopen samo znotraj tega podprograma. Mi pa se trenutno nahajamo zunaj podprograma `faktoriela`.

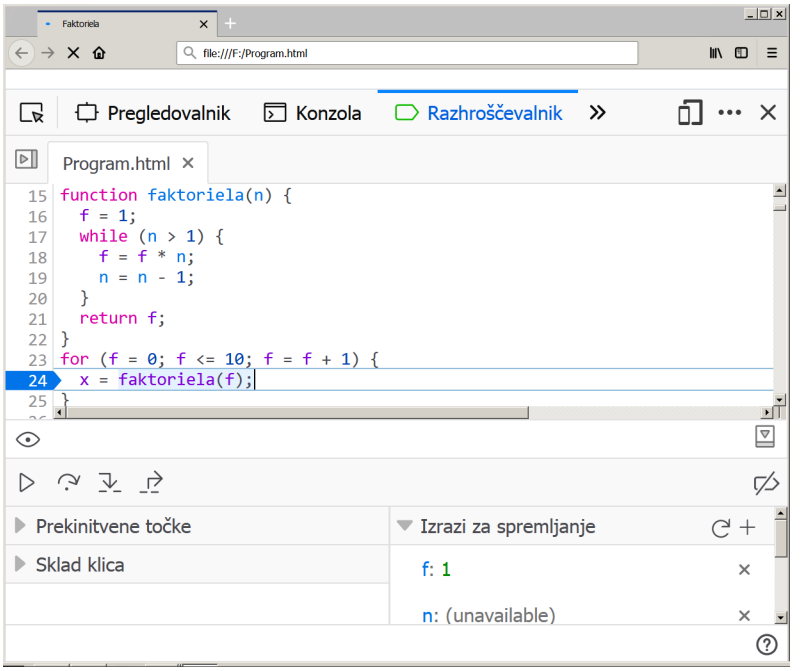
Od tod naprej lahko nadaljujemo na več načinov. Ukaz Prestopi, ki ga izvedemo s pritiskom na F10 ali s klikom na simbol ↷ (spodaj levo), bo izvedel klic podprograma `faktoriela` v enem kosu. Če želimo vstopiti v podprogram in ga izvajati korak za korakom, potem pritisnemo F11 ali kliknemo na simbol ↴. Program lahko tudi poženemo do naslednje prekinitvene točke[1], in sicer tako, da pritisnemo F8 ali kliknemo na simbol ▷.

Pritisnimo zdaj tipko F11, s čimer vstopimo v podprogram `faktoriela`. V razhroščevalniku zaradi tega opazimo dve spremembi: svetlo modro se je obarvala prva vrstica podprograma, ki je naslednja na vrsti za izvajanje, parameter `n` pa je dobil vrednost nič. Ob klicu podprograma se je namreč v (formalni) parameter `n` zapisala vrednost (dejanskega) parametra `f`. Zdaj vidimo v razhroščevalniku takšno sliko:



Pritisnimo zdaj nekajkrat zapored tipko F10 ali F11 (namesto tega lahko tudi samo enkrat pritisnemo Shift+F11 ali kliknemo na simbol ↱, s čimer neposredno izstopimo iz podprograma) in dobimo naslednjo sliko:

[1]Nastavimo lahko poljubno veliko prekinitvenih točk. Če je prekinitvena točka znotraj ponavljalnega stavka, potem se bo program na njej ustavil vsakokrat, ko bo prišel do nje.



Čeprav smo se vrnili na izhodiščno točko, je vrednost spremenljivke `f` zdaj ena. Spremenljivka `f` je torej ohranila vrednost, ki jo je dobila v podprogramu. Če bi nadaljevali s koračnim izvajanjem programa, bi videli, da se ob naslednjem koraku poveča vrednost spremenljivke `f` na dve (zaradi izraza `f = f + 1` v stavku `for`). To je tudi vrednost, ki se uporabi kot dejanski parameter ob naslednjem klicu podprograma `faktoriela`.

Za vajo lahko v definicijo podprograma vrnete deklaracijo lokalne spremenljivke `f` in postopek ponovite. Videli boste, da se zdaj ob vrnitvi iz podprograma vrednost spremenljivke `f` vsakokrat ponastavi na vrednost, ki jo je imela tik pred klicem podprograma. V resnici se vrednost ne ponastavi, temveč se spet pokaže vrednost prvotne (globalne) spremenljivke `f`. Ta spremenljivka je hranila svojo vrednost ves čas, ko smo bili v podprogramu in je nismo mogli videti.

Pozorni bodite na barvo spremenljivk `n` in `f` v programski kodi, prikazani v oknu razhroščevalnika. Barvi sta različni, ker je `n` lokalna, `f` pa globalna spremenljivka. Razhroščevalnik brskalnika Firefox obarva lokalne spremenljivke modro, globalne pa vijolično. To nam lahko pomaga pri iskanju napak, ki so posledica tega, da smo pozabili deklarirati lokalno spremenljivko. Če v definicijo funkcije `faktoriela` vrnete deklaracijo spremenljivke `f`, bo ta postala modre barve.

Na koncu naj omenimo, da iskanje napak ni točna znanost, temveč bolj veščina ali celo umetnost, ki izhaja iz izkušenj. Prvi pogoj, da napako sploh začnemo iskati, je, da opazimo nepravilnost v delovanju programa. To pa je že novo področje, ki se ukvarja s preverjanjem pravilnosti delovanja programa. Za preverjanje pravilnosti programov obstajajo določeni formalni matematični postopki, ki se v resničnem življenju zaradi zahtevnosti redkeje uporabljajo. Najpogostejši pristop preverjanja pravilnosti ostane tako preizkušanje z določenim naborom testnih podatkov. Določanje in izbiranje teh testnih podatkov je pogosto spet stvar iznajdljivosti in izkušenosti.

4.3 Podajanje parametrov po sklicu

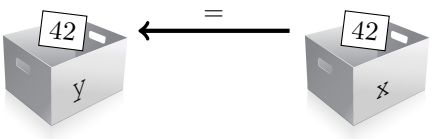
Zelo pomemben pojem v računalniškem programiranju je ***sklic*** ali ***referenca*** (angl. reference) na podatek. Za ilustracijo tega pojma si najprej oglejmo naslednji enostavni program:

```
x = 42;
y = x;
console.log(x, y);
x = 13;
console.log(x, y);
```

V prvi vrstici programa spremenljivki x priredimo vrednost 42. Z vidika računalnika to pomeni, da moramo v pomnilniku najprej rezervirati določen prostor (z imenom x) in nato vanj shraniti vrednost 42. Naslednja slika prikazuje stanje v pomnilniku, pri čemer smo prostor v pomnilniku, namenjen shranjevanju vrednosti, prikazali s škatlo:

Na škatli je zapisano ***ime spremenljivke***, v škatli pa je shranjena njena ***vrednost***. V drugi vrstici programa s priredilnim operatorjem (=) kopiramo vrednost spremenljivke x v spremenljivko y. To pomeni, da se mora v pomnilniku najprej ustvariti prostor z imenom y:

Potem naredimo s pomočjo priredilnega operatorja kopijo vrednosti iz škatle x in jo shranimo v škatlo y:



Ko zdaj v tretji vrstici programa v konzolo izpišemo vrednosti spremenljivk x in y, vidimo, da imata obe spremenljivki zares vrednost 42. V četrti vrstici programa nato spremenimo vrednost spremenljivke x na 13. To pomeni, da smo iz škatle z imenom x odstranili staro vrednost (42) in jo nadomestili z novo (13). Takšno je končno stanje v pomnilniku:

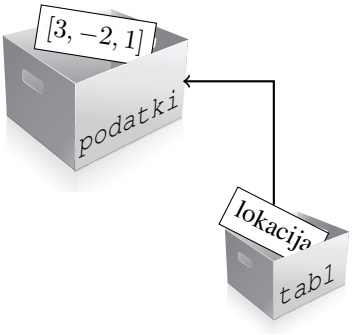
Zadnji priredilni stavek ne vpliva na vrednost spremenljivke `y`, zato zadnja vrstica programa izpiše različni vrednosti za spremenljivki `x` in `y`.

Napišimo še en podoben program, le da namesto skalarnih vrednosti uporabimo tabele:

```
tab1 = [3, -2, 1];
tab2 = tab1;
console.log(tab1, tab2);
tab1[1] = 15;
console.log(tab1, tab2);
```

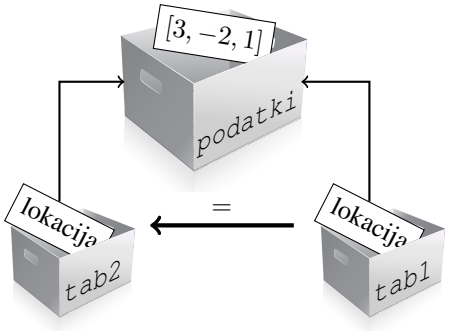
Ko program zaženemo, presenečeni ugotovimo, da se je v njegovi predzadnji vrstici spremenila tudi vrednost tabele `tab2`, čeprav smo spremenili le vrednost drugega elementa tabele `tab1`. Zadnja vrstica programa namreč izpiše dve popolnoma enaki tabeli (t.j. `[3, 15, 1]`).

Da bi razumeli, kaj se je zgodilo, si moramo ogledati razporeditev podatkov v pomnilniku. Obstaja bistvena razlika med skalarnimi spremenljivkami, ki hranijo eno samo vrednost, in tabelami, ki hranijo poljubno mnogo vrednosti. Kot smo videli v prejšnjem primeru, se vrednosti skalarnih spremenljivk hranijo neposredno v spremenljivki sami. S tabelami je zgodba drugačna. Tabele hranijo zgolj informacijo o ***lokaciji*** podatkov v pomnilniku. Povedano drugače, v spremenljivki, ki je tabela, je shranjen le podatek o tem, ***kje*** se dejanski podatki nahajajo. Naslednja slika prikazuje stanje, ko se izvede prva vrstica gornjega programa:

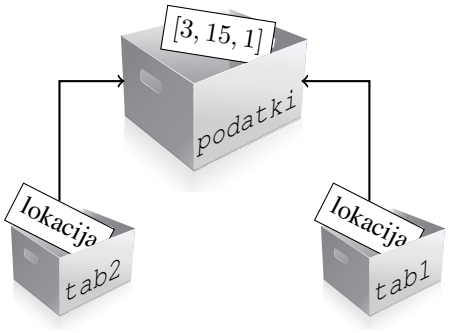


V škatli z imenom `tab1` je podatek o tem, kje se nahaja druga škatla, v kateri so dejanski podatki. Puščica na gornji sliki prikazuje povezavo med vsebino škatle `tab1` in škatlo s podatki.

S priredilnim operatorjem v drugi vrstici kode kopiramo vsebino spremenljivke `tab1` v spremenljivko `tab2`. Vsebina spremenljivke `tab1` pa niso dejanski elementi tabele, temveč zgolj informacija o tem, kje ti elementi so. Taki informaciji pravimo tudi ***sklic*** oziroma ***referenca*** na podatek. Spremenljivka `tab2` tako prejme kopijo sklica na isti podatek, na katerega se sklicuje tudi `tab1`. Takšno je zdaj stanje v pomnilniku:



Ko v četrti vrstici gornjega programa spremenimo drugi element tabele `tab1`, v resnici spremenimo element, ki se nahaja v škatli z imenom `podatki`, ki je tudi edina obstoječa kopija tabele. Ko se izvede stavek `tab1[1] = 15`, je stanje v pomnilniku takšno:



Zdaj vemo, zakaj izpiše zadnja vrstica programa za obe tabeli enake vrednosti: vrednosti, ki jih vsebuje `tab2`, so dejansko ***iste*** (in ne le enake) kot vrednosti, ki jih vsebuje `tab1`, saj se obe tabeli sklicujeta na eno in isto kopijo podatkov.

Upravičeno se nam zastavi vprašanje o smislu uporabe sklica, namesto da bi podatek uporabili neposredno. Prvi in najočitnejši primer uporabe sklica je pri podajanju parametrov podprogramom. Pri klicu podprograma se vedno najprej kopirajo vrednosti dejanskih parametrov v formalne parametre. Zlahka si predstavljamo, da bi obsežne tabele precej upočasnile delovanje programa, če bi morali takšne tabele vsakokrat v celoti kopirati (tako ob vhodu kot tudi ob izhodu iz podprograma). Težava je rešena tako, da se podprogramu ob njegovem klicu poda zgolj sklic na tabelo, prek katerega ima podprogram poln dostop do vseh njenih elementov. To pa hkrati pomeni, da lahko tak podprogram elemente tabele ne le bere, temveč tudi spreminja. To dejstvo lahko večkrat s pridom izkoristimo tudi tako, da uporabimo vhodne parametre podprograma kot dodatno možnost za vračanje vrednosti iz podprograma (poleg uporabe stavka `return`), kakor kaže naslednji primer.

Poglejmo si podprogram, ki kopira elemente ene tabele v drugo. Takole je videti njegova definicija v psevdo jeziku:

```
Podprogram kopirajTabelo:
{
  Vhod:  tabela izvor, zaključena s čuvajem (0);
  Vhod/izhod:  tabela cilj;
  Lokalno:  i;
  Izhod:  —;
```

```
  Ponovi za vsak i od 0 do konca tabele izvor s korakom 1:
  {
    cilj_i ← izvor_i;
  }
  Dodaj čuvaja (0) na konec tabele cilj;
}
```

Ker je tabela ***cilj*** v resnici parameter podprograma (vhod), hkrati pa bomo vanjo zapisali izhodne podatke (kopijo tabele ***izvor***), smo to tabelo v definiciji podprograma zapisali pod kategorijo `Vhod/izhod`. Eksplicitnega vračanja podatkov iz podprograma s stavkom `return` ne potrebujemo, zato smo kategorijo `Izhod` pustili prazno (—). Vidimo, da podprogram kopirajTabelo nima pretirano veliko dela. Najprej mora – element za elementom – kopirati izvorno tabelo v ciljno. Kot zadnje dejanje mora na konec ciljne tabele dodati še čuvaja.

Ko podprogram kličemo, mu lahko za cilj podamo poljubno tabelo, v katero se bo kopirala izvorna tabela:

```
t1 ← [a, b, c, 0];
t2 ← [ ];
Klic:   kopirajTabelo(cilj ← t2, izvor ← t1);
Sporoči:  t1, t2;
```

Tabeli $t1$ in $t2$ se zdaj sklicujeta na enako, ne pa tudi isto vsebino. Če bi na primer spremenili vrednost kakšnega elementa katerekoli od obeh tabel, bi druga od tabel ostala nedotaknjena. Zapišimo še definicijo funkcije in primer njenega klica v JavaScriptu, da bomo lahko njegovo delovanje preverili na računalniku:

```
function kopirajTabelo(cilj, izvor) {
  var i;
  for (i = 0; izvor[i] != 0; i = i + 1) {
    cilj[i] = izvor[i];
  }
  cilj[i] = 0;
}
t1 = ['a', 'b', 'c', 0];
t2 = [];
kopirajTabelo(t2, t1);
console.log(t1, t2);
```

Naloga 4.2 Sami napišite in preizkusite podprogram v jeziku JavaScript, ki tabeli naravnih števil, zaključeni s čuvajem (0), na koncu doda en element. Element boste dodali na mesto čuvaja, za njim pa morate dodati novega čuvaja, da bo dobljena tabela še vedno pravilno zaključena.

Izhajajte iz naslednje definicije podprograma:

```
Podprogram dodajElement:
{
  Vhod/izhod:  tabela tab, zaključena s čuvajem (0);
  Vhod:  element el;
  Lokalno:  i;
```

```
    Izhod:   —;
    Preštej število elementov tabele tab in
      dobljeno število shrani v i;
    tab_i ← el;      //Opomba: doda nov element
    tab_{i+1} ← 0;   //Opomba: doda čuvaja
}
```

Za konec si poglejmo še primer podprograma, ki naj bi med seboj zamenjal vsebini dveh enako dolgih tabel, zaključenih z ničlo. Takole napišemo definicijo funkcije v JavaScriptu, ki naj bi opravila zastavljeno nalogo:

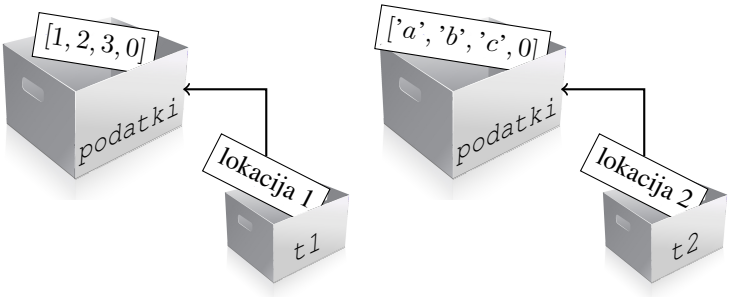
```
function zamenjajTabeli(tab1, tab2) {
  var zacasno;
  zacasno = tab1;
  tab1 = tab2;
  tab2 = zacasno;
}
```

Vendar funkcija ne deluje:

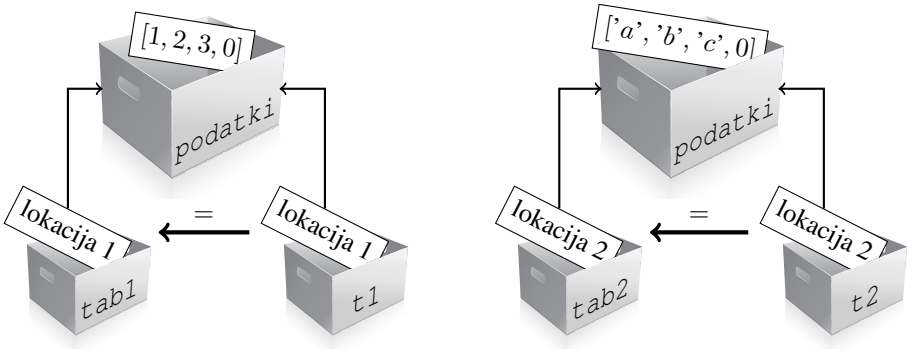
```
t1 = [1, 2, 3, 0];
t2 = ['a', 'b', 'c', 0];
console.log(t1, t2);    //izpiše [1, 2, 3, 0] [a, b, c, 0]
zamenjajTabeli(t1, t2);
console.log(t1, t2);    //izpiše [1, 2, 3, 0] [a, b, c, 0]
```

Zastavi se vprašanje, zakaj se vrednosti v tabelah niso zamenjale, ko pa smo znotraj podprograma vendar delali s sklicema na obe tabeli. Vedeti moramo, da funkcija ob klicu vedno prejme ***kopijo*** parametra, tudi če je ta parameter sklic. V gornji definiciji funkcije `zamenjajTabeli` smo med seboj torej zamenjali ***kopiji*** originalnih vrednosti sklicev (`t1` in `t2`). Sprememba kopije podatka o lokaciji elementov seveda nikakor ne more vplivati na dejanske vrednosti elementov, shranjene v pomnilniku, niti ne more spremeniti vrednosti originalnih sklicev `t1` in `t2`.

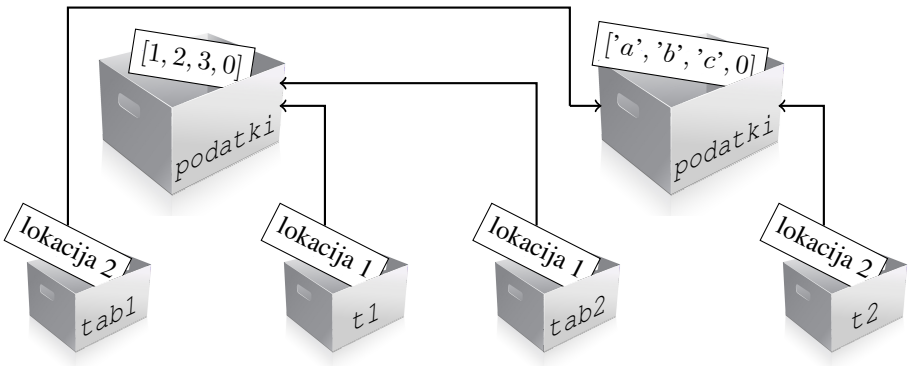
Poglejmo si nekoliko podrobneje, kaj se dogaja v gornjem programu. Tik pred klicem funkcije `zamenjajTabeli` je stanje v pomnilniku takšno:



Ob klicu funkcije `zamenjajTabeli` se vrednosti dejanskih parametrov `t1` in `t2` kopirata v formalna parametra `tab1` in `tab2`. Parameter `tab1` se tako sklicuje na isto tabelo kot spremenljivka `t1`, `tab2` pa se sklicuje na isto tabelo kot `t2`. Stanje v pomnilniku prikazuje naslednja slika:



V funkciji `zamenjajTabeli` zdaj med seboj zamenjamo vrednosti formalnih parametrov `tab1` in `tab2`, kar pa seveda ne vpliva na vrednosti spremenljivk `t1` in `t2`. Parametra `tab1` in `tab2` se zaradi menjave sklicujeta na drugi tabeli kot prej, medtem ko se spremenljivki `t1` in `t2` še vedno sklicujeta na isti tabeli kot pred klicem funkcije. Stanje v pomnilniku je zdaj takšno:



Ko se klic funkcije `zamenjajTabeli` konča, nimamo več dostopa do parametrov `tab1` in `tab2`. V pomnilniku imamo zato spet natanko takšno stanje, kakršno smo imeli tik pred klicem te funkcije.

Videli smo, da z menjavo dveh tabel znotraj podprograma zamenjamo zgolj kopiji sklicev na obe tabeli, kar nima zunaj podprograma nikakršnega učinka. Popolnoma drugače pa bo delovala naslednja funkcija:

```
function zamenjajTabeli(tab1, tab2) {
  var i, zacasno;
  for (i = 0; tab1[i] != 0; i = i + 1) {
    zacasno = tab1[i];
    tab1[i] = tab2[i];
    tab2[i] = zacasno;
  }
}
```

Parametra `tab1` in `tab2` sta še vedno kopiji sklicev na dve tabeli. Vendar zdaj ne spreminjamo več samih kopij sklicev. Zaradi uporabljenih indeksnih operatorjev (`[i]`) spreminjamo elemente tabel, na katere se kopiji `tab1` in `tab2` sklicujeta. Sklicujeta pa se na originalni tabeli v pomnilniku. Funkcija zdaj dejansko zamenja med seboj elemente obeh podanih tabel:

```
t1 = [1, 2, 3, 0];
t2 = ['a', 'b', 'c', 0];
console.log(t1, t2);    //izpiše [1, 2, 3, 0] [a, b, c, 0]
zamenjajTabeli(t1, t2);
console.log(t1, t2);    //izpiše [a, b, c, 0] [1, 2, 3, 0]
```

4.4 Naloge

Za vajo še nekaj izzivov:

Naloga 4.3 Napišite in preizkusite podprogram, ki kot parametra sprejme dve nenegativni celoštevilski vrednosti. Prva naj bo največ štirimestna, druga pa naj ima vrednost med nič in tri. Podprogram naj vrne števko (cifro) iz prve podane vrednosti, ki se nahaja na mestu, ki ga določa druga podana vrednost. Če ima prva podana vrednost manj kot štiri mesta, predpostavite, da so na manjkajočih vodilnih mestih ničle. Na primer, klic `izberiCifro(59, 1)` naj vrne 0, klic `izberiCifro(217, 3)` naj vrne 7, klic `izberiCifro(5432, 0)` pa 5.

Pomoč: Pomagate si lahko z rešitvijo problema na strani 16.

Naloga 4.4 Napišite podprogram, ki za parametra sprejme naravno število in prazno tabelo, zaključeno s čuvajem (0). Podprogram naj v podano tabelo vpiše vse ***prave delitelje*** podanega naravnega števila. Pravi delitelj (angl. proper divisor) števila n je vsak pozitiven delitelj, ki se razlikuje od n. Na primer, pravi delitelji števila 24 so $1, 2, 3, 4, 6, 8$ in 12.

Napisan podprogram uporabite za preverjanje, ali je določeno število ***popolno število***. Popolno število (angl. perfect number) je vsako naravno število, ki je enako vsoti vseh svojih pravih deliteljev. Prva štiri popolna števila so 6, 28, 496 in 8128 (oeis.org/A000396).

Naloga 4.5 Napišite program, ki reši naslednji problem:

> ***Vhod:*** tabela realnih števil $\boldsymbol{t}$ z n elementi;
> ***Zahtevani izhod:*** tabela $\boldsymbol{t}$ z elementi, urejenimi po velikosti (od najmanjšega proti največjemu);

Uporabite lahko ***urejanje z navadnim izbiranjem*** (angl. selection sort), ki ga opisuje naslednji algoritem:

```
Vhod:  t, n;
Izhod:  urejena tabela t;
Ponovi za vsak i od 0 do n − 2 s korakom 1:
{
  Zamenjaj med sabo t_i in najmanjšega od elementov t_i, t_{i+1}, ··· , t_{n−1};
}
Sporoči:  t;
```

Vrstico znotraj ponavljalnega stavka v gornjem algoritmu razvijemo naprej v naslednji algoritem:

```
Klic:   j ← min(t ← t, start ← i, stop ← n);
Zamenjaj med sabo t_i in t_j;
```

Da dokončate nalogo, morate napisati še podprogram z imenom min, ki vrne indeks elementa z najmanjšo vrednostjo. Podprogram naj kot prvi parameter sprejme enorazsežnostno tabelo $\boldsymbol{t}$, v kateri tak element iščemo. Podprogram naj poleg tega sprejme še argument $start$, ki predstavlja indeks elementa, pri katerem začnemo iskati, ter argument $stop$, ki je za ena večji od indeksa elementa, pri katerem končamo z iskanjem.

Nenapisano pravilo pravi, da funkcijam, ki obdelujejo le del tabele od enega do drugega indeksa, podamo drugi indeks za ena prevelik. Za to obstajajo določeni razlogi. Mnogokrat nas na primer zanima število elementov, ki jih bo takšna funkcija obdelala. To število dobimo enostavno tako, da od končnega indeksa (ki je za ena prevelik) odštejemo začetni indeks: $stElementov = stop - start$.

Naloga 4.6 Napišite podprogram z imenom uredi, ki kot parameter sprejme tabelo naravnih števil, zaključeno s čuvajem (0) ter njene elemente uredi po velikosti od najmanjšega proti največjemu. Za rešitev problema uporabite postopek ***urejanja z mehurčki*** (angl. bubble sort), ki ga opisuje naslednji algoritem:

```
Podprogram uredi:
{
  Vhod/izhod:  tabela naravnih števil t, zaključena s čuvajem (0);
  Izhod:   —;
  Ponovi enkrat manj, kolikor je elementov v tabeli t:
  {
    Ponovi za vsak element tabele t razen zadnjega:
    {
      Če je element večji od naslednjega:
      {
        Zamenjaj med sabo element in naslednji element;
      }
    }
  }
}
```

Napisani podprogram preizkusite na naslednji način:

```
tab1 = [3, 7, 2, 66, 2, 0];
tab2 = [15, 8, 449, 61, 7, 38, 0];
uredi(tab1);
uredi(tab2);
console.log(tab1); //izpiše [2, 2, 3, 7, 66, 0]
console.log(tab2); //izpiše [7, 8, 15, 38, 61, 449, 0]
```

Dodatek: Algoritem urejanja z mehurčki lahko naredite v povprečju učinkovitejši, če zunanjega ponavljalnega stavka ne ponavljate tolikokrat, kolikor je elementov v

tabeli (oz. enkrat manj), temveč ga zaustavite, kakor hitro ugotovite, da je tabela urejena.

Naloga 4.7 Naslednji program poišče največji skupni delitelj dveh naravnih števil z uporabo Evklidovega postopka:

```
function nsd(a, b) {
  while (b != a) {
    while (a > b) {
      a = a - b;
    }
    while (b > a) {
      b = b - a;
    }
  }
  return a;
}

x = nsd(21, 49);
```

Zapišite sled programa v obliki parov vrednosti spremenljivk `a` in `b`.

Naloga 4.8 Naslednji program uredi po velikosti od najmanjšega proti največjemu elemente podane tabele, zaključene s čuvajem (0). Urejanje je izvedeno po postopku z ***navadnim vstavljanjem*** (angl. insertion sort):

```
function uredi(t) {
  var i, j, zacasno;
  for (i = 0; t[i] != 0; i = i + 1) {
    j = i;
    while (j > 0 && t[j - 1] > t[j]) {
      zacasno = t[j];
      t[j] = t[j - 1];
      t[j - 1] = zacasno;
      j = j - 1;
    }
  }
}

tabela = [7, 5, 1, 3, 0];
uredi(tabela);
```

Zapišite sled programa, ki jo pusti tabela po vsakokratni menjavi dveh njenih elementov.

Naloga 4.9 Funkcijo `zamenjajTabeli`, ki smo jo napisali na strani 61, lahko zapišemo tudi brez uporabe pomožne lokalne spremenljivke `zacasno`:

```
function zamenjajTabeli(tab1, tab2) {
```

```
    var i;
    for (i = 0; tab1[i] != 0; i = i + 1) {
      tab1[i] = tab1[i] + tab2[i];
      tab2[i] = tab1[i] - tab2[i];
      tab1[i] = tab1[i] - tab2[i];
    }
  }
```

Takšna funkcija deluje seveda samo v primeru, ko so elementi obeh tabel številske vrednosti[a]:

```
t1 = [1, 2, 3, 0];
t2 = [4, 5, 6, 0];
console.log(t1, t2);    //izpiše [1, 2, 3, 0] [4, 5, 6, 0]
zamenjajTabeli(t1, t2);
console.log(t1, t2);    //izpiše [4, 5, 6, 0] [1, 2, 3, 0]
```

Vendar funkcija vsebuje težko izsledljivo napako. Če jo pokličemo tako, da za oba parametra podamo isto tabelo, dobimo na kocu tabelo s samimi ničlami:

```
t1 = [1, 2, 3, 0];
console.log(t1);    //izpiše [1, 2, 3, 0]
zamenjajTabeli(t1, t1);
console.log(t1);    //izpiše [0, 0, 0, 0]
```

Z uporabo razhroščevalnika poskusite ugotoviti, zakaj se to zgodi.

[a] Teoretično je dovolj, da so vrednosti številske. V praksi veljajo še omejitve, ki so pogojene s konkretnim načinom kodiranja posameznih števil, s čimer pa se v tem učbeniku ne bomo ukvarjali.

Naloga 4.10 Napišite podprogram, ki pretvori desetiško celo število med vključno ena in 3999 v rimsko številko. Prvi parameter podprograma naj bo desetiško število, ki ga mora pretvoriti. Drugi parameter naj bo prazna tabela, zaključena s čuvajem (0), kamor bo podprogram vpisal pretvorjeno rimsko številko.

Pomoč: Naslednja tabela prikazuje rimske številke, ki predstavljajo desetiške stotice, desetice in enice:

	0	1	2	3	4	5	6	7	8	9
stotice (10^2)	—	C	CC	CCC	CD	D	DC	DCC	DCCC	CM
desetice (10^1)	—	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
enice (10^0)	—	I	II	III	IV	V	VI	VII	VIII	IX

Iz tabele lahko razberemo, da so vzorci rimskih simbolov v vseh vrsticah enaki, razlika je le v naboru simbolov. Na primer, iz katerekoli enice lahko dobimo ustrezno desetico preprosto tako, da vse I-je zamenjamo z X-i, vse V-je zamenjamo z L-ji in vse X-e zamenjamo s C-ji.

Potrebujemo torej tri nabore rimskih simbolov, za vsako od vrstic gornje tabele po enega. Za stotice bomo uporabljali simbole iz pomožne tabele $\boldsymbol{rimSimboli}_0 =$

$[\mathrm{C}, \mathrm{D}, \mathrm{M}]$, za desetice simbole iz pomožne tabele $\boldsymbol{rimSimboli}_1 = [\mathrm{X}, \mathrm{L}, \mathrm{C}]$ in za enice bomo izbirali simbole iz pomožne tabele $\boldsymbol{rimSimboli}_2 = [\mathrm{I}, \mathrm{V}, \mathrm{X}]$.

Če dejanske simbole v katerikoli od treh vrstic gornje tabele nadomestimo z indeksi simbolov, ki jih bomo izbirali iz ustrezne pomožne tabele ($\boldsymbol{rimSimboli}_0$ do $\boldsymbol{rimSimboli}_2$), potem dobimo takšno tabelo:

0	1	2	3	4	5	6	7	8	9
—	0	00	000	01	1	10	100	1000	02

To tabelo lahko zapišemo kot naslednjo dvorazsežnostno tabelo, kjer vrednost -1 igra vlogo čuvaja:

$$\boldsymbol{rimIndeksi} = [[-1], [0, -1], [0, 0, -1], [0, 0, 0, -1], [0, 1, -1], [1, -1], [1, 0, -1], [1, 0, 0, -1], [1, 0, 0, 0, -1], [0, 2, -1]]\,.$$

Zdaj že lahko zapišemo definicijo podprograma za pretvorbo desetiškega števila v rimsko:

```
Podprogram desetiskoVRimsko:
{
  Vhod: desetisko;
  Vhod/izhod: tabela rimsko, zaključena s čuvajem (0);
  Lokalno: i, j, k, c, rimIndeksi, rimSimboli;
  Izhod: —;
  Izberi tisočice iz desetisko in jih shrani v c;
  Na konec tabele rimsko dodaj c M-jev;
  i ← 0;
  Ponovi trikrat:
  {
    Izberi naslednjo cifro iz desetisko in jo shrani v c;
    Ponovi za vsak j od 0 do konca tabele rimIndeksi_c s korakom 1:
    {
      k ← rimIndeksi_{c,j};
      Na konec tabele rimsko dodaj rimSimboli_{i,k};
    }
    i ← i + 1;
  }
}
```

Za dodajanje elementa na konec tabele lahko uporabite podprogram iz naloge 4.2 na strani 59. Za izbiranje cifre iz desetiške vrednosti pa lahko uporabite podprogram iz naloge 4.3 na strani 62.

5. POGLAVJE

NAČRTOVANJE ALGORITMOV IN PODATKOV

Neko podjetje je v oglasu, s katerim so iskali nove programerje, objavilo naslednjo nalogo:

> Z uporabo dveh trojk, dveh osmic, operatorjev seštevanja, odštevanja, množenja in deljenja ter oklepajev sestavite vrednost 24. Uporabiti morate natanko dve trojki in dve osmici, operatorje pa lahko izbirate poljubno.

Tistega, ki je rešil zastavljeno nalogo, so avtomatično povabili na razgovor za službo.

Nalogo je mogoče rešiti na različne načine. Mi se je bomo lotili tako, da bomo načrtovali program, ki jo bo rešil namesto nas. Problem je ravno pravšnji oreh, da nam bo pomagal ilustrirati nekaj najosnovnejših pristopov k načrtovanju algoritmov in podatkov.

Eden najbolj naravnih pristopov k načrtovanju je tako imenovano ***načrtovanje z vrha navzdol*** (angl. top-down design). Včasih mu pravimo tudi ***postopno načrtovanje*** (angl. stepwise design) oziroma načrtovanje z ***razgradnjo problema*** (angl. problem decomposition). Načrtovanje z vrha navzdol v osnovi pomeni, da sistem postopoma razčlenjujemo na njegove posamezne sestavne dele. V prvem koraku določimo, kakšen bo videti in kako bo deloval celoten sistem. Potem tak sistem razčlenimo na podsisteme, kjer za vsakega posebej določimo, kako bo deloval in kako bo povezan z ostalimi podsistemi. Vsakega od teh podsistemov nato razčlenimo še na podrobnejše podsisteme, dokler ni celoten načrt razdelan do osnovnih elementov.

Takšen pristop k načrtovanju je zelo intuitiven. Na primer, če želimo načrtovati zabavo za rojstni dan, potem bo prvi korak členitve videti takole:

> povabi ljudi,
> nabavi pijačo,

nabavi hrano,
rezerviraj prostor.

Od tod naprej lahko členimo nabavo pijače v nabavo brezalkoholnih in alkoholnih pijač. Nabavo brezalkoholnih pijač naprej členimo v nabavo gaziranih in negaziranih pijač, in tako dalje. Seveda je treba pri takem načrtovanju upoštevati tudi medsebojne povezave posameznih podsklopov: vrsta in količina hrane in pijače bo odvisna od tega, kateri ljudje se bodo odzvali na povabilo in koliko jih bo prišlo. Izbira prostora je prav tako odvisna od ljudi, hkrati pa se bodo nekateri odločali o tem, ali pridejo ali ne, tudi na podlagi lokacije.

Če dobro pomislimo, potem smo pristop z vrha navzdol že uporabili, ko smo načrtovali program, ki najde pot skozi blodnjak. Na primer, določene ukaze v algoritmu na strani 38 (ki že predstavlja zadovoljivo rešitev problema) smo morali izdelati še bolj do potankosti, preden smo lahko zapisali končen program.

Pristop k načrtovanju z vrha navzdol se je začel v praksi pospešeno uveljavljati v začetku sedemdesetih let prejšnjega stoletja, ko je Harlan Mills, raziskovalec v podjetju IBM, razvil koncept ***strukturiranega programiranja***. Pri tem je bila pomembna ugotovitev, da je katerikoli zamisljiv program mogoče sestaviti iz samo treh osnovnih načinov kombiniranja podprogramov:

- z izvajanjem podprogramov enega za drugim (zaporedno izvajanje);
- z izvajanjem enega od dveh podprogramov glede na vrednost določenega logičnega pogoja (odločanje oz. izbira);
- s ponavljanjem izvajanja podprograma, dokler je izpolnjen določen pogoj (ponavljanje oz. iteracija).

Pri tem je ključnega pomena dejstvo, da vsi trije načini ohranjajo osnovno strukturo programskih blokov z enim samim vhodom in izhodom ne glede na to, kako zapletena je struktura rešitve problema. Z drugimi besedami to pomeni, da s kombiniranjem podprogramov (ali osnovnih ukazov) po kateremkoli od gornjih treh načinov dobimo kodo, s katero lahko neposredno nadomestimo katerikoli ukazni blok v obsežnejšem algoritmu. Če torej sledimo konceptu strukturiranega programiranja, se osnovna struktura algoritma med členjenjem problema avtomatično ohranja.

Pomembna posledica uvedbe modela strukturiranega programiranja so bili razumljivejši in bolj zanesljivi programi, ki jih je razvijalcem praviloma uspelo napisati v precej krajšem času.

Poleg načrtovanja z vrha navzdol moramo omeniti še en pogost pristop k načrtovanju, ki mu pravimo ***načrtovanje z dna navzgor*** (angl. bottom-up design). Tu izhajamo iz posameznih obstoječih komponent ali manjših sistemov, ki jih postopoma sestavljamo v bolj zapletene sisteme.

Vrnimo se k načrtovanju rojstnodnevne zabave. Zamislimo si situacijo, ko želimo preizkusiti nekaj novih receptov. Po teh receptih pripravimo različne vrste hrane in pijače ter vsako posebej poskusimo, ali je recept uspel. Zdaj imamo na voljo določene količine pripravljene hrane in pijače in zabavo načrtujemo glede na njihovo vrsto in količino. V tem primeru smo ubrali pot z dna (od priprave hrane in pijače) navzgor (proti organizaciji celotne zabave).

V praksi je načrtovanje z dna navzgor zaželeno med drugim tudi zato, ker lahko določene komponente sistema preizkusimo ločeno in že v zgodnji fazi gradnje celotnega sistema. Pri načrtovanju z vrha navzdol je možno delovanje sistema preizkusiti pogosto šele v zelo pozni fazi njegove gradnje.

Sodobni načini načrtovanja programske opreme običajno združujejo oba pristopa: z vrha navzdol in z dna navzgor. Seveda je za uspešno načrtovanje najprej potrebno, da dobro razumemo celoten sistem, kar nujno vodi k načrtovanju z vrha navzdol. Po drugi strani skušajo razvijalci pri večini programerskih projektov v določeni meri izhajati iz že napisanih kosov programske opreme. Uporaba obstoječih delov kode daje načrtovanju priokus pristopa z dna navzgor. V mnogih inženirskih panogah izkušeni načrtovalci načrtujejo posamezne komponente neodvisno od večjih sistemov. Tako izdelane komponente je možno kasneje zlagati v večje sisteme, podobno kot zlagamo kocke Lego. Takšnemu načrtovanju pravimo ***načrtovanje komponent*** (angl. piece part design).

Vrnimo se zdaj k naši nalogi z dvema trojkama in dvema osmicama. Po principu načrtovanja z vrha navzdol najprej določimo, kako se bomo lotili problema iskanja matematičnega izraza, ki bo dal želeni rezultat:

> Preglej vse možne izraze, ki jih lahko sestaviš iz dveh trojk in dveh osmic ter osnovnih matematičnih operatorjev, in vrni tistega, katerega vrednost je 24;

Vseh možnih izrazov je sicer veliko, vendar ne toliko, da jih računalnik ne bi bil sposoben pregledati v zelo kratkem času. Takšnemu načinu reševanja problemov, ko enostavno preverimo vse obstoječe možnosti, pravimo v računalniškem žargonu postopek ***surove sile*** (angl. brute force).

Gornji algoritem lahko takoj razdelamo v ponavljanje dveh podrobneje opredeljenih korakov:

```
Ponovi za vsak izraz, ki ga lahko sestaviš iz dveh trojk in dveh osmic
ter osnovnih matematičnih operatorjev (izr3388_i):
{
  Izračunaj vrednost izraza izr3388_i;
  Če je izračunana vrednost enaka 24:
  {
    Sporoči:  izr3388_i;
  }
}
```

Izkaže se, da ni prav enostavno sestaviti vseh možnih matematičnih izrazov, ki ustrezajo danemu pogoju. Tudi računanje vrednosti izrazov, ki vsebujejo oklepaje ter operatorje z različnimi prednostmi, je sorazmerno zahteven programerski izziv. Množenje in deljenje v takšnih izrazih je treba izvesti pred seštevanjem in odštevanjem. Prav tako je treba obravnavati dele izrazov v oklepajih pred ostalimi. Gnezdeni pari oklepajev stanje le še poslabšajo.

Življenje si lahko precej poenostavimo, če za računanje uporabimo nekoliko neobičajen način zapisovanja matematičnih izrazov. Poljski matematik in filozof Jan Łukasiewicz je v zgodnjih letih prejšnjega stoletja ugotovil, da je možno zapisati vsak matematični izraz tudi brez uporabe oklepajev. Njegov način zapisovanja postavlja operatorje pred operande, in ne mednje, kakor je to navada v matematiki. V čast temu poljskemu matematiku se njegov zapis imenuje ***poljski zapis*** (angl. Polish notation). S prihodom računalnikov se je razvil tako imenovani ***vzvraten poljski zapis*** (angl. reverse Polish notation (RPN)), kjer se zapisuje operatorje ***za*** operandi. Na primer, namesto običajnega $3 + 8$ bomo isti izraz zapisali v zapisu RPN kot $38+$.

Za računanje izrazov, zapisanih v obliki RPN, bomo potrebovali ***sklad*** (angl. stack), ki ga bomo načrtovali posebej. Sklad je vrsta ***abstraktnega podatkovnega tipa*** (angl. abstract data type). To je matematični model, ki poleg podatkov (t.j. množice dovoljenih vrednosti)

definira tudi operacije nad temi podatki. Sklad bomo načrtovali kot programski objekt, zato si najprej poglejmo, kaj je to ***objekt***.

5.1 Objekt

Objekt je programska tvorba, ki pod enim imenom združuje tako podatke kot tudi operacije, ki jih lahko nad temi podatki izvajamo. Podatkom, ki jih objekt vsebuje, pravimo tudi ***lastnosti*** (angl. properties). Operacijam, ki jih lahko izvajamo nad podatki objekta, pa pravimo ***postopki*** (angl. methods). Vzemimo za primer predvajalnik datotek MP3. Na tak predvajalnik lahko gledamo kot na objekt, ki vsebuje na primer zbirko skladb in podatek o številu skladb (lastnosti). Nad to zbirko lahko izvajamo različne operacije: predvajaj, pojdi na začetek naslednje/prejšnje/trenutne skladbe, ustavi/prekini/nadaljuj predvajanje, označi kot priljubljeno, izbriši skladbo in podobno (postopki). Poleg naloženih skladb ima predvajalnik MP3 še mnoge druge lastnosti, kot na primer trenutna jakost predvajanja ali velikost pomnilnika.

V programskem smislu zapisujemo lastnosti in postopke tako, da najprej navedemo ime objekta, potem pa s pomočjo ***selektorja*** izberemo želeno lastnost ali postopek. Selektor je poseben operator, ki je v večini programskih jezikov zapisan s piko. V primeru predvajalnika MP3 bi bilo to videti takole:

```
predvajalnik.skladbe          //Opomba: seznam skladb
predvajalnik.skladbe_i        //Opomba: i-ta skladba
predvajalnik.stSkladb         //Opomba: število skladb
predvajalnik.pomnilnik        //Opomba: velikost razpoložljivega pomnilnika
predvajalnik.naslednja()      //Opomba: predvajaj naslednjo skladbo
predvajalnik.premor()         //Opomba: začasno prekini predvajanje
```

Iz zapisa vidimo, da so lastnosti v resnici spremenljivke, ki hranijo podatke, postopki pa so podprogrami oziroma funkcije. Slednje je razvidno iz parov okroglih oklepajev za imenoma funkcij naslednja in premor. Par oklepajev nakazuje, da gre za klic podprograma oziroma funkcije.

Za primer bomo zdaj sestavili objekt, ki se bo gibal skozi blodnjak. Nalogo smo že rešili (glej strani 37 do 41), le da v rešitvi nismo uporabili objekta. Spomnimo se, da smo zapisali (kodirali) položaj v blodnjaku in smer gibanja po njem kot dva vektorja s po dvema elementoma. To so podatki, ki bodo zdaj predstavljali lastnosti našega objekta. Nad temi podatki smo izvajali tudi nekaj operacij: tipanje, korak naprej in obračanje v smeri oziroma nasprotni smeri urinega kazalca. To bodo postopki našega objekta. Zapišimo zdaj objekt v psevdo jeziku:

```
Objekt Tezej:        //Opomba: Objekt poimenujemo po starogrškem junaku Tezeju, ki je
{                    //ubil Minotavra in z Ariadnino pomočjo uspešno pobegnil iz blodnjaka.
  Lastnosti:  x, y, dx, dy;
  Postopki:   tipaj, obratZa90, korak;
};

Tezej.tipaj = podprogram:
{
  Vhod:  blodn;
  Lokalno:  i, j;
  i ← Tezej.x + Tezej.dx;
  j ← Tezej.y + Tezej.dy;
```

```
  Izhod:  blodn_{i,j};
};

Tezej.obratZa90 = podprogram:
{
  Vhod:  smer;  //Opomba: smer = 1 pomeni smer ure, smer = −1 pa nasprotno smer ure.
  Lokalno:  zacasno;
  Izhod:  —;
  zacasno ← −smer × Tezej.dx;
  Tezej.dx ← smer × Tezej.dy;
  Tezej.dy ← zacasno;
};

Tezej.korak = podprogram:
{
  Vhod:  —;
  Izhod:  —;
  Tezej.x ← Tezej.x + Tezej.dx;
  Tezej.y ← Tezej.y + Tezej.dy;
};

Tezej.oznaciPoložaj = podprogram:
{
  Vhod:  blodn;
  blodn_{Tezej.x,Tezej.y} ← 3;
};
```

Med lastnostmi objekta predstavljata x in y vrstico in stolpec elementa v blodnjaku, na katerem se objekt trenutno nahaja. Lastnosti $\mathrm{d}x$ in $\mathrm{d}y$ predstavljata spremembo vrstice in stolpca v primeru, da želimo objekt premakniti[1]. Postopek tipaj je zapisan kot podprogram, ki kot parameter sprejme celoten blodnjak (oziroma sklic na blodnjak). Podprogram najprej izračuna številko vrstice (i) in stolpca (j) elementa, na katerega bi se objekt premaknil, če bi izvedli korak naprej. Na koncu vrne vrednost tega elementa.

Postopek (podprogram) obratZa90 sprejme en parameter, s katerim mu sporočimo želeno smer obrata: $smer = 1$ pomeni, da se bo objekt zasukal za $90°$ v smeri urinega kazalca, $smer = -1$ pa nasproti urinega kazalca. Kodo v podprogramu že poznamo iz različice programa brez objekta: med seboj moramo zamenjati vrednosti lastnosti $\mathrm{d}x$ in $\mathrm{d}y$ in eno od njiju množiti z -1, odvisno od smeri obrata. Postopek obratZa90 ne vrača ničesar, ker izvaja operacijo nad lastnostmi objekta.

Naslednji postopek je korak. Ta niti ne sprejema parametrov niti ne vrača vrednosti. Vse, kar mora storiti, je, da lastnosti x prišteje lastnost $\mathrm{d}x$ in lastnosti y prišteje lastnost $\mathrm{d}y$. S tem premakne objekt za en korak naprej po blodnjaku.

Čisto zadnji postopek (z imenom oznaciPoložaj) skrbi za to, da v blodnjaku označimo položaj, na katerem se nahajamo. To storimo preprosto tako, da na ustrezno mesto v blodnjaku vpišemo trojko.

V JavaScriptu zapišemo definicijo gornjega objekta takole:

```
Tezej = Object(); //POZOR: Object() z veliko začetnico!

Tezej.tipaj = function(blodn) {
  var i, j;
  i = Tezej.x + Tezej.dx;
```

[1] V različici brez objekta smo te podatke zapisali kot pol_0 in pol_1 ter $smer_0$ in $smer_1$

```
  j = Tezej.y + Tezej.dy;
  return blodn[i][j];
};

Tezej.korak = function() {
  Tezej.x = Tezej.x + Tezej.dx;
  Tezej.y = Tezej.y + Tezej.dy;
};

Tezej.obratZa90 = function(smer) {
  //smer = 1:  v smeri ure
  //smer = -1:  v nasprotni smeri ure
  var zacasno;
  zacasno = Tezej.dx;
  Tezej.dx = smer * Tezej.dy;
  Tezej.dy = -smer * zacasno;
};

Tezej.oznaciPolozaj = function(blodn) {
  blodn[Tezej.x][Tezej.y] = 3;
};
```

Koda ni nič posebnega. Objekt ustvarimo s klicem funkcije `Object()`, ki vrne sklic na ustvarjen objekt. Tako kot za običajen podprogram tudi za definicijo postopka uporabimo besedico `function`, le da je zapis nekoliko drugačen. Poleg tega moramo paziti, da pred ime postopka vedno zapišemo še ime objekta s selektorjem (piko). Lastnosti objekta ni treba posebej določevati. Uporabljamo jih na enak način kot običajne spremenljivke, le da spredaj dodamo ime objekta s selektorjem.

Program na strani 40 lahko zdaj zapišemo mnogo krajše in predvsem razumljivejše:

```
Tezej.x = 1;
Tezej.y = 10;
Tezej.dx = 1;
Tezej.dy = 0;
smerIskanja = -1; //-1:  vedno skušamo zaviti levo
                  // 1:  vedno skušamo zaviti desno

while (Tezej.tipaj(blodnjak) != 2) {
  Tezej.obratZa90(-smerIskanja);
  while (Tezej.tipaj(blodnjak) == 1) {
    Tezej.obratZa90(smerIskanja);
  }
  Tezej.oznaciPolozaj(blodnjak);
  Tezej.korak();
}
```

V gornjem programu smo uporabili spremenljivko `smerIskanja`, ki določa, v katero smer iščemo pot. Kadar je njena vrednost -1, takrat skušamo zaviti levo, če se to le da. V nasprotnem primeru, kadar je njena vrednost enaka ena, skušamo vedno zaviti desno. To spremenljivko smo uvedli zato, da nam ni treba popravljati programa na dveh mestih, če želimo spremeniti smer iskanja.

V različici programa na strani 40 smo v pogoju zunanjega stavka `while` preverjali vrednost elementa, na katerem stojimo, zaradi česar smo na koncu dejansko izstopili iz blodnjaka. Zdaj pa na tem mestu tipamo en element pred seboj, zato se program ustavi, tik preden Tezej izstopi iz blodnjaka. Vendar to ni težava, saj smo izhod že našli.

Če želite, lahko tudi tokrat rešen blodnjak prikažete v brskalniku s klicem funkcije `narisiBlodnjak`, ki jo najdete v dodatku C.

5.2 Sklad (abstrakten podatkovni tip)

Sklad (angl. stack) lahko obravnavamo kot običajno tabelo, vendar z drugačnim načinom dostopa do elementov. V tabeli lahko v vsakem trenutku pridemo do kateregakoli elementa enako hitro in enako učinkovito, če le poznamo njegovo zaporedno številko (indeks). Takemu načinu dostopa do podatkov pravimo tudi ***naključni dostop*** (angl. random access). Dostop do elementov na skladu pa je navadno omejen tako, da lahko beremo le element, ki je na »vrhu« sklada. Ob branju elementa se ta običajno tudi odstrani s sklada (angl. pop). Tudi dodajanje elementov na sklad je omejeno tako, da lahko nove elemente postavimo le na vrh sklada (angl. push).

Naslednja slika prikazuje tri zaporedna stanja na skladu, potem ko smo nanj postavili elemente z vrednostmi -3, 26 in 91:

Če zdaj s sklada vzamemo en element, potem dobimo element z vrednostjo 91, na skladu pa ostaneta elementa -3 in 26, kakor prikazuje naslednja slika:

Ker s sklada vedno dobimo podatek, ki smo ga nanj odložili nazadnje, pravimo, da sklad deluje po načelu LIFO (angl. last in, first out – zadnji noter, prvi ven).

Sklad se v praksi uporablja v najrazličnejših primerih. Mnogi programski jeziki (npr. grafični jezik PostScript) in navidezna izvajalska okolja (npr. Java navidezni stroj – angl. Java virtual machine) temeljijo na skladih. Klici podprogramov v praktično vseh programskih jezikih prav tako uporabljajo sklad za začasno hranjenje vrednosti parametrov in drugih lokalnih spremenljivk. Tudi mnogi algoritmi, kot na primer sledenje nazaj (angl. backtracking), slonijo na uporabi sklada.

Sklad bomo zapisali kot objekt, ki bo imel dve lastnosti: tabelo $\boldsymbol{podatki}$, ki bo hranila elemente sklada, in spremenljivko vrh, ki bo hranila indeks prvega prostega mesta na skladu. Na primer, ko je sklad prazen, ima vrh vrednost nič, če pa so na skladu trije elementi, ima vrh vrednost tri (ne smemo pozabiti, da je tri zaporedna številka četrtega elementa v tabeli).

Poleg postopkov push in pop, ki bosta služila postavljanju in odvzemanju elementov s sklada, bomo skladu dodali še postopek clear, ki bo sklad izpraznil, in postopek size, ki bo sporočil trenutno število elementov na skladu. Pri tem nalašč uporabljamo angleška imena, ki so splošno sprejeta. Takole je videti zapis objekta $sklad$ v psevdo jeziku:

```
Objekt sklad:
{
  Lastnosti:  podatki, vrh;
  Postopki:   push, pop, size, clear;
};

sklad.podatki ← [ ];
sklad.vrh ← 0;

sklad.push = podprogram:
{
  Vhod:  element;
  Izhod:  —;
  sklad.podatki_{sklad.vrh} ← element;
  sklad.vrh ← sklad.vrh + 1;
};

sklad.pop = podprogram:
{
  Vhod:  —;
  sklad.vrh ← sklad.vrh − 1;
  Izhod:  sklad.podatki_{sklad.vrh};
};

sklad.clear = podprogram:
{
  Vhod:  —;
  Izhod:  —;
  sklad.vrh ← 0;
};

sklad.size = podprogram:
{
  Vhod:  —;
  Izhod:  sklad.vrh;
};
```

Iz definicije postopka push vidimo, da moramo po tem, ko podani element vpišemo na prvo prosto mesto v tabeli $sklad.\boldsymbol{podatki}$, takoj povečati tudi vrh sklada. To naredimo tako, da spremenljivki $sklad.vrh$ prištejemo konstantno vrednost ena. Podobno moramo pred branjem vrhnjega elementa (postopek pop) najprej zmanjšati vrh sklada za ena.

Kot vidimo iz zapisane kode, postopek clear v resnici ne izprazni sklada, temveč zgolj postavi spremenljivko $sklad.vrh$ na vrednost nič. Glede na to, da celoten dostop do sklada temelji izključno na vrednosti te spremenljivke, je popolnoma vseeno, kakšne vrednosti so v resnici zapisane v tabeli $sklad.\boldsymbol{podatki}$ – če je vrednost vrha enaka nič, potem je sklad prazen.

Tudi postopek size je sila preprost. Vse, kar naredi, je to, da vrne vrednost spremenljivke $sklad.vrh$ in upravičeno se nam zastavi vprašanje, zakaj je za to potreben poseben postopek. Eno izmed pomembnih načel objektnega programiranja je tako imenovano ***skrivanje informacije*** (angl. information hiding). S skrivanjem informacije uporabnika objekta (t.j. programerja, ki objekt uporablja pri pisanju svojega programa) odvrnemo od neposrednega dostopa do lastnosti objekta. Namesto neposrednega dostopa ponudimo za upravljanje z lastnostmi objekta ustrezne postopke. Vzemimo za primer, da se v prihodnosti iz takšnega ali drugačnega razloga (npr. zaradi učinkovitejše izrabe pomnilnika) spremeni notranja zgradba objekta (t.j. njegove lastnosti in njihova zgradba, ne pa

tudi postopki). Načelo skrivanja informacije nam v takem primeru zagotavlja, da ni treba spreminjati tudi vse že napisane kode, ki tak spremenjen objekt uporablja.

V vsakdanjem življenju srečamo primere skrivanja informacije praktično na vsakem koraku. Na primer, mnogi vozniki, ki so se naučili voziti avto v času, ko še ni bilo servo volana, spremembe praktično ne bodo opazili, čeprav je v ozadju popolnoma drugačen sistem. Volan z vidika voznika deluje še vedno po istem načelu[2].

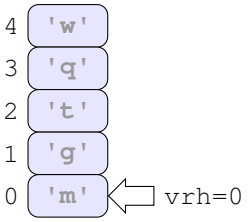
Čas je že, da sklad zapišemo tudi v jeziku JavaScript:

```
sklad = Object();
sklad.podatki = [];
sklad.vrh = 0;

sklad.push = function(element) {
  sklad.podatki[sklad.vrh] = element;
  sklad.vrh = sklad.vrh + 1;
};

sklad.pop = function() {
  sklad.vrh = sklad.vrh - 1;
  return sklad.podatki[sklad.vrh];
};

sklad.size = function() {
  return sklad.vrh;
};

sklad.clear = function() {
  sklad.vrh = 0;
};
```

Napisani objekt preizkusimo z naslednjim programom:

```
sklad.push('a');
sklad.push('b');
sklad.push('c');
console.log(sklad.size());  //izpiše 3
while (sklad.size() > 0) {
  console.log(sklad.pop()); //izpiše c, b in a
}
```

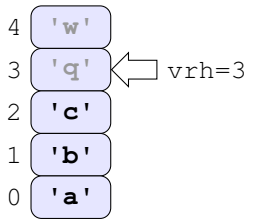
Poskusimo pojasniti princip delovanja sklada še z nekaj slikami, s katerimi bomo ponazorili stanje v pomnilniku. Naslednja slika predstavlja prazen sklad:



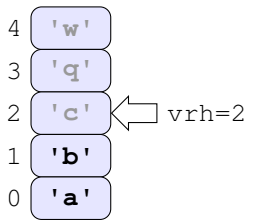
[2]Čeprav bodo izkušeni vozniki z veliko občutka za vožnjo seveda čutili pomembne razlike.

Osenčeni pravokotniki predstavljajo elemente tabele, ki hrani podatke sklada. Na levi strani, pred pravokotniki, so zapisani indeksi posameznih elementov, v pravokotnikih pa so prikazane vrednosti elementov. Vrednosti so izbrane naključno in ta hip niso zares pomembne. Ker je sklad v resnici prazen, so vrednosti elementov prikazane v medlo sivi barvi. Kot že vemo, spremenljivka `vrh` vedno nosi vrednost prvega prostega elementa na skladu. V programerskem žargonu pravimo, da ta spremenljivka ***kaže*** na vrh sklada, kar je na sliki prikazano z belo puščico. Spremenljivka `vrh` trenutno kaže na dno sklada, kar pomeni, da je sklad prazen (vrh sklada je na dnu).

Ko položimo element na sklad (t.j. izvedemo operacijo push), v resnici shranimo vrednost elementa v celico, na katero kaže spremenljivka `vrh`. Hkrati vrednost te spremenljivke povečamo za 1. Naslednja slika kaže stanje na skladu po tem, ko smo v zadnjem programu nanj po vrsti postavili vrednosti `'a'`, `'b'` in `'c'`:



Če s takšnega sklada vzamemo en element (t.j. izvedemo operacijo pop), dobimo vrnjen vrhnji element (t.j. `'c'`), `vrh` pa se zmanjša za ena. Po tem je sklad videti takole:



Naloga 5.1 Za vajo dodajte objektu `sklad` postopek peek (slov. pokukaj), ki vrne vrednost elementa na vrhu sklada, vendar elementa ne odstrani s sklada. Delovanje postopka preizkusite z naslednjim programom:

```
sklad.push('a');
sklad.push('b');
sklad.push('c');
sklad.push('d');
console.log(sklad.size());  //izpiše 4
console.log(sklad.peek());  //izpiše d
console.log(sklad.size());  //izpiše 4
console.log(sklad.pop());   //izpiše d
console.log(sklad.pop());   //izpiše c
console.log(sklad.size());  //izpiše 2
console.log(sklad.peek());  //izpiše b
console.log(sklad.size());  //izpiše 2
```

5.3 Računanje po načelu RPN

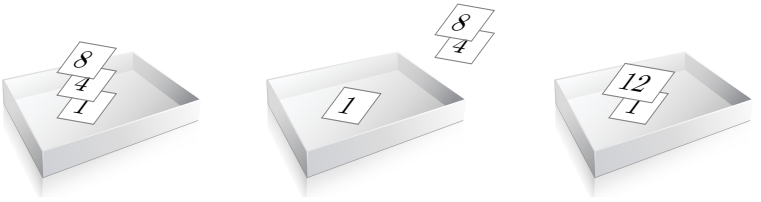
Zdaj ko imamo sklad, se lahko lotimo načrtovanja podprograma, ki bo računal vrednost izraza, zapisanega v obliki RPN. Spomnimo se: takšne izraze bomo potrebovali za reševanje našega problema dveh trojk in dveh osmic.

Oglejmo si najprej, kako poteka računanje vrednosti izraza RPN. Za primer vzemimo naslednji izraz, zapisan v obliki tabele, zaključene s čuvajem:

$$\boldsymbol{rpn} = [1, 4, 8, +, 3, *, -, 0]\,.$$

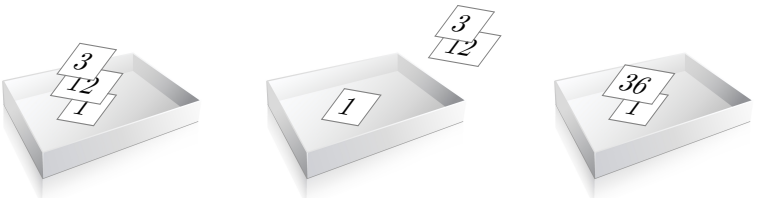
Naši izrazi bodo poleg desetiških števk med ena in devet vsebovali še operatorje seštevanja, odštevanja, deljenja in množenja.

Postopek računanja po načelu RPN je preprost: kadar naletimo na operand (v našem primeru desetiško števko), ga postavimo na sklad. Kadar naletimo na operator, s sklada najprej vzamemo potrebno število elementov (v našem primer dva), izvedemo operacijo ter rezultat postavimo nazaj na sklad. Naslednja slika prikazuje dogajanje na skladu med obdelovanjem prvih štirih elementov gornje tabele $\boldsymbol{rpn}$:

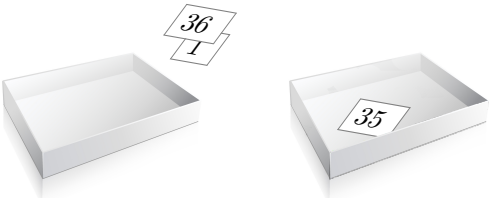


Skrajno levo vidimo stanje po tem, ko smo na sklad postavili števke 1, 4 in 8. V sredini je prikazan prvi korak seštevanja, ko s sklada vzamemo vrhnja dva elementa. Elementa na koncu seštejemo in vsoto odložimo nazaj na sklad. Končno stanje je prikazano na desni.

Peti in šesti element gornje tabele $\boldsymbol{rpn}$ sta števka tri in operator množenja. Naslednja slika prikazuje tri zaporedna stanja po tem, ko smo na sklad postavili trojko, s sklada vzeli vrhnja dva elementa in na koncu odložili njun produkt nazaj na sklad:



Čisto zadnja operacija v tabeli $\boldsymbol{rpn}$ je odštevanje, ki spet vzame (edina preostala) elementa s sklada, ju med seboj odšteje in razliko odloži nazaj na sklad:



Če je število števk in operatorjev ustrezno (v našem primeru mora biti število operatorjev za ena manjše, kakor je število števk), potem nam na skladu ostane en sam element, ki hkrati predstavlja končno vrednost izraza.

S tem smo opravili prvi korak načrtovanja (z vrha navzdol) našega podprograma za izračun vrednosti izraza RPN: pojasnili smo, kako mora sistem delovati. V naslednjem koraku že lahko napišemo algoritem:

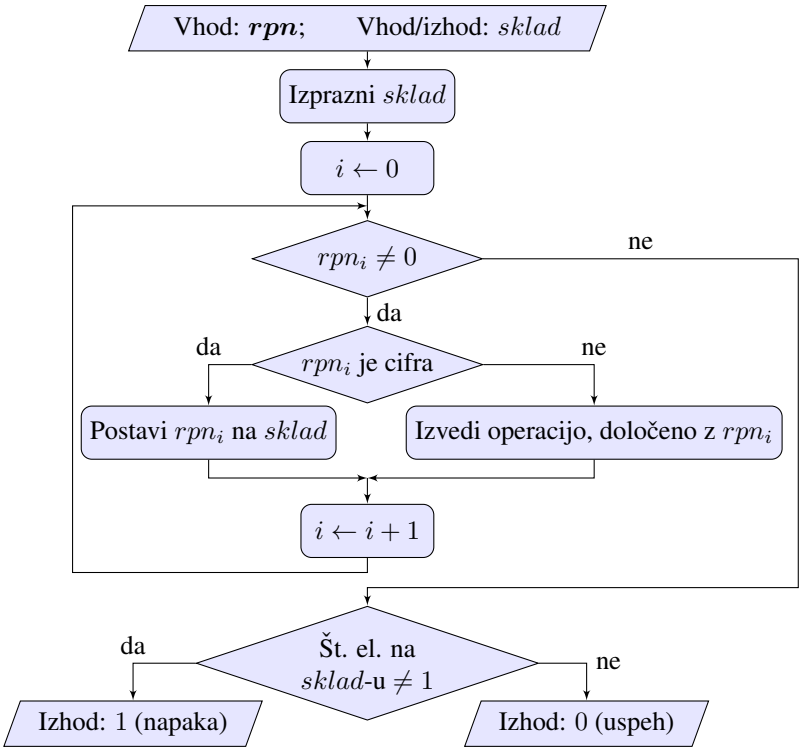
```
Podprogram izracunaj:
{
  Vhod: izraz, zapisan v tabeli rpn;
  Vhod/izhod: sklad;
  Izprazni sklad;
  Ponovi za vsak element tabele rpn:
  {
    Če je rpn_i desetiška števka:
    {
      Postavi rpn_i na sklad;
    }
    sicer:
    {
      Izvedi operacijo, ki jo določa element rpn_i;
    }
  }
  //Opomba: Če izraz v tabeli rpn ne vsebuje napak, je na skladu
            zdaj en sam element, ki predstavlja vrednost izraza.
  Izhod: številka napake (0: uspeh, ≠0: napaka);
}
```

Poleg tabele rpn podprogramu kot parameter podamo tudi sklad, ki ga bomo potrebovali za računanje. Enako kot tabele se v jeziku JavaScript tudi objekti podajajo po sklicu. To dejstvo s pridom uporabimo za vračanje končne izračunane vrednosti, za katero smo videli, da ostane na skladu. Zato je v gornjem algoritmu parameter $sklad$ naveden kot vhodno izhodni parameter. Sklad moramo na začetku izprazniti (za vsak primer, če ni že prazen), sicer podprogram ne bo deloval pravilno. Jedro gornjega algoritma pa je ponavljalni stavek, ki vsak element posebej bodisi postavi na sklad bodisi izvede operacijo, ki jo element določa.

Obstaja možnost, da vhodni izraz (tabela rpn) vsebuje napako. Sporočilo o tem, ali je bilo računanje uspešno ali ne, zakodiramo kot celoštevilsko vrednost, ki jo podprogram vrne s stavkom `return`. Ničla pomeni, da je bilo računanje uspešno, če pa podprogram vrne vrednost, različno od nič, vemo, da je prišlo do ene od možnih napak. Številke napak si lahko izberemo sami. Uporabili bomo številke, ki so zbrane v naslednji tabeli:

Štev. napake	**Opis**
0	brez napake
1	preveč operandov ali prazen izraz
2	premalo operandov
3	nedovoljen simbol
4	deljenje z nič

Naslednji diagram poteka prikazuje nekoliko podrobnejšo različico zadnjega algoritma. Od možnih napak, ki so zbrane v gornji tabeli, lahko v tem koraku načrtovanja zaznamo le napako številka 1 (preveč operandov ali prazen izraz):



Ker gre za podprogram, ki sprejema parametre in vrača vrednosti, smo ukaznim in odločitvenim blokom dodali še vhodno izhodne bloke. Slednji so v diagramu prikazani s podolgovatimi paralelogrami.

V diagramu lahko prepoznamo stavek `for`, ki smo ga uporabili zato, da po vrsti pregledamo vse elemente tabele $\boldsymbol{rpn}$. Čisto na koncu diagrama preverimo, koliko elementov je na skladu. Če je število elementov na skladu različno od ena, potem podprogram vrne napako številka 1 (preveč operandov ali prazen izraz). V resnici bi se dalo sporočilo o tej napaki ločiti na dve sporočili. Izkaže se namreč, da je na tem mestu sklad lahko prazen le, če je tudi tabela $\boldsymbol{rpn}$ prazna (vsebuje zgolj čuvaja). Če pa je na skladu več kot en element, potem to pomeni, da je bilo v izrazu $\boldsymbol{rpn}$ preveč operandov oziroma premalo operatorjev.

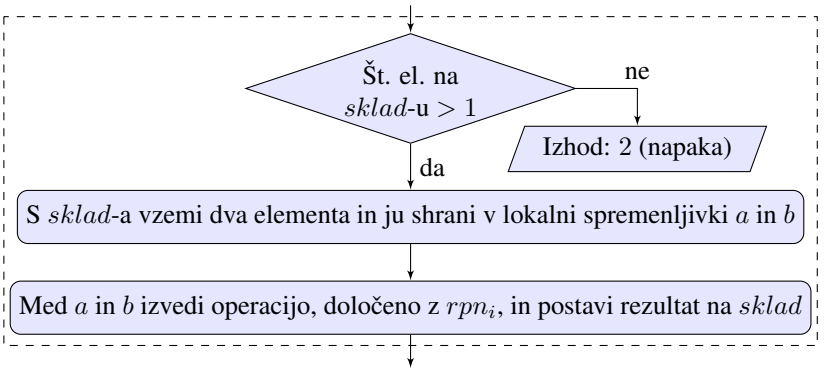
Vsi razen enega bloka v gornjem diagramu poteka že predstavljajo osnovne komponente, ki jih v postopku načrtovanja od zgoraj navzdol ni treba več členiti. Naprej moramo členiti le še ukazni blok »Izvedi operacijo, določeno z rpn_i«. Tu moramo najprej preveriti, ali sta na skladu vsaj dva elementa. Če nista, potem operacije ne moremo izvesti in vrnemo napako. Če je na skladu dovolj elementov, s sklada vzamemo dva elementa in ju shranimo v začasni lokalni spremenljivki a in b. Na koncu izvedemo operacijo in rezultat postavimo na sklad. Takole je videti algoritem v psevdo jeziku:

```
Če vsebuje sklad več kot 1 element:
{
  S sklad-a vzemi dva elementa in ju shrani v
    lokalni spremenljivki a in b;
  Med a in b izvedi operacijo, določeno z
    rpn_i, in postavi rezultat na sklad;
}
```

```
sicer:
{
  Izhod:  2 (premalo operandov);
}
```

Napaka številka 2 (premalo operandov) ne pomeni nujno, da je v celotni tabeli $\boldsymbol{rpn}$ premalo operandov. Pomeni le, da jih je bilo premalo do trenutka, ko smo naleteli na operator na i-tem mestu v tabeli, kar je seveda tudi napaka.

Gornji del algoritma lahko ponazorimo z naslednjim diagramom poteka:



Pomembno je, da ima ta del algoritma obliko z enim samim vhodom in izhodom. To dejstvo je v gornjem diagramu poteka prikazano s črtkanim pravokotnikom. Vse skupaj lahko zato brez težav vstavimo na mesto ukaznega bloka »Izvedi operacijo, določeno z rpn_i« v diagramu poteka na strani 79, s čimer sledimo načelu strukturiranega programiranja (glej stran 68).

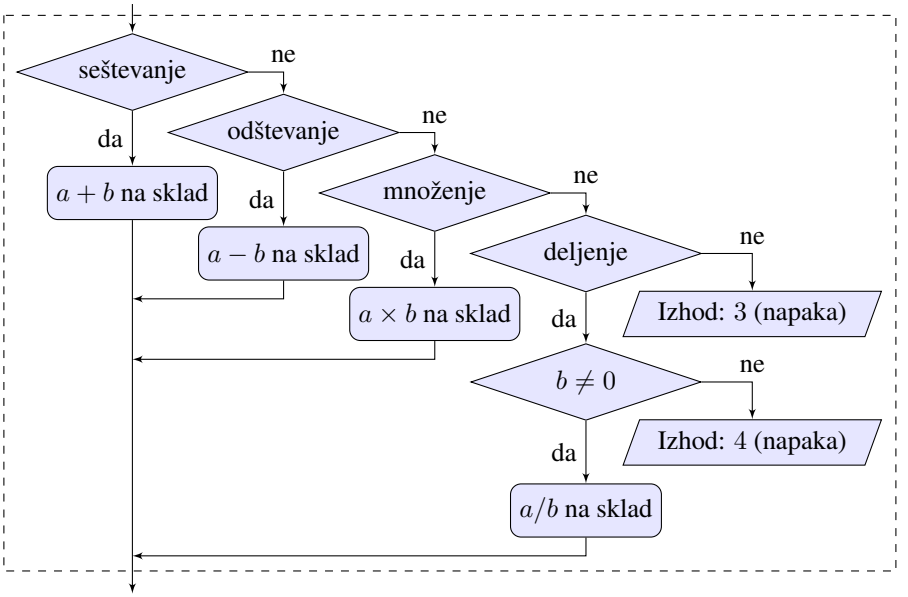
Načelo strukturiranega programiranja dopušča odstopanja, ki v določenih primerih vodijo k bolj razumljivi in obvladljivi kodi. Tak primer je zaznavanje napak med delovanjem programa. Napake pogosto zahtevajo odziv, ki spremeni običajen tok izvajanja programa. V gornjem diagramu poteka vidimo, da v primeru napake predčasno zapustimo podprogram. V resnici smo ustvarili dodaten izhod iz programskega bloka, ki ga uporabimo v primeru napake v vhodnem izrazu, ki onemogoča normalno nadaljevanje izvajanja podprograma.

Na koncu razčlenimo še zadnji ukazni blok gornjega dela algoritma (t.j. blok »Med a in b izvedi operacijo, določeno z rpn_i, in postavi rezultat na sklad«). Celoten blok zapišemo kot verigo stavkov če-sicer, ki po vrsti preverijo, ali element rpn_i ustreza kateri od štirih možnih operacij. Če ne ustreza, potem vrnemo številko napake 3 (nedovoljen simbol). Poleg tega je treba pri deljenju preveriti še, ali je vrednost delitelja različna od nič. Če ni, potem vrnemo številko napake 4 (deljenje z nič). Takole je videti ta del algoritma:

```
Če predstavlja rpn_i seštevanje:
{
  Postavi a + b na sklad;
}
sicer, če predstavlja rpn_i odštevanje:
{
  Postavi a − b na sklad;
}
```

```
sicer, če predstavlja rpn_i množenje:
{
  Postavi a × b na sklad;
}
sicer, če predstavlja rpn_i deljenje:
{
  Če je b ≠ 0:
  {
    Postavi a/b na sklad;
  }
  sicer:
  {
    Izhod:  4 (deljenje z nič);
  }
}
sicer:
{
  Izhod:  3 (nedovoljen simbol);
}
```

Spodaj je prikazan pripadajoč diagram poteka, ki ima spet en sam vhod in izhod, zato ga lahko brez težav vstavimo na mesto ustreznega ukaznega bloka prejšnjega koraka načrtovanja:



Zapišimo zdaj vse skupaj kot podprogram v jeziku JavaScript:

```
function izracunaj(rpn, sklad) {
  var i, a, b;
  sklad.clear();
  for (i = 0; rpn[i] != 0; i = i + 1) {
    if (rpn[i] > 0 && rpn[i] <= 9) {
      sklad.push(rpn[i]);
    }
    else if (sklad.size() > 1) {
      a = sklad.pop();
```

```
      b = sklad.pop();
      if (rpn[i] == '+') {
        sklad.push(a + b);
      }
      else if (rpn[i] == '-') {
        sklad.push(a - b);
      }
      else if (rpn[i] == '*') {
        sklad.push(a * b);
      }
      else if (rpn[i] == '/') {
        if (b != 0) {
          sklad.push(a / b);
        }
        else {
          return 4; //Napaka:  deljenje z nič
        }
      }
      else {
        return 3; //Napaka:  nedovoljen simbol
      }
    }
    else {
      return 2; //Napaka:  premalo operandov
    }
  }
  if (sklad.size() != 1) {
    return 1; //Napaka:  preveč operandov ali prazen izraz
  }
  else {
    return 0; //brez napake
  }
}
```

Delovanje podprograma preverimo z izrazom z začetka tega razdelka:

```
rpn = [1, 4, 8, '+', 3, '*', '-', 0];
if (izracunaj(rpn, sklad) == 0) {
  console.log(sklad.pop()); //izpiše 35
}
```

Preizkusimo podprogram še z napačnim vhodnim izrazom. Nadomestimo tretji element izraza `rpn` z vprašajem, in podprogram `izracunaj` nam vrne kodo napake:

```
rpn = [1, 4, '?', '+', 3, '*', '-', 0];
console.log(izracunaj(rpn, sklad)); //izpiše 3 (nedovoljen simbol)
```

Če zdaj iz izraza odstranimo prvi element, dobimo drugačno kodo napake:

```
rpn = [4, '?', '+', 3, '*', '-', 0];
console.log(izracunaj(rpn, sklad)); //izpiše 2 (premalo operandov)
```

Glede na to, da naš podprogram predpostavlja, da vsi uporabljeni operatorji potrebujejo dva operanda, sta zdaj v gornjem izrazu `rpn` dve napaki: poleg nedovoljenega simbola je v določenem trenutku na skladu tudi premalo operandov. Naš algoritem je napisan tako, da drugo od obeh napak zazna pred prvo in jo tudi edino javi. To seveda ni tako zelo

pomembno. Pomembno je, da napako zazna in uporabnika podprograma vsaj približno usmeri nanjo, da jo bo lažje poiskal.

5.4 Problem dveh trojk in dveh osmic

Podprogram za računanje vrednosti izraza RPN je najpomembnejša komponenta, ki smo jo potrebovali za rešitev problema dveh trojk in dveh osmic. Zdaj lahko nadaljujemo z načrtovanjem algoritma, ki bo ta problem rešil. Na strani 69 smo že zastavili prvi korak načrtovanja:

> Preglej vse možne izraze, ki jih lahko sestaviš iz dveh trojk in dveh osmic ter osnovnih matematičnih operatorjev, in vrni tistega, katerega vrednost je 24;

Če uporabljamo izključno takšne operatorje, ki potrebujejo dva operanda, potem je potreben (ne pa tudi zadosten) pogoj, da je število operatorjev v izrazu za ena manjše od števila operandov. Ker iščemo izraz s štirimi operandi (t.j. z dvema trojkama in dvema osmicama), potrebujemo v izrazu še tri operatorje, kar da skupaj sedem elementov. Če bomo torej generirali vse možne izraze RPN dolžine sedem iz simbolov iz naslednje tabele:

$$\boldsymbol{simboli} = [3, 8, +, -, *, /],$$

potem bodo med tako generiranimi izrazi tudi vsi možni pravilni izrazi z dvema trojkama in dvema osmicama. Ostale bomo enostavno zavrgli.

Naloga, da moramo ustvariti vse možne izraze, se zdi na prvi pogled težja, kot je v resnici. Gre namreč za običajno štetje po modulu šest, le da namesto števk med nič in pet uporabimo simbole iz tabele $\boldsymbol{simboli}$:

```
3333333  3333383  33333+3          //////3
3333338  3333388  33333+8          //////8
333333+  333338+  33333++    ...   //////+
333333-  333338-  33333+-          //////-
333333*  333338*  33333+*          //////*
333333/  333338/  33333+/          ///////
```

Takšno kombiniranje simbolov se v matematiki imenuje ***permutacije s ponavljanjem***. V tem hipu že lahko zapišemo naslednji korak načrtovanja:

```
Ponovi za vse možne permutacije s ponavljanjem dolžine 7,
sestavljene iz simbolov 3,8,+,-,* in /:
{
  Shrani permutacijo simbolov v tabelo rpn;
  Če je vrednost izraza v tabeli rpn enaka 24
  in število trojk v njem enako 2:
  {
    Sporoči:  rpn;
  }
}
```

Vse permutacije simbolov iz tabele $\boldsymbol{simboli}$ dobimo tako, da najprej ustvarimo vse permutacije števk med nič in pet (gre za običajno štetje po modulu šest):

```
0000000  0000010          5555550
0000001  0000011          5555551
0000002  0000012    ...   5555552
0000003  0000013          5555553
0000004  0000014          5555554
0000005  0000015          5555555
```

Te permutacije nato uporabimo kot indekse za odčitavanje simbolov iz tabele $\boldsymbol{simboli}$. Vedeti moramo še, da je vseh takšnih permutacij 6^7, in že lahko zapišemo naslednji korak načrtovanja:

```
Vhod:  simboli ← [3, 8, +, −, ∗, /], stSimbolov ← 6, dolzina ← 7;
Izhod:  Izraz rpn, katerega vrednost je 24,
        in je sestavljen iz elementov iz tabele simboli;
Ponovi za vsak i od 0 do stSimbolov^dolzina − 1 s korakom 1:
{
  Zapiši i v številskem sistemu z osnovo stSimbolov in
  posamezne števke shrani v tabelo indeksi dolžine dolzina;
  rpn ← [0];
  Ponovi za vsak j od 0 do dolzina − 1 s korakom 1:
  {
    k ← indeksi_j;
    Tabeli rpn dodaj element simboli_k;
  }
  Če je vrednost izraza rpn enaka 24 in število trojk v njem enako 2:
  {
    Sporoči:  rpn;
  }
}
```

Gornjemu algoritmu smo dodali še informacijo o vhodnih podatkih, ki jih potrebujemo za njegovo delovanje, in opis zahtevanega izhoda.

Z načrtovanjem smo prišli že tako daleč, da lahko algoritem zapišemo v JavaScriptu:

```
simboli = [3, 8, '+', '-', '*', '/'];
stSimbolov = 6;
dolzina = 7;
for (i = 0; i < Math.pow(stSimbolov, dolzina); i = i + 1) {
  indeksi = [];
  ustvariIndekse(i, stSimbolov, dolzina, indeksi);
  rpn = [0];
  for (j = 0; j < dolzina; j = j + 1) {
    k = indeksi[j];
    dodajElement(rpn, simboli[k]);
  }
  if (izracunaj(rpn, sklad) == 0) {
    if (sklad.pop() == 24 && prestej(rpn, 3) == 2) {
      console.log(rpn);
    }
  }
}
```

V programu smo uporabili tri funkcije, ki jih moramo še napisati, vendar z njimi ne bomo imeli prav veliko dela. Psevdo kodo za funkcijo `dodajElement` smo že zapisali v besedilu naloge 4.2 na strani 59. V JavaScriptu je videti funkcija takole:

```
function dodajElement(tab, el) {
  var i;
  for (i = 0; tab[i] != 0; i = i + 1) {}
  tab[i] = el;
  tab[i + 1] = 0;
}
```

Funkcija `ustvariIndekse` ni nič drugega kot podprogram, ki pretvarja vrednost iz desetiškega zapisa v zapis z osnovo `stSimbolov`. Tudi s takim problemom smo se že ukvarjali, in sicer na strani 26, kjer smo pretvarjali desetiški zapis v dvojiškega. Takole je videti funkcija, zapisana v jeziku JavaScript:

```
function ustvariIndekse(n, osnova, dolzina, tab) {
  var i;
  for (i = dolzina - 1; i >= 0; i = i - 1) {
    tab[i] = n % osnova;
    n = n - (n % osnova);
    n = n / osnova;
  }
}
```

Funkcija `ustvariIndekse` predstavlja bolj splošno rešitev problema kot pa algoritem, ki smo ga sestavili na strani 26. Tam sta bila številski sistem, v katerega smo vrednost pretvarjali, in dolžina pretvorjenega zapisa konstantna, tu pa ju podajamo s spremenljivima parametroma `osnova` in `dolzina`.

Potrebujemo še funkcijo `prestej`, ki prešteje, kolikokrat se v podani tabeli (parameter `tab` v spodnji funkciji) pojavi podani element (parameter `el` v spodnji funkciji). Tudi ta funkcija je sila preprosta, saj samo primerja vsak element tabele z iskano vrednostjo in vsakokrat, ko sta vrednosti enaki, ustrezno poveča števec:

```
function prestej(tab, el) {
  var i, st;
  st = 0;
  for (i = 0; tab[i] != 0; i = i + 1) {
    if (tab[i] == el) {
      st = st + 1;
    }
  }
  return st;
}
```

Ko vse skupaj sestavimo in zaženemo v brskalniku, v konzoli ne dobimo ničesar. Vrednost izraza, ki reši problem dveh trojk in dveh osmic, je natanko 24. Vendar je ta vrednost natančna samo teoretično, računalnik pa je sposoben izračunati zgolj njen približek. Med računanjem se namreč izvede deljenje $8/3$, katerega rezultat je lahko natančen le do omejenega števila decimalnih mest. Nimamo namreč na razpolago neskončno mnogo pomnilnika. V čisto zadnjem stavku `if` našega programa (na strani 84) moramo zato preverjanje enakosti nadomestiti s preverjanjem, ali vrednost leži v določenem majhnem območju okrog 24:

```
//...
if (izracunaj(rpn, sklad) == 0) {
  rezultat = sklad.pop();
```

```
    if (rezultat < 24.001 && rezultat > 23.999 &&
      prestej(rpn, 3) == 2) {
      console.log(rpn);
    }
  }
  //...
```

Zdaj končno dobimo rešitev problema v obliki tabele:

$$\boldsymbol{rpn} = [3, 8, /, 3, -, 8, /, 0],$$

ki jo v klasičnem matematičnem zapisu predstavlja naslednja formula:

$$\frac{8}{3 - \frac{8}{3}}.$$

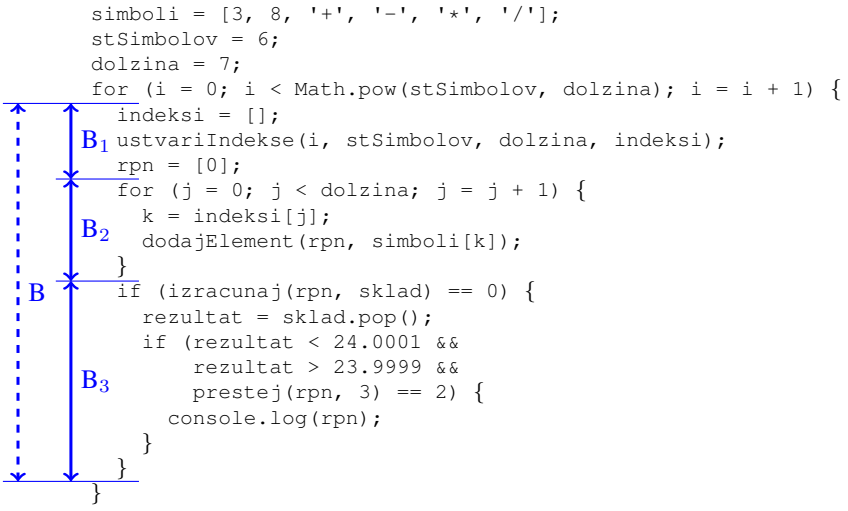
5.5 Nekaj osnovnih napotkov

Za konec poglavja o načrtovanju si oglejmo še nekaj splošnih priporočil, ki naj bi jih programer pri svojem delu upošteval.

5.5.1 Videz kode

Čeprav je za delovanje programa popolnoma nepomembno, kako je videti programska koda, pa je njen videz izjemnega pomena v fazi pisanja programa in kasneje v fazi njegovega vzdrževanja in nadgrajevanja. Najpomembnejša dejavnika pri oblikovanju kode sta način, kako zamikamo vrstice kode (angl. indentation) in kako izbiramo imena spremenljivk.

Zamiki programske kode Za primer vzemimo kodo, ki reši problem dveh trojk in dveh osmic s strani 84:

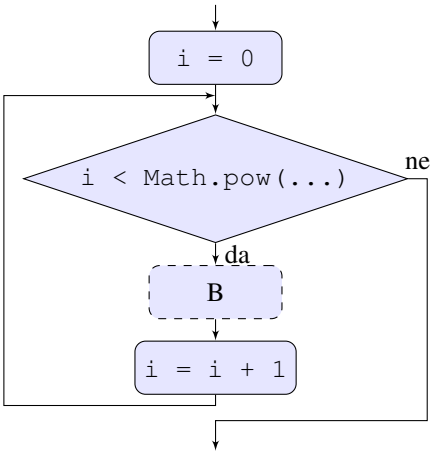



```
simboli = [3, 8, '+', '-', '*', '/'];
stSimbolov = 6;
dolzina = 7;
for (i = 0; i < Math.pow(stSimbolov, dolzina); i = i + 1) {
  indeksi = [];
  ustvariIndekse(i, stSimbolov, dolzina, indeksi);
  rpn = [0];
  for (j = 0; j < dolzina; j = j + 1) {
    k = indeksi[j];
    dodajElement(rpn, simboli[k]);
  }
  if (izracunaj(rpn, sklad) == 0) {
    rezultat = sklad.pop();
    if (rezultat < 24.0001 &&
        rezultat > 23.9999 &&
        prestej(rpn, 3) == 2) {
      console.log(rpn);
    }
  }
}
```

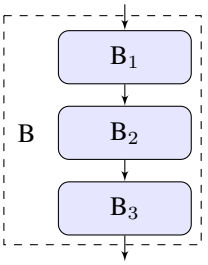
Bodite pozorni na to, kako so določeni deli kode zamaknjeni v desno. Opazite lahko tudi, da so ti zamiki različni. Zamike uporabimo, da z njimi grafično ponazorimo strukturo

programske kode, zaradi česar je koda preglednejša in razumljivejša. Tako oblikovano kodo je lažje vzdrževati in nadgrajevati, že med njenim pisanjem pa se pomembno zmanjša verjetnost za napake.

Osnovni trik je ta, da dele kode, ki se izvajajo zaporedno, pišemo enega pod drugim. Dele kode, ki predstavljajo (odvisni) del kompleksnejšega (ponavljalnega ali odločitvenega) stavka, zamaknemo nekoliko bolj v desno, s čimer nakažemo odvisnost od določenega pogoja. Tako na gornjem primeru hitro vidimo, da je sestavljen iz štirih zaporednih blokov: treh enostavnih priredilnih stavkov in enega ponavljalnega stavka `for`. Vsa koda znotraj tega stavka `for` je zamaknjena za eno stopnjo bolj proti desni, kar kaže, da je to odvisni ukazni blok znotraj zanke `for`. V gornjem programu je ta blok označen z modro črko B, njegov položaj znotraj zanke `for` pa vidimo tudi na naslednjem diagramu poteka (črtkani ukazni blok B):



Naprej vidimo, da je ukazni blok B spet sestavljen iz treh zaporednih blokov B_1, B_2 in B_3. Ti so označeni tako v gornjem programu kot tudi v naslednjem diagramu poteka:



Tudi nadaljnje členjenje kode je lepo razvidno iz njenih zamikov. Blok B_1 vsebuje tri vrstice, ki so vse enako zamaknjene, kar pomeni, da se izvedejo ena za drugo. Blok B_2 ima dve vrstici zamaknjeni še bolj v desno, kar izraža dejstvo, da sta ti dve vrstici del odvisnega ukaznega bloka notranje zanke `for`. Pri zadnjem od treh blokov (B_3) opazimo še en dodaten nivo zamika, kar nakazuje, da je klic funkcije `console.log` odvisni ukazni blok stavka `if`, ki je sam odvisni ukazni blok še enega stavka `if`.

Če isti program zapišemo brez zamikov, postane koda za človeka precej težje berljiva:

```
simboli = [3, 8, '+', '-', '*', '/'];
stSimbolov = 6;
```

```
dolzina = 7;
for (i = 0; i < Math.pow(stSimbolov, dolzina); i = i + 1) {
indeksi = [];
ustvariIndekse(i, stSimbolov, dolzina, indeksi);
rpn = [0];
for (j = 0; j < dolzina; j = j + 1) {
k = indeksi[j];
dodajElement(rpn, simboli[k]);
}
if (izracunaj(rpn, sklad) == 0) {
rezultat = sklad.pop();
if (rezultat < 24.0001 &&
rezultat > 23.9999 &&
prestej(rpn, 3) == 2) {
console.log(rpn);
}
}
}
```

Težave imamo že s tem, da ugotovimo, kje se začnejo in končajo posamezni krmilni stavki `for` in `if`.

Imena spremenljivk in podprogramov Tudi izbira imen spremenljivk in podprogramov igra pomembno vlogo pri razumljivosti kode. Smiselno je izbirati imena, ki sama od sebe govorijo o tem, kaj spremenljivka hrani, oziroma o tem, kaj podprogram počne. Isti program kot zgoraj bi lahko zapisali tudi takole:

```
aaa = [3, 8, '+', '-', '*', '/'];
bbb = 6;
ccc = 7;
for (i = 0; i < Math.pow(bbb, ccc); i = i + 1) {
  ddd = [];
  neki(i, bbb, ccc, ddd);
  eee = [0];
  for (j = 0; j < ccc; j = j + 1) {
    k = ddd[j];
    nekiDruzga(eee, aaa[k]);
  }
  if (seNeki(eee, fff) == 0) {
    aa = fff.test();
    if (aa < 24.0001 &&
        aa > 23.9999 &&
        naredi(eee, 3) == 2) {
      console.log(eee);
    }
  }
}
```

Če bi na enak način spremenili še definicije uporabljenih funkcij, bi program deloval popolnoma enako kot prej. Pisanje takšnega programa – še bolj pa njegovo kasnejše vzdrževanje in nadgrajevanje – pa bi postalo zelo zahtevna in napakam podvržena naloga.

Imena spremenljivk in podprogramov so pogosto sestavljena iz več besed, vendar presledkov v imenih v večini programskih jezikov ne moremo uporabljati. Branjevečbesedbrezpresledkovjesevedatežavno. Zato za ločevanje besed namesto presledkov uporabljamo različne pristope, od katerih sta najpogostejša uporaba znaka podčrtaj (_) (angl. underscore) ali uporaba tako imenovanih ***kameljihČrk*** (angl. camelCase). Običajno v zvezi s

pisavami govorimo o mali črkah (angl. lower case) ali velikih črkah (angl. upper case). PisavakamelječrkeČrke je dobila ime po grbi, ki jo predstavlja velika črka na sredini sestavljenega imena. V resnici gre za način pisanja več besed brez presledkov, kjer za ločevanje besed namesto presledka uporabimo veliko črko vsake naslednje besede. V tem gradivu vsa imena spremenljivk in podprogramov zapisujemo v kameljihČrkah.

5.5.2 Dokumentiranje kode

Tudi dokumentiranje kode ni pomembno za njeno delovanje, je pa ključno za programerje, vzdrževalce in uporabnike. V grobem ločimo dva nivoja dokumentiranja: opombe, ki jih pišemo neposredno v programski kodi, ter ločeno pisanje dokumentacije o programu.

Opombe Opombe, ki jih vstavljamo neposredno v programsko kodo, pišemo bodisi tik ob kodi, na katero se nanašajo, bodisi na začetku določene komponente, kot sta na primer podprogram ali objekt. Opombe so lahko kratka pojasnila o tem, čemu določen odsek kode služi, kadar iz same kode to ni povsem razvidno. Opombe so lahko tudi pojasnila o načinu klicanja podprograma, o tem, kakšne pogoje morajo izpolnjevati parametri, ki jih podajamo, in podobno. S primernim poimenovanjem spremenljivk in podprogramov lahko v določeni meri zmanjšamo potrebo po dodatnih opombah.

Obsežnejše dokumentiranje programske kode V mnogih primerih (npr. ob izdaji novega izdelka na trg) je treba programe opremiti z obsežnejšo dokumentacijo. Ta je lahko namenjena uporabnikom in/ali vzdrževalcem. Za pisanje takšne dokumentacije je potrebno precej znanja in izkušenj, v pomoč pa so nam lahko številna orodja za pisanje programske dokumentacije (angl. software documentation tools).

5.5.3 Omejevanje območja spremenljivk

Za dober nadzor nad napisano kodo je zelo pomembno, da karseda omejimo območje delovanja posameznih spremenljivk. Uporabljamo lokalne spremenljivke, s podprogrami pa komuniciramo prek njihovih parametrov ali stavkov `return`. S tem bomo v veliki meri preprečili možnost nenamerne spremembe vrednosti kakšne ključne spremenljivke. Delo s spremenljivkami lahko nadzorujemo tudi tako, da določene spremenljivke, ki smiselno sodijo skupaj, skrijemo v objekte in z njimi upravljamo preko za to predvidenih postopkov.

5.5.4 Načrtovanje in preizkušanje

Prva misel naj bo tudi zadnja: preden začnemo s pisanjem kode, se prepričamo, da dobro poznamo problem, ki je pred nami. Dogovorimo se o vseh podrobnostih glede zahtev, komu ali čemu bo izdelek namenjen, kako naj deluje, in še mnogo drugega. Če je le mogoče, posamezne komponente preizkusimo posebej. Šele ko se prepričamo, da delujejo zadovoljivo, jih vgradimo v kompleksnejši sitem, ki ga znova preizkusimo.

5.6 Naloge

Za vajo rešite še naslednje naloge:

Naloga 5.2 Načrtujte podprogram obrniElemente, ki obrne vrstni red elementov na položajih od a do vključno $b-1$ v podani tabeli, zaključeni s čuvajem (0). Predpostavite, da velja neenakost $0 \leq a < b$. Če je b večji od števila elementov v tabeli, potem naj podprogram obrne vrstni red elementov od a-tega do zadnjega elementa. Če je a večji od indeksa zadnjega elementa v tabeli, naj podprogram ne naredi ničesar. Za rešitev problema uporabite sklad.

Primer klicev podprograma:

```
tabela = [3, 4, 5, 6, 7, 0];
obrniElemente(tabela, 0, 10, sklad);
console.log(tabela);  //izpiše [7, 6, 5, 4, 3, 0]
tabela = [3, 4, 5, 6, 7, 0];
obrniElemente(tabela, 0, 2, sklad);
console.log(tabela);  //izpiše [4, 3, 5, 6, 7, 0]
tabela = [3, 4, 5, 6, 7, 0];
obrniElemente(tabela, 1, 4, sklad);
console.log(tabela);  //izpiše [3, 6, 5, 4, 7, 0]
tabela = [3, 4, 5, 6, 7, 0];
obrniElemente(tabela, 1, 2, sklad);
console.log(tabela);  //izpiše [3, 4, 5, 6, 7, 0]
```

Naloga 5.3 Velikost sklada je v praksi navadno omejena. V primeru, da je sklad poln (t.j. v njem je največje dovoljeno število elementov), bo operacija push povzročila ***prekoračitev*** ali ***preliv sklada*** (angl. stack overflow). Prav tako se lahko zgodi, da izvedemo operacijo pop na praznem skladu. Takšen klic povzroči ***spodnjo prekoračitev sklada*** (angl. stack underflow).

Dopolnite objekt `sklad` na strani 75 z lastnostjo `maksVelikost`, ki jo nastavite na vrednost tri. Postopkoma `push` in `pop` dodajte kodo, ki bo preverjala, ali je prišlo do preliva oziroma spodnje prekoračitve sklada. Postopek `push` naj v primeru, ko pride do preliva, v konzolo izpiše opozorilo *Stack overflow*. Podani element (ki bi ga moral potisniti na sklad) naj preprosto ignorira. Postopek `pop` naj v primeru spodnje prekoračitve v konzolo prav tako izpiše sporočilo o tem dogodku: izpiše naj opozorilo *Stack underflow*. Poleg tega naj postopek `pop` v primeru spodnje prekoračitve vrne vrednost nič (namesto vrednosti, ki je na vrhu sklada). Delovanje tako dopolnjenega sklada preizkusite z naslednjim programom:

```
sklad.push('a');            //ne izpiše ničesar
sklad.push('b');            //ne izpiše ničesar
sklad.push('c');            //ne izpiše ničesar
sklad.push('d');            //izpiše Stack overflow
console.log(sklad.size());  //izpiše 3
console.log(sklad.pop());   //izpiše c
console.log(sklad.pop());   //izpiše b
console.log(sklad.pop());   //izpiše a
console.log(sklad.pop());   //izpiše Stack underflow in potem 0
console.log(sklad.size());  //izpiše 0
```

Naloga 5.4 ***Diagonalni latinski kvadrat*** je latinski kvadrat (glej nalogo 3.14 na strani 44), ki se mu noben element ne ponovi niti v kateri od njegovih dveh diagonal. Obstaja preprost algoritem, ki sestavi diagonalni latinski kvadrat velikosti $n \times n$ elementov. Algoritem deluje, če je n liho število, ki ni deljivo s tri. V psevdo jeziku ga lahko zapišemo takole:

```
Ponovi za vsak element dvorazsežnostne kvadratne
tabele diagLatin velikosti n × n:
{
  diagLatin_{i,j} ← (2 i + j)%n + 1;
}
```

Napišite podprogram, ki kot prvi parameter sprejme liho celoštevilsko vrednost n, ki ne sme biti deljiva s tri. Podprogram naj kot vhodno izhodni parameter sprejme še tabelo $\boldsymbol{diagLatin}$, v katero naj zapiše diagonalni latinski kvadrat velikosti $n \times n$ elementov.

Naloga 5.5 ***Ortogonalni latinski kvadrat*** je v kombinatoriki latinski kvadrat velikosti $n \times n$ (glej nalogo 3.14 na strani 44), sestavljen iz elementov dveh množic (enakih ali različnih), od katerih vsaka vsebuje n različnih simbolov. Vsako polje ortogonalnega latinskega kvadrata vsebuje urejen par elementov, kjer je prvi iz ene, drugi pa iz druge množice. Vsaka vrstica in vsak stolpec lahko vsebujeta samo po en element iz vsake od obeh množic, poleg tega nobeni dve polji ne smeta vsebovati enakega urejenega para. Tako je videti primer ortogonalnega latinskega kvadrata velikosti 3×3:

Aa	Bc	Cb
Bb	Ca	Ac
Cc	Ab	Ba

Ortogonalni latinski kvadrat se imenuje tudi ***Eulerjev kvadrat***. Ker se namesto malih latinskih črk za označevanje elementov druge množice v literaturi včasih uporabljajo grške črke, pravimo takemu kvadratu tudi ***grško latinski kvadrat***.

Zanimivost: Ortogonalne latinske kvadrate uporabljamo pri načrtovanju eksperimentov in sestavljanju urnikov športnih turnirjev.

Obstaja preprost algoritem, s katerim je mogoče sestaviti ortogonalni latinski kvadrat ($\boldsymbol{ortogLatin}$) iz diagonalnega latinskega kvadrata ($\boldsymbol{diagLatin}$, glej nalogo 5.4 na strani 91). To storimo tako, da vsak element diagonalnega latinskega kvadrata kombiniramo z elementom, ki je glede nanj zrcaljen preko diagonale kvadrata:

```
Ponovi za vsak element dvorazsežnostne kvadratne
tabele ortogLatin velikosti n × n:
{
  ortogLatin_{i,j} ← [diagLatin_{i,j}, diagLatin_{j,i}];
}
```

Napišite podprogram, ki kot prvi parameter sprejme liho celoštevilsko vrednost n, ki ne sme biti deljiva s tri. Podprogram naj kot vhodno izhodni parameter sprejme še tabelo $\boldsymbol{ortogLatin}$, v katero naj zapiše ortogonalni latinski kvadrat velikosti $n \times n$.

Naloga 5.6 ***Magični kvadrat*** je v razvedrilni matematiki in kombinatoričnem načrtovanju kvadratno polje dimenzije $n \times n$, napolnjeno s celimi števili, običajno od ena do n^2. Pri tem mora vsako polje vsebovati različno število, vsote števil v vsaki od vrstic, stolpcev in obeh diagonal pa morajo biti enake. Enega najbolj znanih magičnih kvadratov najdemo na grafiki ***Melanholija I*** nemškega slikarja Albrechta Dürerja, kjer je vsota števil v vsaki od vrstic, stolpcev in obeh diagonal enaka 34[a]:

Magični kvadrat lahko dobimo podobno kot ortogonalni latinski kvadrat (glej nalogo 5.5 na strani 91). Če izhajamo iz diagonalnega latinskega kvadrata $\boldsymbol{diagLatin}$ velikosti $n \times n$ elementov, ki vsebuje števila v območju $1, 2, \cdots, n$, potem dobimo magični kvadrat velikosti $n \times n$ po naslednjem algoritmu:

```
Ponovi za vsak element dvorazsežnostne kvadratne
tabele magicniKvadrat velikosti n × n:
{
  magicniKvadrat_{i,j} ← n × (diagLatin_{i,j} − 1) + diagLatin_{j,i};
}
```

Napišite podprogram, ki kot prvi parameter sprejme liho celoštevilsko vrednost n, ki ne sme biti deljiva s tri. Podprogram naj kot vhodno izhodni parameter sprejme še tabelo $\boldsymbol{magicniKvadrat}$, v katero naj zapiše magični kvadrat velikosti $n \times n$.

Opomba: V magičnem kvadratu lahko med seboj zamenjamo poljubni dve vrstici. Če nato med seboj zamenjamo še dva stolpca z istima zaporednima številkama, kot sta bili zaporedni številki zamenjanih vrstic, dobimo drugačen kvadrat, ki je tudi magični kvadrat. Poskusite pojasniti, zakaj je tako, in napišite program, ki poišče čim več različnih magičnih kvadratov.

[a] Ta vsota je odvisna zgolj od dimenzije kvadrata in jo lahko izračunamo kot $n/2\,(n^2+1)$. To seveda velja ob predpostavki, da so v kvadratu vpisana cela števila od ena do n^2.

Naloga 5.7 V mnogih računalniških sistemih sta datum in ura zapisana z eno samo celoštevilsko vrednostjo, ki predstavlja število sekund, ki so pretekle od polnoči 1.

januarja 1970 po ***univerzalnem usklajenem času*** (angl. coordinated universal time, UTC). Takšnemu načinu zapisovanja časa pravimo tudi ***Unixov čas*** (angl. Unix time).

Načrtujte in izdelajte objekt `datum` s postopkoma `nastavi` in `vrniUro`. Prvi postopek naj v objekt shrani število sekund, ki so pretekle od polnoči 1. januarja 1970. Drugi postopek naj iz podanih sekund izračuna uro (ure, minute in sekunde) v skladu z dogovorom UTC ter jo vrne preko vhodno izhodnega parametra (tabele s tremi elementi). Za preizkušanje objekta lahko uporabite funkcijo jezika JavaScript `Date.now`, ki vrne število pretečenih milisekund po Unixovem času. Dobljeno vrednost delite s 1000 in zaokrožite navzdol (s funkcijo `Math.floor`), da iz milisekund dobite sekunde.

Takole preizkusite izdelan objekt:

```
d = [];
datum.nastavi(Math.floor(Date.now() / 1000));
datum.vrniUro(d);
console.log(d);    //izpiše npr.  [9, 55, 19]
```

Opomba: Slovenija se nahaja v časovnem pasu, ki je eno uro pred UTC-jem (t.j. UTC+1), zato moramo dobljenemu času prišteti eno uro, da dobimo pravi čas. Gornji primer je bil torej zagnan ob 10:55:19.

Pomoč: Najprej izračunajte, koliko sekund je minilo od zadnje polnoči. Ker je vsak dan dolg 86 400 sekund (prestopne sekunde se tu ne upoštevajo), lahko to izračunate kot ostanek pri deljenju s 86 400. Iz dobljenega ostanka dobite ure, minute in sekunde z ustreznimi deljenji, zaokroževanji navzdol in računanji ostankov pri deljenju.

Naloga 5.8 Dopolnite objekt `datum` iz prejšnje naloge s postopkom `vrniDan`. Postopek naj preko parametra (tabele s tremi elementi) sporoči datum v obliki treh celoštevilskih vrednosti: dan, mesec in leto.

Takole preizkusite delovanje postopka:

```
d = [];
datum.nastavi(Math.floor(Date.now() / 1000));
datum.vrniDan(d);
console.log(d);    //izpiše npr.  [28, 11, 2020]
```

Pomoč: Letnico dobite tako, da od skupnega števila sekund odštevate dolžino enega leta (v sekundah), pri čemer morate upoštevati, da so prestopna leta za 86 400 sekund daljša od običajnih. To počnete toliko časa, dokler ne dobite ostanka, ki je krajši od dolžine enega leta. Od preostalih sekund na podoben način odštevate dolžine posameznih mesecev (lahko jih tabelirate), pri čemer morate v primeru prestopnega leta za februar upoštevati dodaten dan. Na koncu od preostalih sekund odštevate posamezne dneve (86 400).

PRAZNA STRAN

6. POGLAVJE

PODATKOVNE STRUKTURE IN ABSTRAKTNI PODATKOVNI TIPI

Omenili smo že, da ***abstrakten podatkovni tip*** (angl. abstract data type) predstavlja matematični model, ki poleg podatkov (t.j. množice dovoljenih vrednosti) določa tudi operacije nad temi podatki. Ko opisujemo obnašanje abstraktnega podatkovnega tipa, imamo v mislih uporabnika podatkovnega tipa. Uporabnik mora vedeti zgolj to, kako se abstraktni podatkovni tip uporablja. Ne zanima ga, niti na kakšen način so podatki zapisani niti kakšna koda se skriva v ozadju. Na primer, ko uporabnik sklada razume, kako delujeta njegova postopka push in pop, je to zanj že dovolj, da sklad uporabi pri svojem delu.

Za razliko od uporabnika mora programer, ki bo sklad sprogramiral, vedeti nekoliko več. Poznati mora način, kako sklad izvesti. Mi smo za izvedbo sklada uporabili tabelo, ki v tem primeru predstavlja ***podatkovno strukturo*** (angl. data structure). Za razliko od abstraktnega podatkovnega tipa, ki je usmerjen proti uporabniku, je podatkovna struktura usmerjena k razvijalcu. Podatkovna struktura predstavlja namreč način fizične izvedbe abstraktnega podatkovnega tipa.

V tem poglavju bomo spoznali še dva abstraktna podatkovna tipa in eno podatkovno strukturo, ki jih v praksi pogosto srečamo.

6.1 Slovar (abstrakten podatkovni tip)

Slovar (angl. dictionary) ali ***asociativna tabela*** (angl. associative array) je abstrakten podatkovni tip, sestavljen iz zbirke parov ključev s pripadajočimi vrednostmi (angl. key-value

pairs). V zbirki se lahko vsak ključ pojavi le enkrat. Poleg tega predvideva slovar naslednje operacije:

- dodajanje para v zbirko (angl. insert);
- brisanje para iz zbirke (angl. delete);
- spreminjanje para v zbirki (angl. modify);
- iskanje vrednosti, povezane s ključem (angl. lookup).

Slovar na določen način spominja na tabelo, le da omogoča mnogo širšo uporabo. Tabela povezuje indekse in vrednosti, ki so shranjene v njej: če poznamo indeks, lahko hitro pridemo do vrednosti ustreznega elementa. V slovarju namesto indeksa uporabljamo ključ, katerega vrednost ni omejena na nenegativna cela števila. V slovarju niti ni potrebno, da so pari kakorkoli urejeni, prav tako ni potrebno, da si ključi sledijo po vrsti. Tako lahko na primer v slovar shranimo para k in 1 000 ter M in 1 000 000 za desetiške predpone kilo in mega. Vsak par lahko shranimo celo dvakrat, kjer je ključ enkrat črka drugič pa številka. Tako lahko po slovarju iščemo po obeh kriterijih.

Od štirih operacij nad slovarjem, ki smo jih našteli malo prej, bomo za začetek izpustili brisanje para iz zbirke. Ker se lahko vsak ključ v slovarju pojavi le enkrat, dodajanje novega para s ključem, ki je že v slovarju, ni dovoljeno. Ta omejitev nas po krajšem razmisleku pripelje do zamisli, da lahko dodajanje in spreminjanje para združimo v eno samo operacijo: če ključ v paru, ki ga želimo dodati, v slovarju že obstaja, enostavno popravimo obstoječo vrednost v slovarju. V nasprotnem primeru dodamo nov par na konec slovarja.

Naloga, ki je pred nami, ni prav zapletena, zato lahko kar zapišemo programsko kodo za naš slovar:

```
slovar = Object();
slovar.podatki = [];  //tabela s podatki
slovar.konec = 0;     //prvo prosto mesto na koncu tabele

slovar.insert = function(kljuc, vrednost) {
  var i;
  for (i = 0; i < slovar.konec; i = i + 1) {
    if (slovar.podatki[i][0] == kljuc) { //Če smo našli ključ,
      slovar.podatki[i][1] = vrednost;   //popravimo njegovo vre-
      return;                            //dnost in končamo izva-
                                         //janje postopka.
    }
  }
  slovar.podatki[i] = [kljuc, vrednost]; //Če ključa nismo našli,
                                         //dodamo nov par in se po-
  slovar.konec = slovar.konec + 1;       //maknemo za mesto naprej.
};

slovar.lookup = function(kljuc) {
  var i;
  for (i = 0; i < slovar.konec; i = i + 1) {
    if (slovar.podatki[i][0] == kljuc) { //Če smo našli ključ, vrne-
      return slovar.podatki[i][1];       //mo pripadajočo vrednost.
    }
  }
  return -1;   //Če ključ ne obstaja, vrnemo -1.
};
```

Podatke slovarja hranimo v dvorazsežnostni tabeli. Tabela `slovar.podatki` je v resnici tabela tabel z dvema elementoma, od katerih je prvi element ključ, drugi element

pa pripadajoča vrednost. Tako predstavlja zapis `slovar.podatki[i][0]` ključ v paru z indeksom `i`. Podobno predstavlja zapis `slovar.podatki[i][1]` vrednost v paru z indeksom `i`. Pomembno je, da ves čas vemo, kje je konec slovarja, zato v lastnosti `slovar.konec` hranimo indeks prvega prostega mesta na koncu slovarja.

V postopku `slovar.insert` smo uporabili stavek `return`, ne da bi v resnici vrnili kakšno vrednost. Ta stavek lahko uporabimo tudi za predčasen izhod iz funkcije, ki ne vrača nobene vrednosti. Če funkcija `insert` ugotovi, da je ključ iz podanega para že v slovarju, potem samo popravi pripadajočo vrednost v slovarju in takoj prekine z delom (`return`). Če ključa v slovarju še ni, potem se bo stavek `for` izvedel do konca. Takoj zatem vstavimo par ključa in vrednosti (v obliki enorazsežnostne tabele z dvema elementoma) na konec slovarja ter povečamo `slovar.konec` za ena. S tem smo povečali število elementov v slovarju za ena.

Postopek `slovar.lookup` je še preprostejši. Če v slovarju najde podani ključ, potem vrne vrednost, ki pripada najdenemu ključu. Če se to ne zgodi, se stavek `for` normalno zaključi in postopek vrne -1. Zavedati se moramo, da smo si s tem omejili možnost, da bi v slovar shranili vrednost -1. Vendar to za nas ne bo težava, ker v primerih, ki sledijo, take vrednosti ne bomo potrebovali.

Preizkusimo zdaj naš slovar za rešitev naslednjega problema: Podana je rimska številka v obliki tabele rimskih cifer, zaključene z ničlo. Napišite program, ki izračuna desetiško vrednost podane rimske številke.

Za rešitev problema potrebujemo najprej slovar, ki nam bo omogočil, da za vsako rimsko cifro neposredno odčitamo njeno vrednost. Takole vstavimo v slovar vse rimske cifre (ključi) s pripadajočimi vrednostmi:

```
rim = ['I', 'V', 'X', 'L', 'C', 'D', 'M', 0];
des = [1, 5, 10, 50, 100, 500, 1000, 0];
for (i = 0; rim[i] != 0; i = i + 1) {
  slovar.insert(rim[i], des[i]);
}
```

S tako zgrajenim slovarjem nas ne čaka več veliko dela. Za vsako cifro v podani rimski številki najprej iz slovarja odčitamo njeno desetiško vrednost. Če ima naslednja cifra v podani rimski številki večjo vrednost od odčitane, potem odčitano vrednost odštejemo od skupne desetiške vsote. V nasprotnem primeru odčitano vrednost prištejemo skupni desetiški vsoti. Takole izgleda dokončan program:

```
letnicaRim = ['M', 'C', 'M', 'X', 'I', 'X', 0];
letnicaDes = 0;
for (i = 0; letnicaRim[i] != 0; i = i + 1) {
  vredn = slovar.lookup(letnicaRim[i]);
  if (vredn < slovar.lookup(letnicaRim[i + 1])) {
    letnicaDes = letnicaDes - vredn;
  }
  else {
    letnicaDes = letnicaDes + vredn;
  }
}
console.log(letnicaDes); //izpiše 1919
```

Za domačo nalogo poskusite rešiti še naslednjo nalogo:

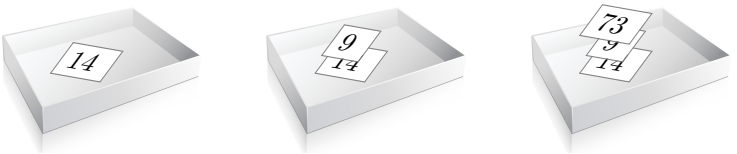
Naloga 6.1 Napišite program, ki izračuna desetiško vrednost števila, zapisanega v šestnajstiškem zapisu. Podano šestnajstiško število je zapisano kot tabela, ki lahko vsebuje desetiške števke in črke od A do F. Tabela naj bo zaključena z -1. Na primer, šestnajstiško (heksadecimalno) število FC03 je podano takole:

```
heks = ['F', 'C', 0, 3, -1];
```

Pomoč: Najprej potrebujete slovar, iz katerega boste odčitavali desetiške vrednosti črk od A do F (10 do 15). Zaradi enostavnosti lahko v slovar vnesete tudi vrednosti od 0 do 9, pri čemer bodo ključi enaki vrednostim. Odčitane desetiške vrednosti morate potem pomnožiti z ustreznimi potencami števila 16 in dobljene zmnožke sešteti. Na primer, šestnajstiško vrednost FC03 pretvorite v desetiško po formuli $15 \times 16^3 + 12 \times 16^2 + 0 \times 16^1 + 3 \times 16^0 = 64\,515$.

6.2 Vrsta (abstrakten podatkovni tip)

Vrsta (angl. queue) je abstrakten podatkovni tip, v katerem na podoben način kot v skladu shranjujemo elemente po vrsti. Razlika je le v tem, da elemente iz vrste odvzemamo na drugem koncu, kakor jih vanjo dodajamo. Naslednja slika prikazuje tri zaporedna stanja, ki jih dobimo, če v vrsto postavimo elemente z vrednostmi 14, 9 in 73:



Element, ki je na sliki čisto spodaj, je prvi na vrsti za branje. Tako je videti vrsta, ko ta element odstranimo:

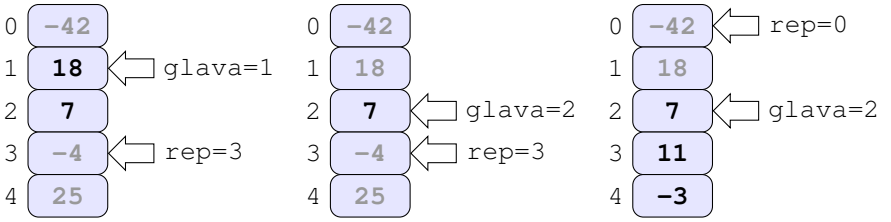
Ker je element, ki ga prvega postavimo v vrsto, tudi prvi na vrsti za branje, spada vrsta med podatkovne tipe vrste FIFO (angl. first in, first out – prvi noter, prvi ven). Operacija dodajanja elementa v vrsto se v angleščini imenuje enqueue, operacija odvzemanja elementa iz vrste pa dequeue.

V resničnem življenju se vrsta uporablja kot ***medpomnilnik*** (angl. buffer), ki hrani podatke za kasnejšo obdelavo. Najpogosteje potrebujemo tak medpomnilnik pri komunikaciji dveh enot, ki med seboj nista popolnoma sinhronizirani. Vzemimo za primer, da si želimo preko spleta ogledati video posnetek. Navadno se del podatkovnega toka pred predvajanjem najprej shrani v medpomnilnik. Predvajalnik nato podatke za predvajanje bere iz medpomnilnika z natanko predpisano hitrostjo. Ta je neodvisna od dejanske hitrosti sprejemanja podatkov s spleta, kar je ključnega pomena za nemoteno predvajanje. Brez

medpomnilnika bi namreč vsaka motnja v sprejemanju podatkov za kratek čas ustavila predvajanje ne glede na to, kako hitra je sicer povezava. Z uporabo medpomnilnika ima predvajalnik vedno na voljo dovolj podatkov, tudi če se vhodni tok za nekaj časa prekine.

Vrsto najenostavneje zgradimo s pomočjo tabele. Ker pa podatke beremo z druge strani, kot jih v vrsto dodajamo, bi morali ob vsakem branju vse elemente tabele premakniti za eno mesto naprej. To je časovno precej potratno, zato uporabimo tako imenovano ***krožno tabelo*** ali ***krožni medpomnilnik*** (angl. circular buffer). Krožna tabela je običajna tabela z vnaprej določeno dolžino, le da na njenem koncu ***ovijemo*** (angl. wrap) operaciji branja in pisanja. V praksi to pomeni, da elementu tabele z najvišjim indeksom sledi element z indeksom nič.

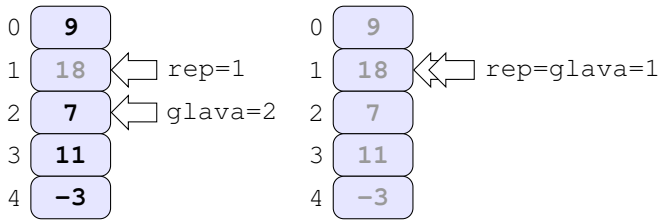
Naslednja slika prikazuje nekaj stanj med pisanjem (enqueue) in branjem (dequeue) iz vrste:



Čisto na levi vidimo stanje, ko sta v vrsti shranjena dva elementa z vrednostma 7 in 18 (prikazana v mastnem tisku, sive vrednosti predstavljajo prosta mesta). Spremenljivka `rep` ima enako funkcijo, kakršno je imela spremenljivka `vrh` pri skladu: hrani indeks prvega prostega mesta v vrsti, kamor lahko vpišemo naslednjo vrednost. Ker podatke beremo z drugega konca, kakor jih pišemo, potrebujemo še eno spremenljivko (`glava`), ki hrani indeks elementa, ki je prvi na vrsti za branje. Če iz vrste na desni strani gornje slike preberemo en podatek, dobimo vrednost 18, vrednost spremenljivke `glava` pa se poveča za ena. Dobimo stanje, ki je prikazano na sredini gornje slike.

Če zdaj v vrsto dodamo dve novi vrednosti 11 in -3, dobimo stanje, ki je prikazano čisto na desni. Ko smo vpisali vrednost -3 na zadnje mesto v tabeli, se je rep ovil na začetek tabele in dobil vrednost nič.

V vrsto lahko zdaj postavimo samo še eno vrednost. Na levi strani naslednje slike vidimo stanje, ki ga dobimo, če v vrsto postavimo vrednost devet:



(polna vrsta) (prazna vrsta)

Ker mora rep ves čas hraniti indeks prostega mesta za vpisovanje, ga ne smemo več povečati. Če bi ga, bi posegli v element, v katerem je vpisan veljaven podatek (t.j. element z indeksom 2 z vrednostjo 7). Kadarkoli torej rep postane za ena manjši od glave, je to znak, da je vrsta polna.

Na koncu preberemo iz vrste vse štiri elemente in dobimo stanje, kakršno je prikazano na desni strani gornje slike. Kadar je vrsta prazna, imata rep in glava enaki vrednosti.

Zapišimo zdaj objekt `vrsta` v jeziku JavaScript:

```
vrsta = Object();
vrsta.podatki = [];
vrsta.rep = 0;
vrsta.glava = 0;
vrsta.dolzina = 5;

vrsta.enqueue = function(el) {
  //Če vrsta še ni polna:
  if ((vrsta.rep + 1) % vrsta.dolzina != vrsta.glava) {
    vrsta.podatki[vrsta.rep] = el;
    vrsta.rep = (vrsta.rep + 1) % vrsta.dolzina;
  }
  else {
    console.log("Overflow:  Queue is full, cannot enqueue.");
  }
};

vrsta.dequeue = function(el) {
  var podatek;
  //Če vrsta ni prazna:
  if (vrsta.rep != vrsta.glava) {
    podatek = vrsta.podatki[vrsta.glava];
    vrsta.glava = (vrsta.glava + 1) % vrsta.dolzina;
    return podatek;
  }
  console.log("Underflow:  Queue is empty, cannot dequeue.");
  return 0;
};
```

V gornji kodi smo ovijanje repa in glave dosegli z operatorjem ostanka pri celoštevilskem deljenju. Kadarkoli vrednost repa ali glave postane enaka dolžini tabele, ta operator poskrbi, da se njuna vrednost ovije na nič. Na primer, izraz `(vrsta.rep + 1) % vrsta.dolzina` vedno vrne indeks elementa, ki v krožni tabeli neposredno sledi repu.

V funkciji `dequeue` smo uporabili pomožno lokalno spremenljivko `podatek`. Vanjo shranimo vrednost elementa, ki je na mestu glave, preden glavo povečamo za ena. Po tem, ko funkcija enkrat vrne želeno vrednost, glave namreč ne moremo več povečati.

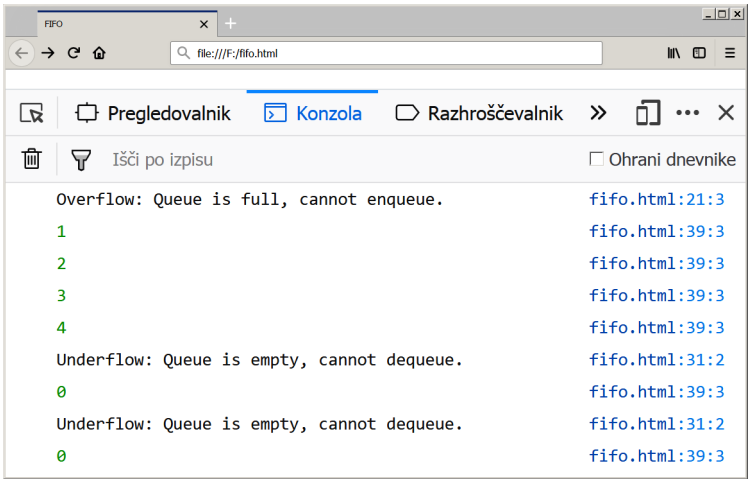
Druga možnost bi bila, da bi najprej povečali glavo za ena in šele potem vrnili vrednost elementa, ki je eno mesto pred indeksom, ki ga hrani glava. Pri tem bi naleteli na manjšo težavo, kajti potrebovali bi ovijanje v obratno smer: Z vrednosti nič bi morala glava dobiti vrednost štiri, kar je indeks predhodnika elementa z indeksom nič. Teoretično bi se moralo dati indeks predhodnega elementa z ovijanjem izračunati na enak način kot indeks naslednjega elementa. Se pravi, izraz `(0 - 1) % 5` bi moral vrniti vrednost štiri. Vendar se to ne zgodi vedno. Spomnimo se, da ostanek pri deljenju negativnih števil ni enoumno določen pri vseh programskih jezikih (glej primer na strani 13). Na primer, v jeziku Python bi dobili za naše potrebe pravilen rezultat, ne pa tudi v jezikih JavaScript ali C.

Oba klica funkcije `console.log` v gornjem programu sta namenjena sledenju pri razhroščevanju kode, ki bo uporabljala naš objekt `vrsta`. Kadarkoli bo koda skušala pisati v polno vrsto oziroma brati iz prazne vrste, bomo dobili v konzoli sporočilo o prelivu (angl. overflow) oziroma spodnji prekoračitvi (angl. underflow) vrste.

Objekt `vrsta` lahko preizkusimo tako, da skušamo v vrsto najprej vpisati nekaj elementov (enega preveč), potem pa jih še nekaj prebrati. Tudi prebrati bomo poskusili več elementov, kot jih je na voljo:

```
for (i = 1; i <= 5; i = i + 1) {
  vrsta.enqueue(i);
}
for (i = 1; i <= 6; i = i + 1) {
  console.log(vrsta.dequeue());
}
```

Ko program zaženemo, dobimo pričakovan izpis:

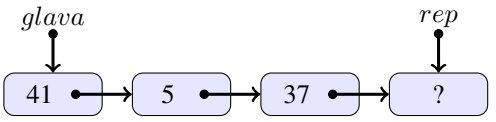


Postopek `enqueue` smo klicali enkrat preveč, zato smo dobili v konzoli najprej sporočilo o prelivu. Številke od ena do štiri, ki sledijo, so posledica prvih štirih klicev postopka `dequeue`. Zadnja dva klica postopka `dequeue` smo izvedli na prazni vrsti, zato smo dobili na koncu dve sporočili o spodnji prekoračitvi. Ob tem se je vsakokrat izpisala ničla, to je namreč vrednost, ki jo naš postopek `dequeue` vrne v primeru, ko je vrsta prazna.

Izvedba vrste s krožno tabelo je sicer enostavna, kompaktna in učinkovita, vendar je neprimerna, kadar imamo opravka z večjimi količinami podatkov. Mehanizem krožne tabele namreč ne omogoča enostavnega dinamičnega spreminjanja kapacitete medpomnilnika med samim delovanjem programa. Zato bi morala takšna tabela ves čas delovanja programa zasedati ogromen del pomnilnika za podatke tudi takrat, ko dejanskih podatkov ni toliko. Če namesto tabele za izvedbo vrste uporabimo povezan seznam, se tej težavi izognemo.

6.3 Povezan seznam (podatkovna struktura)

Vse tri abstraktne podatkovne tipe, ki smo jih spoznali doslej (t.j. sklad, slovar in vrsto), smo izdelali na podlagi tabele. Namesto tabele lahko v ta namen uporabimo tudi ***povezan seznam*** (angl. linked list), ki ga bomo spoznali v tem razdelku. Povezan seznam je zgrajen iz podatkovnih blokov, od katerih vsak (razen zadnjega) vsebuje komponento, ki se sklicuje na naslednji podatkovni blok v seznamu. Posameznemu podatkovnemu bloku povezanega seznama pravimo tudi ***element*** ali ***vozlišče*** (angl. node). Naslednja slika prikazuje primer povezanega seznama, v katerem imamo shranjene tri celoštevilske vrednosti:



Zadnje vozlišče v povezanem seznamu igra vlogo čuvaja. Njegova vrednost večinoma ni pomembna (zato je na sliki označena z vprašajem), pomembna pa je njegova prisotnost, ker marsikdaj poenostavi algoritme, ki upravljajo s podatki v povezanem seznamu. Skupaj z vozlišči povezanega seznama navadno hranimo tudi sklic na prvo ($glava$) in zadnje (rep) vozlišče seznama.

Vozlišče povezanega seznama zapišemo kot programski objekt z dvema lastnostma: $podatek$ in $naslednji$:

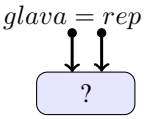
```
Objekti glava, rep, naslednji:
{
  Lastnosti:  podatek, naslednji;
};
```

Lastnost $podatek$ hrani vrednost, ki je shranjena v vozlišču, lastnost $naslednji$ pa je sklic na naslednje vozlišče. Iz gornjega zapisa vidimo, da so na ta način določeni objekti $glava$, rep in $naslednji$. Objekt $naslednji$ je tako določen sam s seboj. Pravimo, da je objekt določen ***rekurzivno*** (t.j. ena od njegovih lastnosti je objekt sam). To ni nič nenavadnega, saj je povezan seznam v osnovi rekurzivna tvorba: vsaj ena od komponent vsakega vozlišča je spet vozlišče oziroma, natančneje, sklic na vozlišče.

Povezan seznam določimo kot objekt z dvema lastnostma (rep in $glava$) ter z vrsto postopkov za dodajanje in odstranjevanje vozlišč na različnih mestih:

```
Objekt seznam:
{
  Lastnosti:  glava, rep;
  Postopki:   insertBeginning,  removeBeginning,   insertEnd,   insertBefore,  delete;
};
```

Lastnosti $glava$ in rep nam popolnoma zadostujeta za dostop do kateregakoli vozlišča v seznamu. Lastnost $glava$ je sklic na prvi element seznama, preko katerega (preko njegove lastnosti $naslednji$) lahko pridemo do drugega elementa. Lastnost $naslednji$ drugega elementa nas nadalje pripelje do tretjega elementa in tako dalje do čuvaja, na katerega se sklicuje lastnost rep. Kadar je seznam prazen, imata lastnosti rep in $glava$ enaki vrednosti, obe pa se sklicujeta na čuvaja, ki je edino vozlišče v seznamu. Prazen seznam prikazuje naslednja slika:

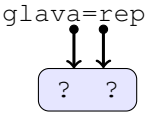


Zapišimo zdaj objekt `seznam` (prazen seznam) v jeziku JavaScript:

```
seznam = Object();
seznam.glava = Object();
seznam.glava.podatek = '?';
```

```
seznam.glava.naslednji = '?';
seznam.rep = seznam.glava;
```

Poleg objekta `seznam` smo ustvarili še objekt `seznam.glava`, ki smo mu dodali lastnosti `podatek` in `naslednji`. Ker želimo ustvariti prazen seznam, je `seznam.glava` v resnici čuvaj. Ker je čuvaj vedno zadnje vozlišče v seznamu (naslednje vozlišče ne obstaja) in ker njegova vrednost ni pomembna, lahko obe njegovi lastnosti nastavimo na kakršnekoli vrednosti. Mi smo ju nastavili na vrednost ?. V zadnji vrstici gornje kode smo sklic na čuvaja, ki je shranjen v lastnosti `seznam.glava`, kopirali še v lastnost `seznam.rep`, ki se zdaj prav tako sklicuje na čuvaja. V tem trenutku imamo stanje, kakršno je prikazano na naslednji sliki:



V vozlišču na sliki sta vpisana dva vprašaja, s čimer smo želeli poudariti dejstvo, da sta obe njegovi lastnosti (tako `podatek` kot tudi `naslednji`) nastavljeni na vrednost ?.

6.3.1 Dodajanje in brisanje vozlišča z začetka seznama

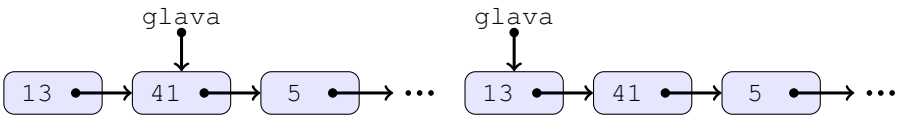
Prva od operacij nad povezanim seznamom, ki jo bomo napisali, je vstavljanje vozlišča na začetek seznama. K objektu `seznam` bomo napisali postopek `insertBeginning`, ki kot parameter sprejme vrednost, ki jo želimo vstaviti na začetek seznama. Takole je videti ta postopek v jeziku JavaScript:

```
seznam.insertBeginning = function(vredn) {
  var nov = Object();
  nov.podatek = vredn;
  nov.naslednji = seznam.glava;
  seznam.glava = nov;
};
```

V prvi vrstici kode ustvarimo objekt `nov`, ki predstavlja novo vozlišče, v drugi vrstici kode pa v njegovo lastnost `podatek` vpišemo podano vrednost. V tretji vrstici kode lastnost `nov.naslednji` usmerimo na vozlišče, ki je bilo do tega trenutka prvo v povezanem seznamu. Ker je sklic na to prvo vozlišče shranjen v lastnosti `seznam.glava`, je dovolj, da enostavno kopiramo vrednost `seznam.glava` v `nov.naslednji`. Čisto na koncu moramo popraviti še vrednost glave, ki mora zdaj kazati na pravkar dodano vozlišče (`seznam.glava = nov`). Odslej je namreč to vozlišče prvo v seznamu.

Ker je spremenljivka `nov` lokalna spremenljivka postopka `insertBeginning`, bi lahko mislili, da se objekt izgubi, ko se postopek zaključi. Spremenljivka `nov` zunaj postopka zares ne obstaja več, vendar to ni težava. Ne smemo pozabiti, da v njej ne hranimo objekta, temveč zgolj sklic na objekt. Ta sklic pa v zadnji vrstici postopka kopiramo v lastnost `seznam.glava`. Pomembno je vedeti, da objekt v pomnilniku obstaja toliko časa, dokler se v programu hrani vsaj ena kopija sklica na ta objekt.

Leva stran spodnje slike prikazuje stanje po tem, ko se izvedejo prve tri vrstice funkcije `seznam.insertBeginning` v primeru, da smo v seznam vpisali vrednost 13. Na desni strani vidimo stanje, ko se izvede še zadnja vrstica kode:

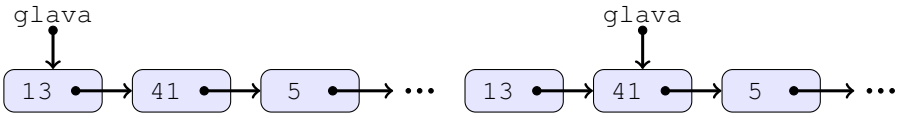


Naslednji postopek odstrani element z začetka seznama in vrne vrednost, ki je v njem zapisana:

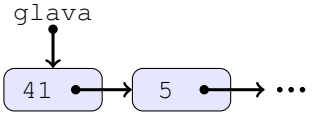
```
seznam.removeBeginning = function() {
  var vrni;
  vrni = seznam.glava.podatek;
  if (seznam.glava != seznam.rep) {
    seznam.glava = seznam.glava.naslednji;
  }
  return vrni;
};
```

V prvih dveh vrsticah ustvarimo lokalno spremenljivko `vrni` in vanjo zapišemo vrednost, ki je zapisana v vozlišču, na katero se sklicuje `seznam.glava`. To lahko naredimo tudi, če je seznam prazen. Takrat bo spremenljivka `vrni` dobila vrednost ?, ki je zapisana v čuvaju. Če seznam ni prazen (t.j. če je izpolnjen pogoj `seznam.glava != seznam.rep`), premaknemo glavo na drugi element seznama. Sklic na drugi element seznama je shranjen v lastnosti `naslednji` prvega elementa, na katerega se še vedno sklicuje glava. Glavo zato premaknemo na drugi element tako, da vanjo zapišemo vrednost spremenljivke `seznam.glava.naslednji`.

Naslednja slika prikazuje stanje pred (levi del slike) in po (desni del slike) brisanju prvega elementa iz seznama. Funkcija `seznam.removeBeginning` bi v tem primeru vrnila vrednost 13:



Spremenljivka `seznam.glava` je bila edina, ki je hranila sklic na prvi element v seznamu. Takoj ko smo znotraj stavka `if` v funkciji `seznam.removeBeginning` spremenili njeno vrednost, smo izgubili edino obstoječo kopijo tega sklica. Zato se vozlišče avtomatično izbriše iz pomnilnika in dobimo naslednje končno stanje brez elementa z vrednostjo 13:



Zdaj ko imamo na voljo operacijo, ki vstavi, in operacijo, ki odvzame element z začetka povezanega seznama, lahko tak seznam uporabimo kot sklad. Zaradi večje jasnosti ustrezno preimenujemo obe operaciji v `push` oziroma `pop`:

```
seznam.push = seznam.insertBeginning;
seznam.pop = seznam.removeBeginning;
```

V mnogih jezikih predstavlja ime funkcije (brez oklepajev) sklic na kodo funkcije v pomnilniku. Zaradi tega je po izvršenih gornjih dveh vrsticah kode klic `seznam.push()` popolnoma enakovreden klicu `seznam.insertBeginning()`. Isto velja za drugi par postopkov `pop` in `removeBeginning`. Tako napisan sklad se obnaša popolnoma enako, kakor se je obnašal sklad, ki smo ga napisali z uporabo tabele:

```
seznam.push(2);
seznam.push(4);
seznam.push(8);
console.log(seznam.pop());  //izpiše 8
console.log(seznam.pop());  //izpiše 4
console.log(seznam.pop());  //izpiše 2
```

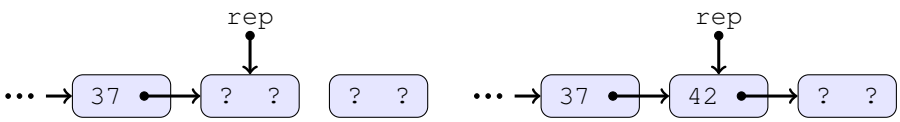
6.3.2 Dodajanje vozlišča na konec seznama

Napišimo še postopek, ki bo dodal element na konec povezanega seznama:

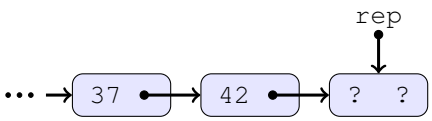
```
seznam.insertEnd = function(vredn) {
  var cuvaj = Object();
  cuvaj.podatek = '?';
  cuvaj.naslednji = '?';
  seznam.rep.podatek = vredn;
  seznam.rep.naslednji = cuvaj;
  seznam.rep = cuvaj;
};
```

V prvih treh vrsticah enostavno ustvarimo novega čuvaja. Potem v četrti vrstici v vozlišče, ki je bilo doslej čuvaj (in na katero se še vedno sklicuje rep), vpišemo vrednost, podano s parametrom `vredn` (`seznam.rep.podatek = vredn`). V predzadnji vrstici usmerimo lastnost `naslednji` tega vozlišča proti na novo ustvarjenemu čuvaju (`seznam.rep.naslednji = cuvaj`).

Stanje po tem, ko se izvedejo prve tri vrstice gornje kode, je prikazano na levi strani spodnje slike. Na desni strani spodnje slike vidimo stanje, ko se izvedeta četrta in peta vrstica, pri čemer smo na konec seznama dodali vrednost 42:



V zadnji vrstici postopka `insertEnd` usmerimo proti na novo ustvarjenemu čuvaju še rep povezanega seznama (`seznam.rep = cuvaj`), s čimer dobimo stanje na naslednji sliki:



V tem trenutku ima naš povezani seznam postopek, s katerim lahko odstranimo element z njegovega začetka, in postopek, s katerim lahko dodamo element na njegov konec. Zato lahko tak seznam uporabimo kot vrsto:

```
seznam.dequeue = seznam.removeBeginning;
seznam.enqueue = seznam.insertEnd;
```

Ena od prednosti take vrste pred izvedbo s krožno tabelo je ta, da velikost vrste s povezanim seznamom teoretično ni omejena. Vanjo lahko zapišemo toliko elementov, kolikor je na voljo pomnilnika. Za konec še preizkusimo, kako deluje naš povezani seznam v vlogi vrste:

```
seznam.enqueue(5);
seznam.enqueue(7);
seznam.enqueue(11);
console.log(seznam.dequeue());  //izpiše 5
console.log(seznam.dequeue());  //izpiše 7
console.log(seznam.dequeue());  //izpiše 11
```

6.3.3 Dodajanje in brisanje poljubnega vozlišča

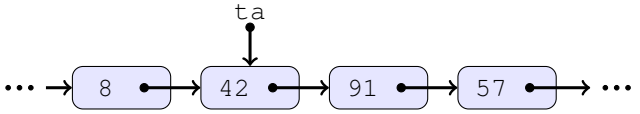
Včasih želimo iz seznama bodisi izbrisati poljuben element bodisi vanj vriniti nov element na poljubno mesto. Če so elementi shranjeni v tabeli, potem moramo ob tem premakniti še vse elemente, ki se nahajajo za mestom, kjer se je zgodila sprememba. Ena od prednosti povezanega seznama je ta, da lahko vanj vstavimo ali iz njega odstranimo vrednost s poljubnega mesta brez potrebe po premikanju vseh ostalih elementov po pomnilniku.

Preden lahko določen element izbrišemo – ali pred njega vrinemo drug element – moramo ta element poiskati. Element poiščemo tako, da začnemo pri prvem elementu (t.j. pri glavi) in se postopoma pomikamo po seznamu z elementa na element, dokler ne pridemo bodisi do iskanega elementa bodisi do čuvaja. Ko se pomikamo po povezanem seznamu, moramo v posebni spremenljivki hraniti sklic na trenutno vozlišče. Predpostavimo, da spremenljivka `ta` hrani sklic na trenutno vozlišče. Potem se pomaknemo na ***naslednje*** vozlišče s takšnim stavkom:

```
ta = ta.naslednji;
```

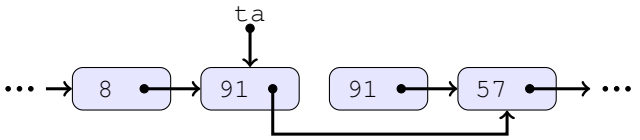
Če želimo odstraniti vozlišče, na katero kaže spremenljivka `ta`, naletimo na težavo, ker prek te spremenljivke ne moremo priti do ***predhodnega*** vozlišča. To vozlišče namreč potrebujemo, da popravimo vrednost njegove lastnosti `naslednji`, ki mora po brisanju elementa kazati na element, ki je bil doslej naslednji element pravkar izbrisanega vozlišča.

Za brisanje vozlišča, na katero kaže spremenljivka `ta`, zato uporabimo preprost trik[1]. Naslednja slika kaže stanje, kjer želimo iz povezanega seznama odstraniti vozlišče, v katerem je vpisana vrednost 42:

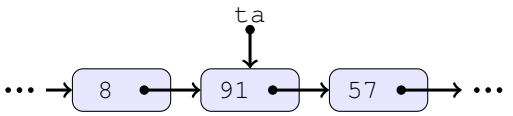


[1]Namesto tega bi lahko pred vsakim pomikom na naslednje vozlišče preprosto shranili povezavo na trenutno vozlišče v pomožno spremenljivko. Uporabili pa bi lahko tudi ***dvojno povezan seznam*** (angl. doubly linked list), v katerem vsako vozlišče vsebuje sklic tudi na svojega predhodnika.

Po brisanju vozlišča želimo imeti stanje, kjer vozlišču, ki se nahaja ***pred*** vozliščem, na katero kaže `ta`, sledi vozlišče, ki se nahaja ***za*** vozliščem, na katero kaže `ta`. Z drugimi besedami, želimo, da osmici v seznamu na gornji sliki sledi vrednost 91. To lahko dosežemo tako, da v vozlišče, na katero kaže `ta`, kopiramo njegovega naslednika. Zdaj imamo dve vozlišči, ki hranita vrednost 91, in obe se sklicujeta na vozlišče, ki hrani vrednost 57:



Takoj ko smo preusmerili lastnost `naslednji` vozlišča, na katero kaže `ta`, smo izgubili edini obstoječi sklic na staro vozlišče, ki hrani vrednost 91. Ker ne obstaja več nobena kopija sklica na to vozlišče, se vozlišče avtomatično odstrani iz pomnilnika. Dobimo končno stanje, ki smo ga želeli:



Dodajmo našemu objektu `seznam` postopek `delete`, ki iz povezanega seznama odstrani element z vrednostjo, ki mu jo podamo kot parameter. Če je takšnih vrednosti v seznamu več, naj odstrani le prvo, če pa takšne vrednosti ni v seznamu, naj postopek ne naredi ničesar.

Celoten algoritem lahko strnemo v naslednji zapis, ki povzema vse opisane korake, potrebne za brisanje elementa iz seznama:

```
seznam. delete = podprogram:
{
  Vhod:  vredn;
  Lokalno:  ta;
  Izhod:  —;
  Usmeri ta na prvo vozlišče seznama:
  Ponavljaj, dokler ta ne kaže na zadnje vozlišče seznama:
  {
    Če je podatek, shranjen v vozlišču ta, enak vredn:
    {
      Če vozlišču ta sledi zadnje vozlišče (čuvaj):
      {
        Usmeri rep seznama na vozlišče ta;
      }
      Kopiraj naslednje vozlišče v vozlišče ta;
      Izhod;
    }
    Premakni ta na naslednje vozlišče;
  }
};
```

Posebej velja opozoriti na drugi pogojni stavek, ki preverja, ali je vozlišče, ki ga brišemo, zadnje pred čuvajem. Ker naš algoritem v resnici odstrani naslednje vozlišče, bo v tem primeru odstranil čuvaja, vlogo novega čuvaja pa bo prevzelo vozlišče, na katero se

sklicuje ta. Ker ta manever povzroči, da je čuvaj zdaj drugo vozlišče, kot je bilo doslej, moramo poskrbeti, da se bo tudi rep skliceval na tega novega čuvaja.

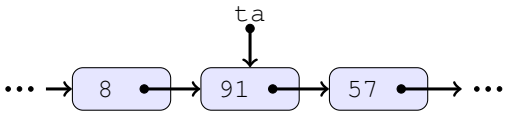
V jeziku JavaScript zapišemo postopek `delete` takole:

```
seznam.delete = function(vredn) {
  var ta;
  ta = seznam.glava;
  while (ta != seznam.rep) {
    if (ta.podatek == vredn) {
      if (ta.naslednji == seznam.rep) {
        seznam.rep = ta;
      }
      ta.podatek = ta.naslednji.podatek;
      ta.naslednji = ta.naslednji.naslednji;
      return;
    }
    ta = ta.naslednji;
  }
};
```

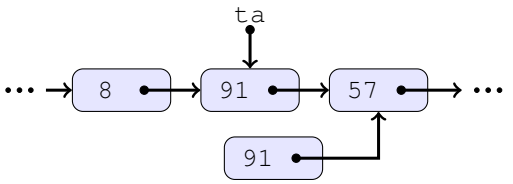
Naloga 6.2 Z uporabo povezanega seznama zgradite slovar, ki deluje tako kot slovar, ki smo ga napisali v razdelku 6.1. Poleg postopkov `insert` in `lookup` naj slovar vsebuje še postopek `delete`. Ta postopek naj iz slovarja odstrani vnos, ki mu pripada ključ, ki ga podamo kot parameter. Slovar preizkusite z naslednjim programom:

```
slovar.insert('a', 97);
slovar.insert('b', 98);
slovar.insert('c', 99);
console.log(slovar.lookup('a')); //izpiše 97
console.log(slovar.lookup('b')); //izpiše 98
console.log(slovar.lookup('c')); //izpiše 99
slovar.delete('b');
console.log(slovar.lookup('a')); //izpiše 97
console.log(slovar.lookup('b')); //izpiše -1
console.log(slovar.lookup('c')); //izpiše 99
```

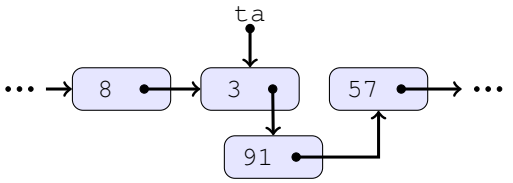
Podoben trik s kopiranjem, ki smo ga uporabili za brisanje vozlišča, lahko uporabimo tudi takrat, kadar želimo pred določeno vozlišče vstaviti novo vozlišče (insertBefore). Vzemimo za primer, da želimo v seznam vstaviti novo vozlišče z vrednostjo tri, in sicer pred vozlišče, na katero se sklicuje `ta`. Na naslednji sliki je to vozlišče, ki hrani vrednost 91:



V prvem koraku algoritma ustvarimo novo vozlišče in vanj kopiramo vozlišče, na katero se sklicuje `ta`. Dobimo stanje, ki ga prikazuje naslednja slika:



V drugem koraku v vozlišče `ta` vpišemo novo vrednost in poskrbimo, da njegovo naslednje vozlišče postane vozlišče, ki smo ga ustvarili v prvem koraku. Ko to opravimo, dobimo želeno stanje, ki je prikazano na naslednji sliki. Vrednost tri smo vstavili pred vrednost 91:



Naloga 6.3 Za vajo dodajte povezanemu seznamu funkcijo `insertBefore`, ki vstavi podani parameter pred prvi element v seznamu, katerega vrednost je manjša ali enaka podanemu parametru. Če takšnega elementa ni, naj podani parameter vstavi na konec seznama. Izhajajte iz zgoraj opisanega algoritma. Napisano funkcijo uporabite za urejanje elementov tabele po velikosti po postopku ***navadnega vstavljanja*** (angl. insertion sort). Postopek navadnega vstavljanja jemlje elemente enega za drugim iz vhodne tabele in za vsakega posebej poišče pravo mesto v urejenem seznamu, kamor ga tudi vstavi.

Pomoč: Algoritem si lahko poenostavite, če pred začetkom iskanja primernega mesta v seznamu vpišete v čuvaja vrednost, ki jo želite vstaviti v seznam. Paziti morate, da v primeru, ko nov element vstavite tik pred čuvaja, ustrezno popravite tudi vrednost repa seznama.

Dodatek: postopku `insertBefore` dodajte še en parameter, katerega vrednost je lahko ena ali -1. Če je njegova vrednost ena, naj postopek deluje tako kot prej. Če je njegova vrednost -1, potem naj postopek vstavi podano vrednost pred prvi element v seznamu, katerega vrednost je ***večja*** ali enaka podani vrednosti.

6.4 Naloge

Za vajo rešite še naslednje naloge:

Naloga 6.4 Krožnemu medpomnilniku na strani 100 dodajte postopek `size`, ki vrne trenutno število elementov, ki so shranjeni v medpomnilniku. Postopek preizkusite z naslednjim programom:

```
for (i = 0; i < 4; i = i + 1) {
```

```
  vrsta.enqueue(i);
}
console.log(vrsta.size()); //izpiše 4
for (i = 0; i < 3; i = i + 1) {
  vrsta.dequeue();
}
console.log(vrsta.size()); //izpiše 1
vrsta.enqueue(25);
vrsta.enqueue(61);
console.log(vrsta.size()); //izpiše 3
```

Naloga 6.5 Povezanemu seznamu, ki smo ga ustvarili v razdelku 6.3, dodajte postopek `size`, ki vrne število elementov, ki so trenutno v seznamu. Postopek preizkusite z naslednjim programom:

```
console.log(seznam.size()); //izpiše 0
seznam.insertBeginning(88);
seznam.insertBeginning(12);
seznam.insertBeginning(7);
console.log(seznam.size()); //izpiše 3
seznam.delete(12);
console.log(seznam.size()); //izpiše 2
seznam.delete(88);
console.log(seznam.size()); //izpiše 1
```

Naloga 6.6 Povezanemu seznamu, ki smo ga ustvarili v razdelku 6.3, dodajte postopek `izpis`, ki izpiše vrednosti vseh elementov, ki so trenutno v seznamu. Postopek preizkusite z naslednjim programom:

```
seznam.insertEnd(10);
seznam.insertEnd(35);
seznam.insertEnd(18);
seznam.insertEnd(1);
seznam.insertEnd(4);
seznam.izpis();  //izpiše 10, 35, 18, 1, 4
```

Naloga 6.7 Povezanemu seznamu, ki smo ga ustvarili v razdelku 6.3, dodajte postopek `povprecje`, ki izračuna in vrne povprečno vrednost vseh elementov, ki so trenutno v seznamu (ob predpostavki, da so v seznamu vpisane izključno številske vrednosti). Tako dopolnjen seznam uporabite za izdelavo programa, ki izračuna enostavno drseče povprečje zaporedja, zapisanega v enorazsežnostni tabeli.

Enostavno drseče povprečje (angl. simple moving average) zaporedja vrednosti dobimo tako, da vsako vrednost nadomestimo s povprečno vrednostjo zadnjih n vrednosti zaporedja. Za zaporedje, zapisano v tabeli $\boldsymbol{p}$, izračunamo i-ti element enostavnega drsečega povprečja po naslednji formuli:

$$\bar{p}_i = \frac{1}{n} \sum_{j=1}^{n} p_{i+n-j}. \tag{6.1}$$

Glede na to, da za izračun vsakega elementa enostavnega drsečega povprečja potrebujemo n elementov, mora imeti tabela $\boldsymbol{p}$ vsaj n elementov. Končna tabela z enostavnim drsečim povprečjem $\bar{\boldsymbol{p}}$ je za $n-1$ element krajša od tabele $\boldsymbol{p}$. Na primer, enostavno drseče povprečje tabele $\boldsymbol{p} = [2, 4, 6, 7, 8, 9]$ pri $n = 3$ je $\bar{\boldsymbol{p}} = [4,\ 5{,}67,\ 7,\ 8]$.

Naloga 6.8 Adam Bohorič, slovenski protestant, slovničar in učitelj, je sredi 16. stoletja določil pravopisna pravila za slovensko pisavo, ki se po njem imenuje bohoričica. Ena od značilnosti te pisave je, da se v njej glasovi č, š in ž zapisujejo kot kombinacije dveh znakov. Na primer, velik Č se zapisuje kot ZH, velika Š in Ž pa kot SH.

Z uporabo vrste napišite program, ki v besedilu, podanem v obliki tabele velikih črk, zaključene z ničlo, nadomesti vsako črko Č s kombinacijo črk Z in H, vsako črko Š ali Ž pa s kombinacijo črk S in H. Dobljena tabela bo v splošnem daljša od izvorne tabele, saj bo program vsako od črk Č, Š in Ž nadomestil z dvema črkama. Na primer, tabela

```
['K', 'R', 'I', 'Ž', 'I', 'Š', 'Č', 'E', 0].
```

bo postala

```
['K', 'R', 'I', 'S', 'H', 'I', 'S', 'H', 'Z', 'H', 'E', 0].
```

PRAZNA STRAN

7. POGLAVJE

REKURZIJA

O rekurziji govorimo, kadar je kakšna stvar določena sama s seboj ali s stvarjo istega tipa. Naslednja slika pločevinke Drostejevega kakava prikazuje primer vizualne rekurzije:

Ženska na etiketi pločevinke drži v roki pladenj s pločevinko, na kateri je pomanjšana slika iste ženske, ki v roki spet drži pladenj s pločevinko s pomanjšano sliko ženske, in

tako dalje. Pločevinko je leta 1904 oblikoval Jan Misset, po imenu kakava pa se takšni vizualni rekurziji pravi tudi ***Drostejev učinek*** (angl. Droste effect).

Zanimiv primer Drostejevega učinka dobimo, če s kamero snemamo ekran, na katerem posneto sliko tudi predvajamo. Na ekranu se tako pojavi slika ekrana. Ker tudi to snemamo in predvajamo, se zdaj na ekranu pojavi slika ekrana, na katerem je prikazana slika ekrana. In tako naprej, teoretično do neskončnosti. V praksi je takšna rekurzija omejena z ločljivostjo ekrana.

V računalništvu je rekurzija proces izvajanja določenega algoritma, v katerem eden od korakov tega algoritma vključuje izvajanje algoritma samega. Tako kot se to zgodi z rekurzivno sliko na ekranu, moramo tudi pri rekurzivnih algoritmih paziti, da se ne bi ponavljali v neskončnost. Poskrbeti moramo za pogoje, pod katerimi se bo rekurziven proces prej ali slej končal. V nadaljevanju bomo videli, kako lahko z rekurzivnimi postopki rešimo določene probleme.

Ko smo govorili o načrtovanju programov z vrha navzdol, smo spoznali, da je priporočljivo kompleksnejše probleme najprej razčleniti na več enostavnejših problemov. Rešitve teh enostavnejših problemov nato sestavimo v končno rešitev začetnega problema. Pogosto se izkaže, da je možno zastavljeni problem poenostaviti v problem, ki je istega tipa kakor prvotni problem. V tem primeru uporabimo ***rekurzijo*** (angl. recursion) in govorimo o postopku načrtovanja ***deli in vladaj*** (angl. divide and conquer), ki je ključ do načrtovanja mnogih pomembnih algoritmov.

Rekurziven postopek je določen z naslednjima dvema lastnostma:

1. Enostaven osnovni primer (angl. base case), ki za rešitev ne potrebuje rekurzije in ga lahko rešimo neposredno. Osnovni primer deluje tudi kot ustavitveni pogoj, brez katerega bi se rekurzija ponavljala v nedogled.
2. Eno ali več rekurzivnih pravil, ki vse ostale primere poenostavijo proti osnovnemu primeru.

Poglejmo si primer računanja faktoriele naravnega števila. Problem lahko na rekurziven način opišemo takole:

$$0! = 1 \text{ (osnovni primer)},$$
$$n! = n\,(n-1)!$$

V drugi vrstici gre za očitno poenostavitev osnovnega problema (t.j. računanja faktoriele števila n), saj le-ta vključuje računanje faktoriele manjšega števila (t.j. števila $n-1$).

S programerskega vidika izvedemo rekurzijo tako, da v definiciji določenega podprograma enkrat ali večkrat kličemo ta isti podprogram. Pri tem moramo paziti, da podprogram kličemo s parametrom, ki pomeni poenostavitev problema proti osnovnemu primeru. Takšnemu podprogramu, ki vsaj na enem mestu vsebuje klic samega sebe, pravimo rekurziven podprogram oziroma ***rekurzivna funkcija***.

Rekurzivno funkcijo, ki izračuna faktorielo naravnega števila n, zapišemo takole:

```
function faktoriela(n) {
  if (n == 0) {
    return 1; //osnovni primer
  }
  return n * faktoriela(n - 1);
}
```

Delovanje funkcije preizkusimo na običajen način:

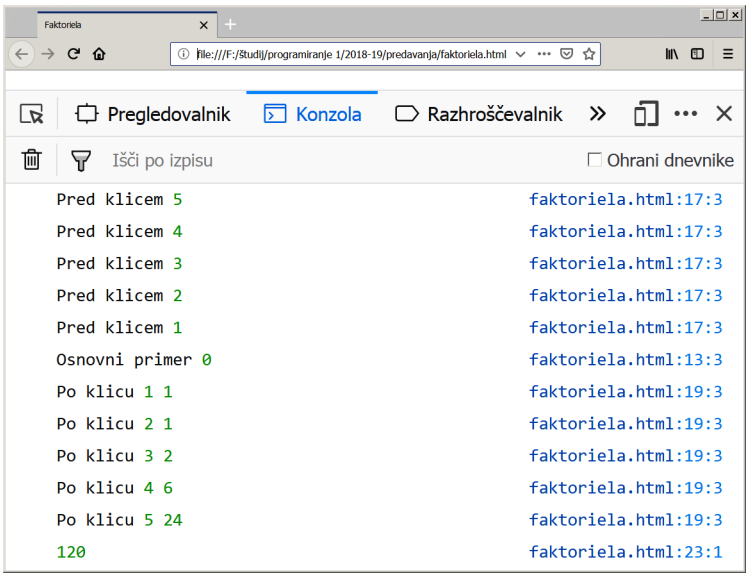
```
console.log(faktoriela(5)); //izpiše 120
```

Stavek `if` v gornji funkciji reši osnovni primer, preostanek kode pa predstavlja rekurzivno definicijo faktoriele. V rekurzivnem klicu funkcije `faktoriela` uporabimo parameter, ki je za ena manjši od vrednosti, ki jo računamo. Na ta način se vsak naslednji klic funkcije izvede z manjšo vrednostjo parametra `n`, dokler na koncu ne pridemo do zadnjega klica, kjer je vrednost `n`-ja enaka nič (osnovni primer). Takrat funkcija `faktoriela` vrne vrednost ena. Vendar to vrednost vrne znotraj tistega klica funkcije, iz katerega je bila klicana. V tem klicu funkcije je bil `n` enak ena, s čimer se mora vrnjena vrednost še pomnožiti in stavek `return` vrne zmnožek 1×1. Ta zmnožek se vrne znotraj klica funkcije, v katerem je bil `n` enak dve. Vrnjena vrednost se zato zdaj pomnoži z dve in zmnožek se vrne v klic funkcije, v katerem je bil `n` enak tri. Postopek se nadaljuje nazaj do prvega klica funkcije, ki vrne končni rezultat.

Za boljšo predstavo o tem, kako delujejo rekurzivni klici funkcije, nekoliko predelajmo gornjo funkcijo `faktoriela` in opazujmo sled parametra `n` ter lokalne spremenljivke `f`. Pomožno lokalno spremenljivko `f` smo dodali zato, da bi lahko opazovali, kakšno vrednost vrne vsakokratni klic funkcije:

```
function faktoriela(n) {
  var f;
  if (n == 0) {
    console.log("Osnovni primer", n);
    return 1;
  }
  console.log("Pred klicem", n);
  f = faktoriela(n - 1);
  console.log("Po klicu", n, f);
  return n * f;
}
console.log(faktoriela(5));
```

Ko program zaženemo v brskalniku, dobimo takšno sled:



Iz slike se lepo vidi, da se funkcija `faktoriela` kliče petkrat zapored, ne da bi se katerikoli od klicev vmes končal. Šele zadnji klic funkcije, kjer je bila vrednost parametra n enaka nič (t.j. klic iz funkcije, v kateri je `n` enak ena), predstavlja osnovni primer, ki konča verigo rekurzivnih klicev. Klici funkcij se zdaj začnejo en za drugim vračati. Vsakokrat, ko se klic funkcije konča, izpišemo še vrednost lokalne spremenljivke `n`, ki mora biti seveda enaka, kakor je bila pred klicem. Iz gornje slike lahko tako vidimo, da se je izvajanje funkcije, ki smo jo prvo klicali, končalo zadnje. Takšno obnašanje nas spominja na sklad. Tudi v resnici se ob vsakem klicu funkcije (in ne le ob rekurzivnem) uporabi sklad: na njem se začasno hranijo vrednosti vseh lokalnih spremenljivk (tako deklariranih spremenljivk kot tudi parametrov). Ko se klic funkcije konča, se prostor, ki je bil rezerviran za njene lokalne spremenljivke, znova sprosti. Običajno nam tega ni treba vedeti, v primeru rekurzije pa se nam lahko kaj hitro zgodi, da se sklad zapolni, če gremo z rekurzivnimi klici pregloboko.

Naloga 7.1 Napišite rekurzivno funkcijo, ki poišče največji skupni delitelj dveh nenegativnih celih števil, pri čemer je vsaj eno večje od nič. Uporabite Evklidov algoritem, ki temelji na opažanju, da se največji skupni delitelj (nsd) dveh števil ne spremeni, če večje od obeh števil zamenjamo z njuno razliko:

$$\mathrm{nsd}(a, b) = \mathrm{nsd}(a - b, b), \text{če } a > b. \tag{7.1}$$

Če je razlika obeh števil še vedno večja od drugega števila, to drugo število znova odštejemo od razlike, sicer števili med seboj zamenjamo. Postopek ponavljamo, dokler števili ne postaneta enaki (takrat je njuna razlika enaka nič). Namesto večkratnega odštevanja lahko uporabimo tudi operator ostanka pri celoštevilskem deljenju, kar nam da naslednji rekurzivni algoritem:

$\mathrm{nsd}(a, 0) = a$ (osnovni primer),
$\mathrm{nsd}(a, b) = \mathrm{nsd}(b, a\%b)$.

7.1 Rekurzija in iteracija

Primere, kot sta računanje faktoriele in največjega skupnega delitelja, je mogoče zapisati z rekurzivno funkcijo, ki sama sebe kliče le enkrat. Vse takšne primere lahko veliko učinkoviteje rešimo z uporabo ***iteracije*** (angl. iteration), kjer posamezne izračune ponavljamo s ponavljalnim stavkom `for` ali `while`. Rekurzivna različica programa je navadno občutno počasnejša zaradi večkratnega klicanja funkcije, ki je časovno sorazmerno potratna operacija. Ko smo na primer primerjali čas izvajanja iterativne različice programa za računanje faktoriele na strani 13 s časom izvajanja rekurzivne različice na strani 114, smo ugotovili, da je rekurzivna različica trikrat počasnejša od iterativne.

Prava moč rekurzije se pokaže šele pri kompleksnejših problemih, kjer iterativni pristop vodi k bistveno bolj zapleteni in nepregledni kodi. Značilnost funkcij, ki rešujejo tovrstne probleme, je ta, da same sebe kličejo več kot enkrat. Nekaj takšnih primerov bomo spoznali v nadaljevanju, na tem mestu pa si poglejmo primer rekurzivnega računanja števil Fibonaccijevega zaporedja (glej nalogo 2.12 na strani 20), ki je zanimiv iz več vidikov. Rekurzivni algoritem za računanje Fibonaccijevih števil lahko zapišemo takole:

$F_0 = 0$, $F_1 = 1$ (osnovni primer),
$F_n = F_{n-1} + F_{n-2}$.

Iz tega lahko takoj zapišemo rekurzivno funkcijo, ki izračuna n-ti člen zaporedja (klic `Fib(n)` vrne vrednost člena F_n):

```
function Fib(n) {
  if (n < 2) {
    return n;
  }
  return Fib(n - 1) + Fib(n - 2);
}
```

Opazimo, da funkcija `Fib` kliče samo sebe dvakrat. In vendar lahko funkcijo z istim učinkom sorazmerno enostavno zapišemo tudi v iterativni obliki:

```
function Fib(n) {
  var i, f0, f1, fn;
  f0 = 0;
  f1 = 1;
  for (i = 0; i < n; i = i + 1) {
    fn = f1 + f0;
    f0 = f1;
    f1 = fn;
  }
  return f0;
}
```

Iterativna različica funkcije je nekoliko bolj zapletena od rekurzivne, ker ne sledi neposredno definiciji Fibonaccijevega zaporedja. Kljub temu je tudi v tem primeru rekurzivna rešitev počasnejša, čeprav krivda za to tokrat ni zgolj v počasnosti klicev funkcij.

7.2 Časovna zahtevnost algoritma

V računalništvu predstavlja ***časovna zahtevnost*** (angl. time complexity) količino časa, ki ga porabi algoritem za svoje izvajanje. Časovno zahtevnost običajno ocenimo tako, da preštejemo število osnovnih operacij, ki jih mora algoritem izvesti. Ob tem predpostavljamo, da potrebuje vsaka od osnovnih operacij za svoje izvajanje konstantno količino časa. Količina časa, ki ga porabi algoritem za izvajanje, in število osnovnih operacij, ki jih mora pri tem izvesti, se potemtakem razlikujeta zgolj za nek konstanten faktor.

Ker je čas, ki ga algoritem porabi za izvajanje, odvisen od podatkov, navadno ocenjujemo čas, ki ga porabi v najslabšem možnem primeru (angl. worst-case time complexity). Navadno je zelo težko natančno določiti čas izvajanja algoritma. Zlasti pri majhnih vhodnih podatkih so časi izvajanja zaradi vplivov ostalih delov algoritma (na primer različnih inicializacijskih postopkov) težko določljivi. Poleg tega so časi pri majhnih vhodnih podatkih večinoma zelo kratki, zaradi česar nas redko sploh zanimajo. Zato se pri določanju časovne zahtevnosti običajno osredotočimo na oceno zahtevnosti v odvisnosti od naraščanja vhodnih podatkov, čemur pravimo ***asimptotična ocena zahtevnosti***. Časovno zahtevnost podajamo kot funkcijo količine vhodnih podatkov n (na primer bitov ali elementov), pri čemer nas zanima le člen z največjim prispevkom.

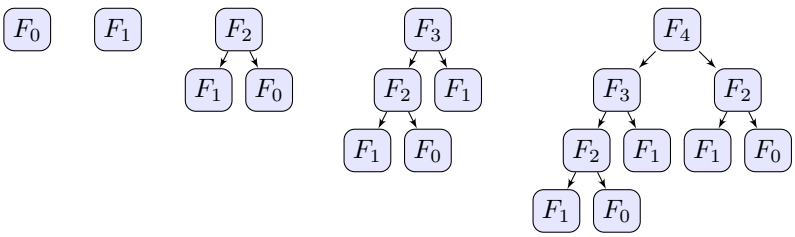
Vzemimo za primer, da izračunamo časovno zahtevnost nekega[1] algoritma $T(n) = 1/2\,n^2 + 1/2\,n$. Pri oceni opustimo linearni člen, ker pri velikih $n$jih h končni vrednosti ne prispeva veliko. Izkaže se tudi, da imajo konstantni faktorji veliko manjši vpliv na časovno zahtevnost, kot jo ima odvisnost od velikosti vhodnih podatkov n. Zato pri navajanju časovne zahtevnosti opustimo tudi konstantni faktor in ostane nam ocena n^2. Kadar navajamo časovno zahtevnost algoritma, uporabljamo tako imenovani ***zapis z velikim Ojem*** (angl. big O notation). Ta zapis se v matematiki uporablja za opisovanje obnašanja funkcije v limitnem primeru, ko gre njen argument proti zelo veliki ali neskončni vrednosti. Črka O se uporablja zato, ker se naraščanju funkcijske vrednosti v angleščini pravi tudi order (slov. red) funkcije. Za naš primer algoritma s časovno odvisnostjo od velikosti vhodnih podatkov $T(n) = 1/2\,n^2 + 1/2\,n$ bi tako navedli, da je njegova časovna zahtevnost $T(n) = \mathcal{O}(n^2)$. Takšni zahtevnosti pravimo tudi ***kvadratna časovna zahtevnost***. Poznamo še vrsto drugačnih časovnih zahtevnosti kot na primer konstantna ($\mathcal{O}(1)$), linearna ($\mathcal{O}(n)$) ali logaritemska ($\mathcal{O}(\log n)$).

Hitro lahko ocenimo časovno zahtevnost iterativnega algoritma za izračun n-tega člena Fibonaccijevega zaporedja iz prejšnjega razdelka. Glavni del algoritma predstavlja zanka `for`, ki se ponovi n-krat. Iterativni algoritem za izračun Fibonaccijevega zaporedja ima torej časovno zahtevnost $T(n) = \mathcal{O}(n)$ oziroma ***linearno časovno zahtevnost***. V praksi to pomeni, da za izračun člena s k-krat večjim indeksom potrebujemo k-krat več časa.

Zgodba je povsem drugačna z rekurzivno različico Fibonaccijevega algoritma. Za oceno njegove časovne zahtevnosti bo zadostovalo, da za osnovno operacijo štejemo klic funkcije `Fib`. Oceniti moramo, kako z indeksom člena, ki ga računamo (n), narašča število klicev funkcije `Fib`.

Označimo s C_n število klicev funkcije `Fib`, ki so potrebni za izračun n-tega člena Fibonaccijevega zaporedja F_n. Ker predstavlja $n < 2$ osnovni primer, za $n = 0$ in $n = 1$ ne pride do rekurzivnega klica funkcije `Fib`. Tako imamo $C_0 = 1$ in $C_1 = 1$. Za $n = 2$ moramo poleg začetnega klica funkcije (t.j. `Fib(2)`) klicati funkcijo še za oba osnovna primera (t.j. `Fib(1)` in `Fib(0)`). Tako dobimo, da je $C_2 = 3$. Za $n = 3$ moramo poleg začetnega klica funkcije klicati funkcijo še za $n = 2$ in $n = 1$, potem pa moramo seveda za $n = 2$ klicati funkcijo še dvakrat: za $n = 1$ in $n = 0$. Za izračun člena F_3 potrebujemo tako že $C_3 = 5$ klicev.

Pri izračunu števila klicev za večje parametre n si pomagajmo z naslednjo sliko, ki prikazuje drevesa vseh potrebnih klicev funkcije `Fib` za izračun členov Fibonaccijevega zaporedja od F_0 do F_4:



Iz slike lahko razberemo, da potrebujemo za vsak naslednji člen (od vključno člena F_2 naprej) toliko klicev kot za izračun prejšnjih dveh členov skupaj plus dodaten začetni klic funkcije. Vse ugotovitve strnemo v naslednji dve enačbi:

[1] Na strani 123 bomo videli, da ima takšno časovno zahtevnost algoritem za urejanje z navadnim vstavljanjem.

$$C_0 = C_1 = 1,$$
$$C_n = C_{n-1} + C_{n-2} + 1,$$

iz katerih izračunamo naslednje zaporedje:

$$1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, \cdots$$

To zaporedje govori o tem, koliko klicev funkcije `Fib` potrebujemo, da izračunamo posamezni člen Fibonaccijevega zaporedja. Vidimo, da potrebujemo za izračun člena F_{10} že 177 klicev funkcije. Da lahko izračunamo časovno zahtevnost algoritma, moramo ugotoviti, kakšno je razmerje dveh zaporednih vrednosti gornjega zaporedja za dovolj velik n. Pokazati je mogoče, da je to razmerje enako razmerju zlatega reza[2]:

$$\lim_{n\to\infty} \frac{C_n}{C_{n-1}} = \frac{1+\sqrt{5}}{2} = 1{,}6180339887... \tag{7.2}$$

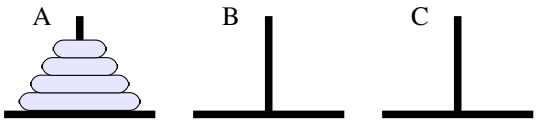
Za nas to pomeni, da moramo za izračun vsakega člena opraviti približno $1{,}62$-krat več klicev funkcije `Fib` kot za izračun zadnjega člena pred njim. Rekurzivna različica algoritma za izračun členov Fibonaccijevega zaporedja ima potemtakem ***eksponentno časovno zahtevnost*** oziroma, natančneje, zahtevnost $T(n) = \mathcal{O}(1{,}62^n)$. Algoritmi z eksponentno zahtevnostjo so izredno neučinkoviti. Izmerili smo, da računanje člena F_{32} z iterativnim postopkom vzame približno $0{,}1$ μs, računanje člena F_{64} pa približno $0{,}2$ μs. Po drugi strani potrebujemo za računanje člena F_{32} z rekurzivno funkcijo 30 ms, za računanje člena F_{64} pa že celih 42 ur! Nauk te zgodbe je, da krajši in na videz preprostejši program ni nujno tudi učinkovitejši.

Za konec ocenimo še časovne zahtevnosti operacij nad povezanim seznamom, ki smo jih srečali proti koncu 6. poglavja. Hitro ugotovimo, da so operacije insertBeginning, removeBeginning in insertEnd popolnoma neodvisne od števila vozlišč v seznamu. Za algoritme, katerih čas izvajanja je neodvisen od velikosti podatkov, pravimo, da imajo ***konstantno časovno zahtevnost*** oziroma imajo časovno zahtevnost $T(n) = \mathcal{O}(1)$. Operaciji delete in insertBefore zahtevata, da najprej poiščemo element, ki ga želimo brisati oziroma pred katerega želimo vriniti nov element. Element v povezanem sezamu lahko poiščemo le tako, da po vrsti pregledamo vse elemente, začenši s prvim elementom v seznamu. Čas, ki ga potrebujemo za takšno iskanje, je linearno odvisen od števila elementov. Zato imata operaciji delete in insertBefore ***linearno časovno zahtevnost*** oziroma časovno zahtevnost $T(n) = \mathcal{O}(n)$.

7.3 Hanojski stolp

Hanojski stolp (angl. Tower of Hanoi) je matematična uganka. Sestavljena je iz treh pokončnih palic in poljubnega števila diskov različnih velikosti, ki jih lahko nataknemo na katerokoli palico. Na začetku so vsi diski zloženi na prvi palici in urejeni po velikosti, tako da je največji na dnu, najmanjši pa na vrhu. Naslednja slika prikazuje začetno stanje uganke s štirimi diski:

[2]Dve količini sta v razmerju zlatega reza (angl. golden ratio), kadar je njuno razmerje enako razmerju njune vsote proti večji od obeh količin. Zapisano algebraično, količini a in b, za kateri velja $a > b > 0$, sta v razmerju zlatega reza, če velja enakost $(a+b)/a = a/b$.



Palice smo označili s črkami od A do C. Cilj igre je prestaviti vse diske s palice A na palico B, pri čemer moramo upoštevati tri pravila:

1. Naenkrat lahko premaknemo le en disk.
2. V vsaki potezi vzamemo vrhnji disk z enega od stolpov in ga postavimo na vrh drugega stolpa ali nataknemo na prazno palico.
3. Nikoli ne smemo postaviti večjega diska na vrh manjšega diska.

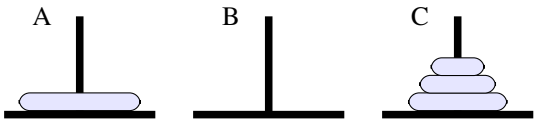
Za rešitev lahko uporabimo tako iterativen kot tudi rekurziven postopek, ki bosta imela oba ***eksponentno časovno zahtevnost*** (t.j. $T(n) = \mathcal{O}(2^n)$, pri čemer je n število diskov). Najmanjše število potez, s katerimi lahko rešimo to uganko, je namreč v vsakem primeru $2^n - 1$. Zaradi narave problema pa je rekurzivna rešitev mnogo enostavnejša od iterativne.

Osnovni primer rekurzivnega postopka predstavlja uganka z enim samim diskom. Tega enostavno premaknemo s palice A na palico B in naloga je rešena.

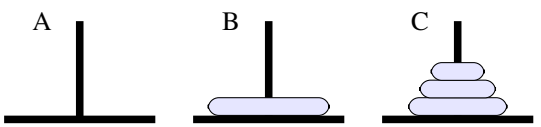
Rekurzivni del rešitve (za n diskov, pri čemer je $n > 1$) sestavimo iz treh potez:

1. prestavimo $(n - 1)$ diskov s palice A na palico C;
2. prestavimo preostali (največji) disk s palice A na palico B;
3. prestavimo $(n - 1)$ diskov s palice C na palico B.

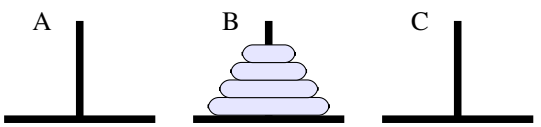
Pomembno je, da smo pri rekurzivnem delu rešitve problem z n diski poenostavili na dva problema z $(n-1)$ diski. Če izhajamo iz gornje slike s štirimi diski, dobimo po prvem koraku rekurzivnega dela rešitve naslednje stanje:



V drugem koraku premaknemo največji disk s palice A na palico B:



V zadnjem koraku prestavimo tri diske s palice C na palico B in uganka je rešena:



Napišimo zdaj rekurzivno funkcijo, ki bo v konzolo izpisala navodila za rešitev uganke. V funkciji v resnici ne bomo premikali diskov (t.j. podatkov po pomnilniku). Vsakokrat, ko bo treba premakniti en sam disk, bomo zgolj izpisali navodilo za premik tega diska. Navodilo bomo izpisali v obliki para črk, od katerih bo prva predstavljala palico, s katere

je treba disk sneti, in druga palico, na katero je treba disk natakniti. Takole je videti dokončana funkcija:

```
function HanojskiStolp(n, start, cilj, pomoc) {
  //Premakne n diskov s palice start na palico cilj,
  //pri čemer uporabi pomožno palico pomoc.
  if (n == 1) {
    //Osnovni primer:  en disk s palice start na palico cilj:
    console.log(start, "-->", cilj);
  }
  else {
    //(n-1) diskov s palice start na palico pomoc:
    HanojskiStolp(n - 1, start, pomoc, cilj);
    //En disk s palice start na palico cilj:
    console.log(start, "-->", cilj);
    //(n-1) diskov s palice pomoc na palico cilj:
    HanojskiStolp(n - 1, pomoc, cilj, start);
  }
}
```

Funkcija sprejme štiri parametre: število diskov `n` ter oznake treh palic `start`, `cilj` in `pomoc`. Ko se bo funkcija izvedla do konca, bomo dobili v konzoli izpisano navodilo, kako prestaviti `n` diskov s palice `start` na palico `cilj`, pri čemer uporabimo dodatno palico `pomoc`.

Če želimo rešitev uganke s štirimi diski, pokličemo funkcijo takole:

```
HanojskiStolpi(4, 'A', 'B', 'C');
```

Po pričakovanju dobimo rešitev s 15 ($= 2^4 - 1$) potezami:

```
A --> C     B --> C     B --> A
A --> B     A --> C     C --> B
C --> B     A --> B     A --> C
A --> C     C --> B     A --> B
B --> A     C --> A     C --> B
```

Obstaja legenda, ki pravi, da je v znamenitem indijskem templju Kaši Višvanath velika soba s tremi starimi stebri, obkroženimi s 64 zlatimi diski. V njej hindujski duhovniki, zavezani starodavni prerokbi, že od stvaritve sveta naprej premikajo zlate diske v skladu s strogimi pravili boga Brahme. Legenda pravi, da se bo v trenutku, ko bodo duhovniki zaključili zadnjo potezo uganke, svet končal. Tudi če je legenda resnična, se nam ni treba bati konca sveta. Izmerili smo čas izvajanja gornjega programa in ugotovili, da za izpis rešitve uganke z desetimi diski (1023 potez) potrebuje približno 120 milisekund. Za izpis rešitve uganke s 64 diski bi program tako potreboval skoraj 70 milijonov let! Če poleg tega upoštevamo še dejstvo, da duhovniki ne morejo premakniti 1023 zlatih diskov v 120 milisekundah (po nekaterih različicah[3] legende jim je dovoljeno premakniti le en disk na dan), potem je časa res še dovolj.

[3]Ena od različic te legende govori tudi, da se zgodba dogaja v nekem templju v Hanoju, glavnem mestu Vietnama, po katerem se uganka tudi imenuje.

7.4 Dvojiško iskanje

Iskanje podatkov je ena najosnovnejših operacij na vseh področjih programiranja. Tudi mi smo se z njo že velikokrat srečali. Spoznali smo, da lahko v tabeli ali povezanem seznamu najdemo želeno vrednost tako, da po vrsti enega za drugim pregledamo vse elemente. Takšnemu načinu iskanja pravimo tudi ***linearno iskanje*** (angl. linear search). Ugotovili smo že, da takšen pristop izkazuje linearno časovno zahtevnost $T(n) = \mathcal{O}(n)$.

Mnogokrat se pripeti, da iščemo vrednost v tabeli, kjer so podatki tako ali drugače urejeni. Takrat lahko uporabimo ***dvojiško iskanje*** (angl. binary search), ki je veliko učinkovitejše od linearnega iskanja.

Vzemimo za primer, da želimo poiskati vrednost 13 v naslednji tabeli:

$$\boldsymbol{t} = [1, 5, 6, 9, 12, 13, 15, 17, 19, 22, 25, 47, 78, 123, 817, 9001]$$

Iskati začnemo pri elementu na sredini tabele. Vendar ima lahko tabela tudi sodo število elementov (kot na primer gornja tabela $\boldsymbol{t}$) in je sredina tabele med dvema elementoma. V tem primeru se dogovorimo, da izmed teh dveh elementov za srednji element izberemo tistega z manjšim indeksom. Med postopkom iskanja bomo večino časa iskali elemente samo v delu tabele med indeksoma a in b (vključno), pri čemer mora veljati $0 \leq a \leq b < n$, kjer je n število elementov v tabeli. V splošnem nas bo zato zanimal srednji element med indeksoma a in b, ki ga lahko izračunamo kot $\lfloor (a + b)/2 \rfloor$. Operator $\lfloor x \rfloor$ predstavlja zaokroževanje navzdol (t.j. največjo celoštevilsko vrednost, ki je manjša ali enaka x). Indeks srednjega elementa gornje tabele $\boldsymbol{t}$ (katere elementi imajo indekse od nič do 15) je tako $\lfloor 15/2 \rfloor = 7$, njegova vrednost pa je $t_7 = 17$. Ker je ta element večji od iskane vrednosti, vemo, da se lahko iskana vrednost nahaja kvečjemu v delu tabele med elementoma t_0 in t_6. Postopek nadaljujemo tako, da spet poiščemo srednji element tega manjšega intervala. Srednji element je zdaj element $t_3 = 9$, ki je manjši od iskane vrednosti. To pomeni, da se lahko iskana vrednost nahaja kvečjemu na intervalu elementov od t_4 do t_6. Srednji element tega intervala je $t_5 = 13$, kar je tudi element, ki smo ga iskali.

Videli smo, da pri dvojiškem iskanju v vsakem koraku razpolovimo število elementov, ki jih moramo pregledati. To je pomembna ugotovitev, na podlagi katere ocenimo časovno zahtevnost algoritma: če podvojimo število vhodnih podatkov, se čas računanja poveča le za eno enoto. Algoritem dvojiškega iskanja ima zato ***logaritemsko časovno zahtevnost*** $T(n) = \mathcal{O}(\log n)$. Naj omenimo, da spadajo algoritmi, ki izkazujejo logaritemsko časovno zahtevnost, med učinkovitejše algoritme.

Zapišimo zdaj formalen postopek za dvojiško iskanje vrednosti x v podtabeli tabele $\boldsymbol{t}$ od vključno elementa z indeksom a do vključno elementa z indeksom b:

$a > b$ (osnovni primer; končni indeks je manjši od začetnega; elementa ni v tabeli);
$sr \leftarrow \lfloor (a + b)/2 \rfloor$ (izračunamo indeks srednjega elementa);
$x = t_{sr}$ (osnovni primer; indeks iskanega elementa je sr);
$b \leftarrow sr - 1$, če $x < t_{sr}$ (premik zgornje meje; iskali bomo do srednjega elementa);
$a \leftarrow sr + 1$, če $x > t_{sr}$ (premik spodnje meje; iskali bomo za srednjim elementom);
dvojiskoIskanje$(\boldsymbol{t}, a, b, x)$ (rekurzivni klic s krajšim območjem med a in b).

V postopku imamo dva osnovna primera. Prvi osnovni primer nastopi, če postane začetni indeks iskanja a večji od končnega indeksa b. To se primeri takrat, ko iskane vrednosti ni v tabeli in se zaradi višanja spodnje meje oziroma nižanja zgornje meje indeksa nazadnje prekrižata. Če se to zgodi, bomo vrnili vrednost -1, ki ne predstavlja veljavnega indeksa

in jo zato uporabimo kot znak, da iskanega elementa ni v tabeli. Drugi osnovni primer imamo, če je srednji element enak iskani vrednosti. Takrat vrnemo indeks tega elementa.

V rekurzivnem delu najprej popravimo bodisi spodnjo bodisi zgornjo mejo iskanja. Ker gre v obeh primerih za premik proti nasprotni meji, s tem v resnici vsakokrat poenostavimo problem proti enemu od obeh osnovnih primerov tako, da skrajšamo območje iskanja. Na koncu izvedemo rekurzivni klic funkcije.

Funkcija `dvojiskoIskanje`, zapisana v jeziku JavaScript, je videti takole:

```
function dvojiskoIskanje(t, a, b, x) {
  var sr;
  if (a > b) {
    return -1;
  }
  sr = Math.floor((a + b) / 2);
  if (x == t[sr]) {
    return sr;
  }
  if (x < t[sr]) {
    return dvojiskoIskanje(t, a, sr - 1, x);
  }
  return dvojiskoIskanje(t, sr + 1, b, x);
}
```

Preizkusimo še, kako funkcija deluje:

```
console.log(dvojiskoIskanje([3, 6, 9, 14, 23], 0, 4, 9)); //izpiše 2
console.log(dvojiskoIskanje([3, 6, 9, 14, 23], 0, 4, 5)); //izpiše -1
```

V zadnjih dveh primerih smo vrednosti iskali po celi tabeli. Glede na to, da že osnovni klic funkcije zahteva, da podamo začetni in končni indeks, lahko vrednosti iščemo le po določenem delu tabele. Na primer:

```
console.log(dvojiskoIskanje([3, 6, 9, 14, 23], 2, 4, 6)); //izpiše -1
console.log(dvojiskoIskanje([3, 6, 9, 14, 23], 1, 3, 14)); //izpiše 3
```

7.5 Urejanje s kopico

Tako kot iskanje je tudi urejanje (angl. sort) podatkov pomembno na mnogih področjih. V nalogi 6.3 na strani 109 smo na primer spoznali postopek urejanja z navadnim vstavljanjem. Algoritem je sicer preprost, vendar spada med manj učinkovite algoritme za urejanje podatkov. Časovno zahtevnost algoritmov za urejanje ocenjujemo na podlagi števila potrebnih primerjav. Predpostavimo, da pri urejanju z navadnim vstavljanjem uporabimo čuvaja na koncu seznama, v katerega vstavljamo podatke. Za vstavljanje prvega podatka potrebujemo zato eno primerjavo, za vstavljanje drugega podatka dve primerjavi, za vstavljanje tretjega podatka tri primerjave, in tako dalje. Pri tej oceni smo upoštevali najslabši možni primer. Za vstavljanje n podatkov potrebujemo tako v najslabšem primeru vsoto vseh primerjav za vstavljanje od prvega do vključno n-tega podatka:

$$\sum_{i=1}^{n} i = \frac{1}{2}\, n\,(n+1) = \frac{1}{2}\, n^2 + \frac{1}{2}\, n. \tag{7.3}$$

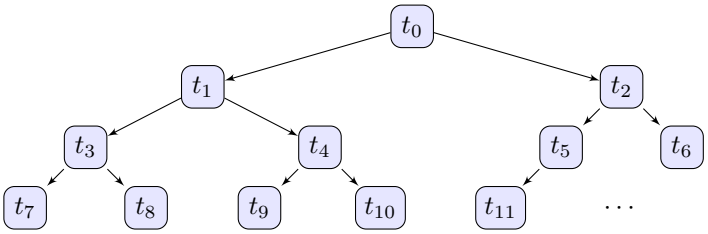
Ker upoštevamo le vodilni člen brez konstantnega faktorja, ugotovimo, da ima algoritem za urejanje z navadnim vstavljanjem ***kvadratno časovno zahtevnost*** $T(n) = \mathcal{O}(n^2)$[4].

Videli bomo, da je ***urejanje s kopico*** (angl. heap sort) veliko učinkovitejši algoritem. Za postopek urejanja s kopico moramo podatke najprej zložiti v ***dvojiško drevo*** (angl. binary tree). Dvojiško drevo je podatkovna struktura, sestavljena iz vozlišč, pri čemer je lahko vsako vozlišče ***starš*** (angl. parent) največ dveh ***potomcev*** (angl. child). Po drugi strani ima vsak potomec natanko enega starša. Edinemu vozlišču, ki nima starša, pravimo ***koren*** (angl. root). Vozliščem, ki nimajo potomca, pravimo ***listi*** (angl. leaf).

Dogovorimo se, da bomo podatke iz tabele

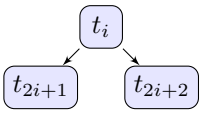
$$\boldsymbol{t} = [t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, \cdots]$$

zložili v dvojiško drevo na naslednji način:



Podatkov ne bomo v resnici zlagali v dvojiško drevo. Iz gornjega drevesa lahko razberemo, da je levi potomec vozlišča z indeksom i vedno vozlišče z indeksom $2i+1$. Podobno je desni potomec vozlišča z indeksom i vedno vozlišče z indeksom $2i + 2$. Ti dve enačbi bomo uporabljali, kadar se bomo v programu želeli pomakniti s starša na enega od njegovih dveh potomcev.

Osnovna operacija pri urejanju s kopico je ***pogrezanje kopice*** (angl. heapify). Ta operacija rekurzivno zamenjuje vozlišče z večjim izmed njegovih dveh potomcev (če je le-ta večji tudi od starša). Rekurzija se konča takrat, ko bodisi nobeden od potomcev ni večji od starša bodisi smo prispeli do lista (vozlišča brez potomcev). Vzemimo na primer naslednjo trojico vozlišč:



Rekurzivni algoritem pogrezanja kopice lahko zapišemo takole:

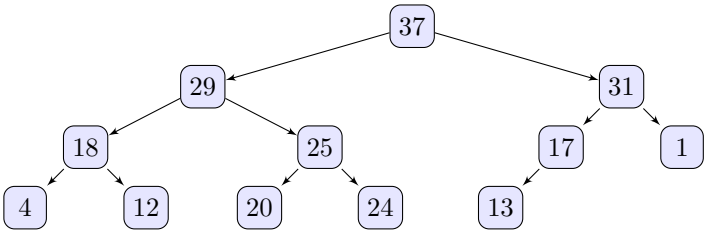
> Če $t_i = \text{maks}(t_i, t_{2i+1}, t_{2i+2})$, končaj s pogrezanjem kopice (osnovni primer),
> sicer zamenjaj t_i z $\text{maks}(t_{2i+1}, t_{2i+2})$
> in pogrezni kopico s korenom v potomcu, ki smo ga pravkar zamenjali s staršem.

Rekurzivni del tega postopka ponovi pogrezanje kopice iz enega od potomcev trenutnega vozlišča, kar nedvomno predstavlja poenostavitev primera. Del drevesa, ki izhaja iz potomca, je namreč vedno manjši od drevesa, ki izhaja iz starša.

S pogrezanjem kopice iz korena dosežemo, da se v koren drevesa prestavi element z največjo vrednostjo. Tega potem zamenjamo z zadnjim listom in ponovimo pogrezanje

[4]Kvadratno časovno zahtevnost imata tudi algoritma urejanja z navadnim izbiranjem in urejanja z mehurčki, ki smo ju spoznali v nalogah 4.5 in 4.6 na straneh 62 in 63.

kopice brez tega zadnjega lista. Da pa to deluje, moramo algoritem začeti z ***maksimalno kopico***. Maksimalna kopica je drevo, v katerem nobeno vozlišče nima neposrednega potomca, ki bi bil večji od njega. Naslednja slika prikazuje primer maksimalne kopice:



Maksimalno kopico ustvarimo tako, da pokličemo rekurzivno funkcijo za pogrezanje kopice za vsa vozlišča, ki imajo potomce. To storimo po vrsti od spodaj navzgor.

Zapišimo zdaj celoten algoritem, ki uredi tabelo $\boldsymbol{t}$ z n elementi po postopku urejanja s kopico:

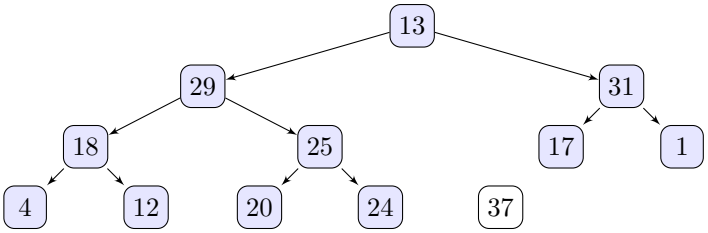
```
//Opomba: maksimiranje kopice:
Ponovi za vsak i od ⌊n/2⌋ − 1 do 0 s korakom −1:
{
  Pogrezni podkopico tabele t
    s korenom v vozlišču z indeksom i;
}
//Opomba: urejanje s kopico:
Ponovi za vsak i od n − 1 do 1 s korakom −1:
{
  Zamenjaj elementa t0 in ti;
  Pogrezni kopico iz prvih i elementov
    tabele t s korenom v vozlišču z indeksom 0;
}
```

V prvem od gornjih dveh ponavljalnih stavkov maksimiramo kopico. Ugotovili smo že, da ima levi potomec vozlišča z indeksom i indeks $2i + 1$, desni potomec pa ima indeks $2i + 2$. To pomeni, da je največji indeks vozlišča, ki še ima lahko potomca (t.j. indeks zadnjega starša), enak $\lfloor n/2 \rfloor - 1$. Če je n liho število, potem ima zadnji starš dva potomca, sicer ima le enega. Z zaokroževanjem navzdol dosežemo, da dobimo v obeh primerih isti indeks za zadnjega starša.

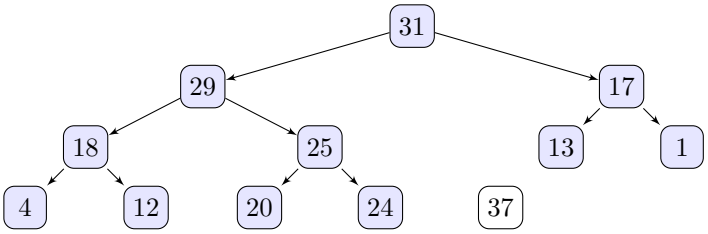
Za uspešno maksimiranje moramo kopice pogrezati od spodaj navzgor, se pravi od zadnjega starša z indeksom $\lfloor n/2 \rfloor - 1$ do korena, ki ima indeks nič. V tem primeru pod nobenim vozliščem, iz katerega začnemo pogrezati kopico, ne bo nobenega vozlišča, ki bi imelo kakšnega potomca večjega od sebe. To je pomembno za delovanje algoritma.

Drugi ponavljalni stavek predstavlja postopek urejanja. Če začnemo z maksimirano kopico, potem je največja vrednost v korenu (t.j. v vozlišču z indeksom nič). To vrednost zamenjamo z zadnjim listom. Potem znova pogreznemo kopico iz korena, vendar brez zadnjega vozlišča. To ponovimo $(n - 1)$-krat in tabela je urejena.

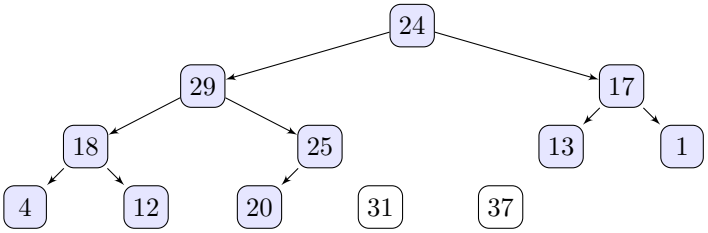
Vzemimo za primer maksimalno kopico s prejšnje slike. Ko zamenjamo koren z zadnjim listom in zmanjšamo kopico, dobimo naslednje stanje:



Vozlišče z vrednostjo 37 zdaj ni več del kopice, v korenu pa je vrednost 13. V naslednjem koraku pogreznemo zmanjšano kopico, pri čemer začnemo pri vrednosti 13, ki je zdaj v korenu. V postopku pogrezanja se bo vrednost 13 najprej zamenjala z vrednostjo 31, potem pa še z vrednostjo 17. Dobimo naslednjo kopico, ki je spet maksimalna:

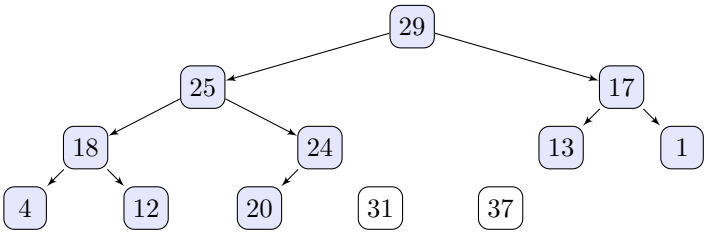


Spet imamo največjo vrednost v korenu. Zamenjamo jo z zadnjim listom ter zmanjšamo kopico. Dobimo naslednje stanje:



Zadnji dve vozlišči zdaj nista več del kopice, sta pa še vedno del tabele, ki jo urejamo. Tako imamo zdaj na predzadnjem in zadnjem mestu tabele že drugo največjo in največjo vrednost.

Pri naslednjem pogrezanju se vrednost 24, ki je v korenu, zamenja najprej z vrednostjo 29 in potem še s 25. Na koncu se vrednost 24 primerja še z 20, ki pa ni večja od nje. Tako se to pogrezanje na tem mestu konča. Dobimo naslednjo sliko:



Vrednost 29, ki je zdaj v korenu, je tretja po velikosti in se bo po menjavi z zadnjim listom (ki ima trenutno vrednost 20) postavila v tabeli na tretje mesto od zadaj. S ponavlja-

njem postopka pogrezanja kopice iz korena in menjave korena z zadnjim listom pridemo na koncu do tabele, v kateri so elementi urejeni po velikosti od najmanjšega proti največjemu.

Sledi program za urejanje s kopico, zapisan v JavaScriptu:

```
function zamenjaj(t, i, j) {
  //V tabeli t med seboj zamenja elementa z indeksoma i in j.
  var zacasno;
  zacasno = t[i];
  t[i] = t[j];
  t[j] = zacasno;
}

function pogrezniKopico(t, n, i) {
  //Parameter n predstavlja število elementov v tabeli t,
  //ki jih upoštevamo pri pogrezanju.
  //Parameter i predstavlja vozlišče, iz katerega pogrezamo kopico
  //(i je indeks starša, ki ga bomo primerjali z obema potomcema).
  var maks, levi, desni;
  maks = i;
  //Indeksa levega in desnega potomca:
  levi = i * 2 + 1;
  desni = i * 2 + 2;
  //Če levi potomec obstaja in je večji od starša:
  if (levi < n && t[levi] > t[maks]) {
    maks = levi;
  }
  //Če desni potomec obstaja in je večji od večjega
  //od obeh elementov iz prejšnje primerjave:
  if (desni < n && t[desni] > t[maks]) {
    maks = desni;
  }
  //Če trenutni starš ni večji od obeh potomcev:
  if (maks != i) {
    zamenjaj(t, maks, i);
    //Nadaljuj pogrezanje pri potomcu, ki smo ga
    //pravkar zamenjali s staršem:
    pogrezniKopico(t, n, maks);
  }
}

function urejanjeSKopico(t, n) {
  var i;
  for (i = Math.floor(n / 2) - 1; i >= 0; i = i - 1) {
    pogrezniKopico(t, n, i);
  }
  for (i = n - 1; i > 0; i = i - 1) {
    zamenjaj(t, 0, i);
    pogrezniKopico(t, i, 0);
  }
}
```

Program preizkusimo z naslednjo kodo:

```
tabela = [9, -11, 76, 43, 365, -363, 0, 1, 17, 6];
n = 10;
urejanjeSKopico(tabela, n);
console.log(tabela);
```

Za konec ocenimo še časovno zahtevnost algoritma urejanja s kopico. Iz gornjega programa lahko hitro razberemo, da se daleč največ časa porabi za klice rekurzivne funkcije `pogrezniKopico`. Najprej opazimo, da vsebuje funkcija `urejanjeSKopico` v obeh ponavljalnih stavkih skupaj približno $3/2\,n$ osnovnih klicev funkcije `pogrezniKopico`. Vsak od teh klicev zaradi rekurzije povzroči še največ d klicev iste funkcije, pri čemer je d največja možna globina[5] vozlišča v ***popolnem dvojiškem drevesu*** (angl. perfect binary tree), za katero velja, da so vsi njegovi listi na isti globini in da imajo vsi starši po dva potomca. Popolno dvojiško drevo ima tako na prvem nivoju eno vozlišče (koren), na drugem dve, na tretjem štiri, na četrtem osem vozlišč in tako dalje. Za popolno dvojiško drevo z globino d tako potrebujemo $n = 2^d - 1$ vozlišč, iz česar sledi, da je njegova globina $d = \log_2(n+1)$. Ocenjeno maksimalno število klicev funkcije `pogrezniKopico` je tako $3/2\,n\,d = 3/2\,n\log_2(n+1)$. Iz tega dobimo končno oceno časovne zahtevnosti $T(n) = \mathcal{O}(n \log n)$, čemur pravimo ***skoraj linearna časovna zahtevnost*** (angl. quasilinear time complexity). V primerjavi z urejanjem z navadnim vstavljanjem je torej urejanje s kopico precej učinkovitejši algoritem.

7.6 Vzvratno sledenje

Vzvratno sledenje (angl. backtracking) je splošen algoritem, s katerim iščemo rešitve določenih računskih problemov, med katerimi jih je večina tako imenovanih ***problemov z zadovoljevanjem omejitev*** (angl. constraint satisfaction problem). Med njimi najdemo različne logične uganke, kot so križanke, problem osmih kraljic (glej nalogo 7.6 na strani 132) ali sudoku (glej nalogo 7.7 na strani 132). V postopku vzvratnega sledenja postopoma gradimo možnega kandidata za rešitev ter ga opustimo, kakor hitro ugotovimo, da kandidata ni mogoče dokončati do končne veljavne rešitve. Pri tem vzvratno sledimo delni poti, ki smo jo opravili pri gradnji neuspešne rešitve, dokler ne pridemo do točke, od koder lahko začnemo graditi novega možnega kandidata za rešitev. Poglejmo si načelo delovanja takšnega algoritma na primeru blodnjaka, ki smo ga rešili že v razdelku 3.6.

Postopek reševanja z algoritmom z vzvratnim sledenjem se začne podobno kot postopek, ki smo ga uporabili v razdelku 3.6: V vsakem trenutku poskusimo, če je možno nadaljevati pot v levo. Če to ni mogoče, poskusimo, če je mogoče pot nadaljevati naravnost. Če tudi to ni mogoče, poskusimo še v desno. Če poti ni mogoče nadaljevati v nobeno od treh smeri, potem rešitev od tega mesta naprej ne obstaja. V tem hipu se je treba vrniti nazaj do mesta, od koder še nismo preizkusili vseh možnih smeri.

Če problem rešimo z rekurzivnimi klici funkcij, si nam ni treba posebej zapomniti, kje smo že bili, da se lahko tja vrnemo. Ko se rekurzivni klici funkcije vračajo v prejšnje klice funkcije, je ta informacija avtomatično na voljo v njenih lokalnih spremenljivkah. Končen učinek algoritma je ta, da najde pot, ki ne obiskuje slepih rokavov blodnjaka, kot smo to videli v rešitvi v razdelku 3.6.

Takole je videti rekurzivni postopek vzvratnega sledenja za iskanje poti skozi blodnjak:

> Če je pred tabo čuvaj, si našel izhod (osnovni primer).
> Če je prosta pot v levo, naredi korak v levo in išči dalje,
> sicer, če je prosta pot naprej, naredi korak naprej in išči dalje,
> sicer, če je prosta pot v desno, naredi korak v desno in išči dalje,
> sicer rešitev od tod naprej ne obstaja.

[5]Globina vozlišča je celoštevilska vrednost, ki je za ena večja od števila njegovih prednikov.

Navodilo *išči dalje*, ki se pojavi trikrat v drugem delu gornjega algoritma, predstavlja rekurzivno ponovitev postopka. Pri tem gre nedvomno za poenostavitev problema proti osnovnemu primeru, saj se pred tem vsakokrat pomaknemo za korak naprej, od koder je pot do rešitve (če le-ta obstaja) zagotovo krajša. Osnovni primer je seveda situacija, ko je sosednje polje, proti kateremu smo usmerjeni, čuvaj. Do rešitve (izhoda) je od tod le še korak, zato na tem mestu ustavimo rekurzijo. Kaj pa, če rešitev ne obstaja? Tudi v tem primeru moramo poskrbeti za to, da se rekurzija ustavi, kar se zgodi v zadnji vrstici gornjega algoritma: Če poti ni mogoče nadaljevati v nobeno od treh smeri, potem se rekurzivni klici od tod naprej ne bodo nadaljevali.

Takole je videti rekurzivna funkcija za iskanje poti po blodnjaku, zapisana v jeziku JavaScript (pri tem uporabimo tabelo `blodnjak` s strani 40):

```
function resiBlodnjak(blodnjak, x, y, dx, dy) {
  //x in y predstavljata položaj v blodnjaku,
  //dx in dy pa smer iskanja.
  var i, zacasno;
  if (blodnjak[x + dx][y + dy] == 2) {
    //Pred nami je izhod, našli smo rešitev:
    return 1;
  }
  //Obrat v levo:
  zacasno = dx;
  dx = -dy;
  dy = zacasno;
  //Iščemo v tri različne smeri:
  for (i = 0; i < 3; i = i + 1) {
    //Če je možen en korak v trenutni smeri:
    if (blodnjak[x + dx][y + dy] == 0) {
      //V blodnjak vpišemo korak, ki ga bomo preizkusili:
      blodnjak[x + dx][y + dy] = 3;
      //Poskusimo rešiti blodnjak od tod naprej:
      if (resiBlodnjak(blodnjak, x + dx, y + dy, dx, dy) == 1) {
        //V tej smeri smo našli pot do izhoda:
        return 1;
      }
      //V tej smeri ni rešitve.  Odstranimo vpisani korak:
      blodnjak[x + dx][y + dy] = 0;
    }
    //Obrat v desno:
    zacasno = -dx;
    dx = dy;
    dy = zacasno;
  }
  //Poti ni bilo mogoče najti v nobeni
  //od treh smeri.  Rešitev torej ne obstaja:
  return 0;
}
```

Prvi parameter funkcije je sklic na tabelo `blodnjak`. Pomembno je, da je to sklic, saj delamo ves čas z eno samo kopijo blodnjaka. Vanj sproti zapisujemo opravljeno pot (vpisane vrednosti tri predstavljajo pot), jo pa tudi brišemo (vpišemo nazaj vrednost nič), kadar se moramo po njej vračati. Po drugi strani je pomembno, da se ostali štirje parametri podajajo po vrednosti. Parametra `x` in `y` predstavljata trenutni položaj v blodnjaku, parametra `dx` in `dy` pa predstavljata trenutno smer v blodnjaku. Ti parametri so lokalne spremenljivke funkcije, ki hranijo vrednosti položaja in smeri, ki smo jih imeli v trenutku tega klica. Če poti v to smer kasneje ne najdemo, se rekurzivni klici odvijejo nazaj in shranjene

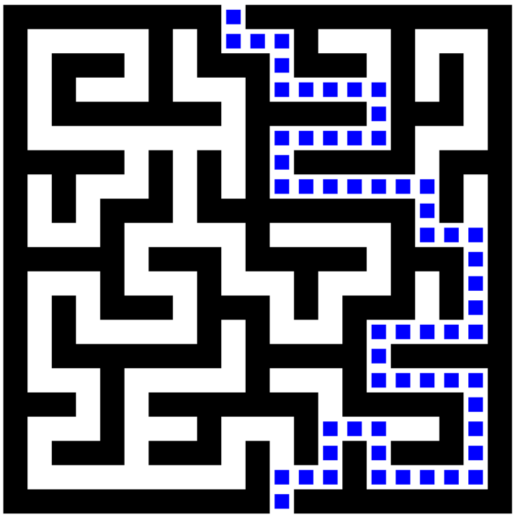
vrednosti poti in položaja v lokalnih spremenljivkah uporabimo, da izbrišemo korak, za katerega smo ugotovili, da ne gre v smeri rešitve.

Funkcija `resiBlodnjak` vrne vrednost ena v primeru, da smo našli izhod (osnovni primer). Potem se obrnemo v levo in v ponavljalnem stavku `for` po vrsti poskusimo napraviti tri korake: v levo, naprej in v desno. Če je korak možen (pred nami je vrednost nič), potem ta korak vpišemo v blodnjak (vpišemo vrednost tri) ter poskusimo rešiti blodnjak iz tega novega položaja z rekurzivnim klicem funkcije `resiBlodnjak`. Če funkcija vrne vrednost ena, vemo, da smo pot našli. Zato končamo klic funkcije in ob tem vrnemo vrednost ena. Tako se rekurzivni klici končajo eden za drugim in v blodnjaku ostane vpisana rešitev. Če rekurzivni klic funkcije `resiBlodnjak` vrne vrednost nič, vemo, da iskanje ni bilo uspešno. V tem primeru najprej iz blodnjaka odstranimo korak, ki smo ga pred tem vpisali (nazaj vpišemo vrednost nič). Potem se obrnemo v desno in, če nismo preizkusili že vseh treh smeri, postopek ponovimo. Če se stavek `for` konča, ne da bi našli pot v katerikoli od treh možnih smeri, potem pot ne obstaja in funkcija vrne vrednost nič.

In še klic funkcije, da preizkusimo njeno delovanje:

```
if (resiBlodnjak(blodnjak, 0, 10, 1, 0) == 1) {
  console.log("Našli smo pot!");
}
else {
  console.log("Pot skozi blodnjak ne obstaja.");
}
```

Program je našel skozi blodnjak naslednjo pot, ki ne obiskuje slepih rokavov, kakor jih je obiskovala pot v rešitvi na strani 41:



Če želite program preizkusiti na računalniku, lahko za prikaz rešenega blodnjaka v brskalniku uporabite funkcijo `narisiBlodnjak`, ki jo najdete v dodatku C.

Naloga 7.2 Predelajte program, ki po postopku vzvratnega sledenja reši blodnjak, tako da bo program v primeru, da skozi blodnjak obstaja več poti, našel vse poti.

Namig: Uporabite globalno spremenljivko, v katero vpišete zaporedno številko rešitve, ki vas zanima. Program naj za začetek prikaže zgolj rešitev, katere zaporedno številko ste mu podali (za prikaz rešitve uporabite funkcijo `narisiBlodnjak`, ki jo najdete v dodatku C).

7.7 Naloge

Za vajo rešite še naslednje naloge:

Naloga 7.3 Napišite rekurzivno funkcijo, ki sešteje vrednosti vseh elementov v tabeli. Funkcija naj kot parametera sprejme tabelo in njeno dolžino.

Pomoč: Izhajajte iz naslednjega rekurzivnega opisa algoritma:

$\text{vsota}(\boldsymbol{t}, 1) = t_0$ (osnovni primer),
$\text{vsota}(\boldsymbol{t}, n) = t_{n-1} + \text{vsota}(\boldsymbol{t}, n-1)$.

Naloga 7.4 V razdelku 7.2 smo videli, da izkazuje računanje členov Fibonaccijevega zaporedja z neposredno uporabo njegove rekurzivne definicije eksponentno časovno zahtevnost. Za vajo napišite rekurzivno funkcijo za računanje členov Fibonaccijevega zaporedja, ki bo sama sebe klicala le enkrat, s čimer se bodo členi zaporedja računali z linearno časovno zahtevnostjo $T(n) = \mathcal{O}(n)$.

Pomoč: Takšna funkcija mora sprejeti tri parametre, kakor kaže naslednji primer klica funkcije za izračun člena F_{64}:

```
console.log(Fib(64, 0, 1)); //izpiše 10610209857723
```

Naloga 7.5 Napišite rekurzivno funkcijo, ki v podani tabeli poišče element z največjo vrednostjo. Funkcija naj kot prvi parameter sprejme tabelo, kot drugi parameter pa njeno dolžino. Pri reševanju uporabite naslednjo funkcijo, ki vrne vrednost večjega od dveh podanih parametrov:

```
function vecji(a, b) {
  if (a > b) {
    return a;
  }
  return b;
}
```

Pomoč: Izhajajte iz naslednjega rekurzivnega opisa algoritma:

$\text{maks}(\boldsymbol{t}, 1) = t_0$ (osnovni primer),
$\text{maks}(\boldsymbol{t}, n) = \text{vecji}(t_{n-1}, \text{maks}(\boldsymbol{t}, n-1))$.

Naloga 7.6 Z uporabo vzvratnega sledenja poiščite vsaj eno od rešitev problema osmih dam.

Problem osmih dam (angl. eight queens puzzle) je znan šahovski problem, ki zahteva postavitev osmih dam na šahovnico tako, da se med seboj ne napadajo. Z drugimi besedami to pomeni, da v nobeni vrstici, stolpcu ali diagonali šahovnice velikosti 8×8 polj ne sme biti več kot ena dama.

Pomoč: Pri reševanju izhajajte iz naslednjega rekurzivnega postopka, ki skuša postaviti damo v vrstico n ($0 \leq n \leq 7$):

> $n = 8$ (osnovni primer; postavljenih je vseh osem dam).
> Ponovi za vsako polje n-te vrstice:
> Če polje ni napadeno, nanj postavi damo in poskušaj rešiti problem za naslednjo vrstico $(n + 1)$. Če je problem rešen, se vrni iz funkcije, sicer odstrani nazadnje postavljeno damo.
> Če dame ni bilo mogoče postaviti na nobeno od osmih polj, potem rešitev ne obstaja.

Dodatek: Podobno kot v nalogi 7.2 na strani 130 poskusite poiskati vse možne rešitve problema osmih dam.

Naloga 7.7 Z uporabo vzvratnega sledenja napišite program, ki reši logično uganko sudoku. Cilj uganke je zapolniti kvadratno mrežo velikosti 9×9 polj s številkami od ena do devet. Vsaka od številk se lahko v vsaki vrstici, stolpcu in enem od devetih kvadratov velikosti 3×3 pojavi le enkrat. Na začetku uganke so nekatere številke že vpisane, in sicer tako, da obstaja le ena možna rešitev uganke. Teh številk ni dovoljeno spreminjati.

Pri programiranju lahko za preizkus uporabite naslednji sudoku:

	7		4		3			
	8	6		2	9	7		
1	2			7		6	4	
	5				6		9	
	6			3			2	
	4		2				6	
	3	8		4			7	2
		7	3	9		5	8	
			7		5		3	

V programu uporabite sudoku v obliki dvorazsežnostne tabele, kjer ničle predstavljajo prazna polja, v katera je treba vpisati številke:

```
sudoku = [
  [0, 7, 0, 4, 0, 3, 0, 0, 0],
  [0, 8, 6, 0, 2, 9, 7, 0, 0],
  [1, 2, 0, 0, 7, 0, 6, 4, 0],
  [0, 5, 0, 0, 0, 6, 0, 9, 0],
```

```
    [0, 6, 0, 0, 3, 0, 0, 2, 0],
    [0, 4, 0, 2, 0, 0, 0, 6, 0],
    [0, 3, 8, 0, 4, 0, 0, 7, 2],
    [0, 0, 7, 3, 9, 0, 5, 8, 0],
    [0, 0, 0, 7, 0, 5, 0, 3, 0]
];
```

Pomoč: Progam naj najprej poskusi vstaviti v prvo prazno polje vrednost ena. Če to ni možno, naj poskusi z vrednostjo dve, in tako naprej, dokler ne pride do vrednosti, ki jo je možno vstaviti. Potem naj se premakne na naslednje prazno polje in ponovi postopek. Če v določeno polje ni mogoče vstaviti nobene od devetih vrednosti, potem se mora vrniti na prejšnje polje in tam poskusiti z naslednjo možno vrednostjo. Ko jo najde, gre spet za eno polje naprej in od tam začne znova poskušati od vrednosti ena naprej.

PRAZNA STRAN

A

UPORABLJENI OPERATORJI IN MATEMATIČNE FUNKCIJE

V tem dodatku povzemamo operatorje in matematične funkcije jezika JavaScript, ki smo jih srečali v učbeniku. V naslednji tabeli so našteti operatorji s kratkimi opisi:

Operator	Opis
`+ -`	Aritmetični operaciji seštevanja in odštevanja oziroma predznak.
`* /`	Aritmetični operaciji množenja in deljenja.
`%`	Ostanek pri celoštevilskem deljenju (glej opis na strani 13).
`=`	Spremenljivki na svoji levi priredi vrednost izraza na svoji desni.
`==`	Ali sta leva in desna stran enaki?
`!=`	Ali sta leva in desna stran različni?
`>`	Ali je vrednost na levi večja od vrednosti na desni?
`<`	Ali je vrednost na levi manjša od vrednosti na desni?
`>=`	Ali je vrednost na levi večja ali enaka vrednosti na desni?
`||`	Povezuje dve trditvi z logičnim operatorjem ALI (disjunkcija).
`&&`	Povezuje dve trditvi z logičnim operatorjem IN (konjunkcija).

Zadnja dva operatorja iz gornje tabele uporabimo za povezovanje dveh primerjalnih izrazov (trditev) v eno samo trditev. Dve trditvi, povezani z logičnim operatorjem ALI, dasta pravilno trditev, če je vsaj ena od obeh trditev pravilna. Primer v naslednji tabeli prikazuje štiri možne izide:

`(x)`	`(y)`	`(x < 4)`	`(y > 10)`	`(x < 4 || y > 10)`
5	10	ne	ne	ne
6	12	ne	da	da
-5	7	da	ne	da
0	23	da	da	da

Dve trditvi, povezani z logičnim operatorjem IN, dasta pravilno trditev samo, če sta obe trditvi pravilni. Primer v naslednji tabeli prikazuje štiri možne izide:

`(a)`	`(b)`	`(a == 0)`	`(b != 0)`	`(a == 0 && b != 0)`
-4	0	ne	ne	ne
7	-2	ne	da	ne
0	0	da	ne	ne
0	5	da	da	da

V Jeziku JavaScript obstaja matematični objekt `Math`, ki smo ga v tem učbeniku uporabljali za računanje matematičnih funkcij. V naslednji tabeli je seznam nekaterih konstant in funkcij tega objekta:

Funkcija	**Opis**
`Math.E`	Eulerjevo število e oz. osnova naravnega logaritma (2,718281828459045)
`Math.PI`	konstanta π (3,141592653589793)
`Math.abs(x)`	absolutna vrednost spremenljivke x
`Math.cos(x)`	kosinus kota x
`Math.log(x)`	naravni logaritem spremenljivke x
`Math.floor(x)`	najbližja celoštevilsko vrednost, ki je še manjša od x
`Math.pow(x, y)`	potenca (angl. power) x^y
`Math.random()`	naključna vrednost s pol odprtega intervala $[0, 1)$
`Math.sin(x)`	sinus kota x
`Math.sqrt(x)`	kvadratni koren (angl. square root) spremenljivke x
`Math.tan(x)`	tangens kota x

Opomba: Argumenti kotnih funkcij morajo biti podani v radianih.

B

REŠITVE NEKATERIH NALOG

V tem dodatku lahko najdete nekaj programov, ki rešijo zahtevnejše naloge iz učbenika. Programi seveda ne predstavljajo edine možne rešitve, kakor tudi niso nujno najboljše rešitve zastavljenih problemov. Programi so brez razlage in opomb, s čimer vam še vedno ostane izziv, da razvozlate, kako natanko delujejo. Takšnemu postopku razčlenjevanja sistema z namenom, da bi razumeli, kako deluje, pravimo ***obratno*** ali ***vzvratno inženirstvo*** (angl. reverse engineering). V resničnem življenju obstaja mnogo razlogov za izvajanje obratnega inženirstva, najmočnejši motivi pa izvirajo iz tržnih in vojaških interesov.

B.1 Naloga 3.3 (stran 30)

```
  IBAN = [281856n, 192002134567892n];  //napačna
//IBAN = [281856n, 192001234567892n];  //pravilna
//IBAN = [281856n, 1920012345678929n]; //napačna


ok = 0;
if (IBAN[1] <= 999999999999999n) {
  preveri = IBAN[1] * 1000000n + IBAN[0];
  if (preveri % 97n == 1n) {
   ok = 1;
  }
}
```

```
if (ok == 0) {
  console.log("Vnešena številka IBAN je napačna.");
}
```

B.2 Naloga 3.4 (stran 36)

```
t = [
  [1.3, 0.1, 4.6, 3.8, 0],
  [0.3, 1.1, 0],
  [1.5, 2.5, 0.4, 0],
  [0]
];
vsota = 0;
for (i = 0; t[i][0] != 0; i = i + 1) {
  for (j = 0; t[i][j] != 0; j = j + 1) {
    vsota = vsota + t[i][j];
  }
}
console.log(vsota);
```

B.3 Naloga 3.12 (stran 43)

```
p = [1];
n = 2;
for (i = 1; i <= n; i = i + 1) {
  for (j = i - 1; j > 0; j = j - 1) {
    p[j] = p[j] + p[j - 1];
  }
  p[i] = 1;
}
console.log(p);
```

B.4 Naloga 3.14 (stran 44)

```
kvadrat = [
  [1, 2, 3, 4],
  [2, 3, 4, 1],
  [3, 4, 1, 2],
  [4, 1, 2, 3]
];
n = 4;
jeLatinski = 1;
pom = [];
for (prelet = 0; prelet < 2; prelet = prelet + 1) {
  for (i = 0; i < n; i = i + 1) {
    for (j = 0; j < n; j = j + 1) {
      pom[j] = 0;
    }
```

```
    for (j = 0; j < n; j = j + 1) {
      if (prelet == 0) {
        vredn = kvadrat[i][j];
      }
      else {
        vredn = kvadrat[j][i];
      }
      if (vredn == pom[vredn - 1]) {
        jeLatinski = 0;
      }
      else {
        pom[vredn - 1] = vredn;
      }
    }
  }
}
if (jeLatinski == 1) {
  console.log("To je latinski kvadrat.");
}
else {
  console.log("To ni latinski kvadrat.");
}
```

B.5 Naloga 4.1 (stran 52)

```
1 2 3 6 7 14
```

B.6 Naloga 4.6 (stran 63)

```
function uredi(t) {
  var i, j, zacasno;
  if (t[0] == 0) {
    return;
  }
  for (i = 0; t[i + 1] != 0; i = i + 1) {
    for (j = 0; t[j + 1] != 0; j = j + 1) {
      if (t[j] > t[j + 1]) {
        zacasno = t[j];
        t[j] = t[j + 1];
        t[j + 1] = zacasno;
      }
    }
  }
}

function uredi(t) {
  var i, j, zacasno, urejeno;
  if (t[0] == 0) {
    return;
  }
  urejeno = 0;
  while (urejeno == 0) {
    urejeno = 1;
```

```
    for (j = 0; t[j + 1] != 0; j = j + 1) {
      if (t[j] > t[j + 1]) {
        zacasno = t[j];
        t[j] = t[j + 1];
        t[j + 1] = zacasno;
        urejeno = 0;
      }
    }
  }
}
```

B.7 Naloga 4.10 (stran 65)

```
function izberiCifro(vredn, mesto) {
  mesto = 3 - mesto;
  while (mesto > 0) {
    vredn = vredn - (vredn % 10);
    vredn = vredn / 10;
    mesto = mesto - 1;
  }
  return vredn % 10;
}

function dodajElement(t, element) {
  var i;
  for (i = 0; t[i] != 0; i = i + 1) {}
  t[i] = element;
  t[i + 1] = 0;
}

function desetiskoVRimsko(desetisko, rimsko) {
  var i, j, k, c, rimIndeksi, rimSimboli;
  rimIndeksi = [[-1], [0, -1], [0, 0, -1], [0, 0, 0, -1],
                [0, 1, -1],[1, -1],[1, 0, -1], [1, 0, 0, -1],
                [1, 0, 0, 0, -1], [0, 2, -1]];
  rimSimboli = [['C', 'D', 'M'], ['X', 'L', 'C'], ['I', 'V', 'X']];
  if (desetisko < 1 || desetisko > 3999) {
    return -1; //Neveljavna vrednost, ni bilo pretvorbe.
  }
  c = izberiCifro(desetisko, 0);
  for (i = 0; i < c; i = i + 1) {
    dodajElement(rimsko, 'M');
  }
  for (i = 0; i < 3; i = i + 1) {
    c = izberiCifro(desetisko, i + 1);
    for (j = 0; rimIndeksi[c][j] != -1; j = j + 1) {
      k = rimIndeksi[c][j];
      dodajElement(rimsko, rimSimboli[i][k]);
    }
  }
  return 0; //Veljavna vrednost, uspešna pretvorba.
}

rim = [0];
if (desetiskoVRimsko(2018, rim) == 0) {
  console.log(rim);
}
```

```
else {
  console.log("Te vrednosti ne znam pretvoriti.");
}
```

B.8 Nalogi 5.7 in 5.8 (stran 92)

```
function jePrestopno(leto) {
  if (leto % 4 != 0) {
    return 0;
  }
  else if (leto % 100 != 0) {
    return 1;
  }
  else if (leto % 400 != 0) {
    return 0;
  }
  else {
    return 1;
  }
}
datum = Object();
datum.nastavi = function(sekunde) {
  datum.sekunde = sekunde;
};
datum.vrniUro = function(u) {
  var sekunde;
  sekunde = datum.sekunde % (24 * 3600);
  u[0] = Math.floor(sekunde / 3600);
  u[1] = Math.floor(sekunde / 60) % 60;
  u[2] = sekunde % 60;
};
datum.vrniDan = function(d) {
  var sekunde, odstej, meseci;
  meseci = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];
  sekunde = datum.sekunde;
  d[0] = 1;
  d[1] = 1;
  d[2] = 1970;
  odstej = 365 * 24 * 3600;
  while (sekunde - odstej > 0) {
    sekunde = sekunde - odstej;
    odstej = 365 * 24 * 3600;
    d[2] = d[2] + 1;
    if (jePrestopno(d[2]) == 1) {
      odstej = odstej + 24 * 3600;
    }
  }
  odstej = meseci[0] * 24 * 3600;
  while (sekunde - odstej > 0) {
    sekunde = sekunde - odstej;
    odstej = meseci[d[1]] * 24 * 3600;
    d[1] = d[1] + 1;
    if (d[1] == 2 && jePrestopno(d[2])) {
      odstej = odstej + 24 * 3600;
    }
  }
  odstej = 24 * 3600;
```

```
  while (sekunde - odstej > 0) {
    sekunde = sekunde - odstej;
    d[0] = d[0] + 1;
  }
};
d = [];
datum.nastavi(Math.floor(Date.now() / 1000));
datum.vrniUro(d);
console.log(d);
datum.vrniDan(d);
console.log(d);
```

B.9 Naloga 6.2 (stran 108)

```
slovar = Object();
slovar.glava = Object();
slovar.glava.podatek = '?';
slovar.glava.naslednji = '?';
slovar.rep = slovar.glava;
slovar.insert = function(kljuc, vrednost) {
  var novo = Object();
  novo.podatek = [kljuc, vrednost];
  novo.naslednji = slovar.glava;
  slovar.glava = novo;
};
slovar.lookup = function(kljuc) {
  var ta;
  ta = slovar.glava;
  while (ta != slovar.rep) {
    if (ta.podatek[0] == kljuc) {
      return ta.podatek[1];
    }
    ta = ta.naslednji;
  }
  return -1;
};
slovar.delete = function(kljuc) {
  var ta;
  ta = slovar.glava;
  while (ta != slovar.rep) {
    if (ta.podatek[0] == kljuc) {
      if (ta.naslednji == slovar.rep) {
        slovar.rep = ta;
      }
      ta.podatek = ta.naslednji.podatek; //Zakaj je dovolj, da
                                         //kopiramo samo sklic
                                         //na podatek?
      ta.naslednji = ta.naslednji.naslednji;
      return;
    }
    ta = ta.naslednji;
  }
};
```

B.10 Naloga 6.3 (stran 109)

```
seznam.insertBefore = function(vredn, vrstniRed) {
  var ta, nov;
  ta = seznam.glava;
  seznam.rep.podatek = vredn;
  while (vrstniRed * vredn > vrstniRed * ta.podatek) {
    ta = ta.naslednji;
  }
  nov = Object();
  nov.podatek = ta.podatek;
  nov.naslednji = ta.naslednji;
  ta.podatek = vredn;
  ta.naslednji = nov;
  if (ta == seznam.rep) {
    seznam.rep = nov;
  }
};

var tabela = [1, 3, 6, 3, 1, -8, 1, 3, -4, -3];
var dolzina = 10;

for (i = 0; i < dolzina; i = i + 1) {
  seznam.insertBefore(tabela[i], 1);     //Uredi od najmanjšega
                                         //proti največjemu.
  //seznam.insertBefore(tabela[i], -1);  //Uredi od največjega
                                         //proti najmanjšemu.
}
for (i = 0; i < dolzina; i = i + 1) {
  tabela[i] = seznam.removeBeginning();
}

console.log(tabela);
```

B.11 Naloga 6.4 (stran 109)

```
vrsta.size = function() {
  var velikost;
  velikost = vrsta.rep - vrsta.glava;
  if (velikost < 0) {
    velikost = velikost + vrsta.dolzina;
  }
  return velikost;
};
```

B.12 Naloga 6.6 (stran 110)

```
seznam.izpis = function() {
  var ta;
  ta = seznam.glava;
```

```
  while (ta != seznam.rep) {
    console.log(ta.podatek);
    ta = ta.naslednji;
  }
};
```

B.13 Naloga 6.7 (stran 110)

```
seznam.enqueue = seznam.insertEnd;
seznam.dequeue = seznam.removeBeginning;

seznam.povprecje = function() {
  var ta, n, vsota;
  vsota = 0;
  n = 0;
  ta = seznam.glava;
  while (ta != seznam.rep) {
    vsota = vsota + ta.podatek;
    n = n + 1;
    ta = ta.naslednji;
  }
  if (n == 0) {
    return 0;
  }
  return vsota / n;
};

zaporedje = [2, 4, 6, 7, 8, 9];
dolzina = 6;
n = 3;
drsecePovprecje = [];
i = 0;
for (k = 0; k < dolzina; k = k + 1) {
  seznam.enqueue(zaporedje[k]);
  if (k >= n - 1) {
    drsecePovprecje[i] = seznam.povprecje();
    i = i + 1;
    seznam.dequeue();
  }
}
console.log(drsecePovprecje);
```

B.14 Naloga 7.1 (stran 116)

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

B.15 Naloga 7.4 (stran 131)

```
function Fib(n, f0, f1) {
  if (n == 0) {
    return f0;
  }
  return Fib(n - 1, f1, f0 + f1);
}
```

B.16 Naloga 7.6 (stran 132)

```
sahovnica = [
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0, 0, 0, 0]
 ];

function jeVarno(sah, vrst, stolp) {
  var i, j;
  for (i = 0; i < vrst; i = i + 1) {
    if (sah[i][stolp] == 1) {
      return 0;
    }
  }
  i = vrst;
  j = stolp;
  while (i >= 0 && j >= 0) {
    if (sah[i][j] == 1) {
      return 0;
    }
    i = i - 1;
    j = j - 1;
  }
  i = vrst;
  j = stolp;
  while (i >= 0 && j < 8) {
    if (sah[i][j] == 1) {
      return 0;
    }
    i = i - 1;
    j = j + 1;
  }
  return 1;
}

function postaviKraljice(sah, n) {
  var i;
  if (n == 8) {
    return 1;
  }
```

```
  for (i = 0; i < 8; i = i + 1) {
    if (jeVarno(sah, n, i) == 1) {
      sah[n][i] = 1;
      if (postaviKraljice(sah, n + 1) == 1) {
        return 1;
      }
      sah[n][i] = 0;
    }
  }
  return 0;
}

postaviKraljice(sahovnica, 0);
for (i = 0; i < 8; i++) {
  console.log(sahovnica[i]);
}
```

B.17 Naloga 7.7 (stran 132)

```
sudoku = [
  [0, 7, 0, 4, 0, 3, 0, 0, 0],
  [0, 8, 6, 0, 2, 9, 7, 0, 0],
  [1, 2, 0, 0, 7, 0, 6, 4, 0],
  [0, 5, 0, 0, 0, 6, 0, 9, 0],
  [0, 6, 0, 0, 3, 0, 0, 2, 0],
  [0, 4, 0, 2, 0, 0, 0, 6, 0],
  [0, 3, 8, 0, 4, 0, 0, 7, 2],
  [0, 0, 7, 3, 9, 0, 5, 8, 0],
  [0, 0, 0, 7, 0, 5, 0, 3, 0]
];

function jeVpisMozen(sudoku, vrst, stolp, n) {
  var i, j, iStart, jStart;
  for (i = 0; i < 9; i = i + 1) {
    if (n == sudoku[vrst][i] || n == sudoku[i][stolp]) {
      return 0;
    }
  }
  iStart = vrst - vrst % 3;
  jStart = stolp - stolp % 3;
  for (i = iStart; i < iStart + 3; i = i + 1) {
    for (j = jStart; j < jStart + 3; j = j + 1) {
      if (n == sudoku[i][j]) {
        return 0;
      }
    }
  }
  return 1;
}

function resiSudoku(sudoku, polje) {
  var n, vrst, stolp;
  if (polje == 81) {
    return 1;
  }
  vrst = Math.floor(polje / 9);
  stolp = polje % 9;
```

```
  if (sudoku[vrst][stolp] != 0) {
    return resiSudoku(sudoku, polje + 1);
  }
  for (n = 1; n < 10; n = n + 1) {
    if (jeVpisMozen(sudoku, vrst, stolp, n) == 1) {
      sudoku[vrst][stolp] = n;
      if (resiSudoku(sudoku, polje + 1) == 1) {
        return 1;
      }
      sudoku[vrst][stolp] = 0;
    }
  }
  return 0;
}

resiSudoku(sudoku, 0);
for (i = 0; i < 9; i++) {
  console.log(sudoku[i]);
}
```

PRAZNA STRAN

C

KODA ZA PRIKAZ BLODNJAKA

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Blodnjak</title>
</head>
<body>
  <canvas id="blodnjak" width="800" height="800"></canvas>
  <script>

stranica = 20;

function risiKvadratek(x, y, dim, rob, barva) {
  var c, ctx;
  c = document.getElementById("blodnjak");
  ctx = c.getContext("2d");
  ctx.fillStyle = barva;
  ctx.fillRect(y * dim + rob, x * dim + rob,
               dim - 2 * rob, dim - 2 * rob);
}

function narisiBlodnjak(bl) {
  var i, j;
  for (i = 0; i < bl.length; i++) {
    for (j = 0; j < bl[0].length; j++) {
```

```
      if (bl[i][j] == 1) {
        risiKvadratek(i, j, stranica, 0, "black");
      }
      if ((bl[i][j] == 3)) {
        risiKvadratek(i, j, stranica,
          Math.floor(stranica/5), "blue");
      }
    }
  }
}

blodnjak = [
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
[2, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 2],
[2, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 2],
[2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 2],
[2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2],
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
];
pol = [1, 10];
smer = [1, 0];

while (blodnjak[pol[0]][pol[1]] != 2) {
  zacasno = smer[0];
  smer[0] = -smer[1];
  smer[1] = zacasno;
  while (blodnjak[pol[0] + smer[0]][pol[1] + smer[1]] == 1) {
    zacasno = -smer[0];
    smer[0] = smer[1];
    smer[1] = zacasno;
  }
  blodnjak[pol[0]][pol[1]] = 3;
  pol[0] = pol[0] + smer[0];
  pol[1] = pol[1] + smer[1];
}

narisiBlodnjak(blodnjak);

  </script>
</body>
</html>
```

VIRI

[1] Fajfar, I. (2016). *Start programming using HTML, CSS, and JavaScript*, Chapman & Hall/CRC.

[2] Anželj, G. s sod. (2015). *Računalništvo in informatika 1: e-učbenik za informatiko v gimnaziji*, Založba Univerze na Primorskem, https://lusy.fri.uni-lj.si/ucbenik/book/index.html.

[3] Aho, A. V., Hopcroft, J. E., Ullman, J. D. (1974). *The design and analysis of computer algorithms*, Addison–Wesley Publishing Company.

[4] Ammeraal, L. (1994). *Programs and data structures in C*, John Wiley & Sons Ltd.

[5] Cormen, T. H. s sod. (2009) *Introduction to algorithms*, The MIT Press.

PRAZNA STRAN

O AVTORJU

Iztok Fajfar je dobil prvi osebni računalnik v začetku osemdesetih let prejšnjega stoletja: ZX Spectrum z neverjetnimi 48 KB RAMa. Računalniki so kmalu postali njegova strast in nepogrešljivi spremljevalci. Sodeloval je pri mnogih projektih razvoja programske opreme tako za domača kot tudi tuja podjetja. Njegovo raziskovalno delo vključuje evolucijske algoritme, zlasti genetsko programiranje. Fajfar poučuje računalniško programiranje na vseh ravneh, od strojnega do objektno usmerjenega, in na vseh stopnjah študija. Je tudi avtor več učbenikov, med drugim učbenika *Start programming using HTML, CSS, and JavaScript*, ki ga je izdala priznana mednarodna založba. Zaposlen je kot izredni profesor na Fakulteti za elektrotehniko.

PRAZNA STRAN

IZ RECENZIJE

V pričujočem učbeniku avtor vešče prepleta razlago posameznih delov programskega jezika z razlago algoritmov. Zasnoval ga je tako, da izhaja iz reševanja matematičnih, fizikalnih in drugih praktičnih problemov, ki so inženirjem blizu. S tem se izogne suhoparnemu kataloškemu naštevanju elementov programskega jezika in tehnik gradnje algoritmov. Zelo dobrodošla je tudi predstavitev najpomembnejših tehnik načrtovanja programov in iskanja napak v njih. Učbenik zato predstavlja uporaben pripomoček, s katerim bodo študentje lažje in bolj poglobljeno spremljali potek predavanj.

Iz vsebinskih poudarkov in razlag je razvidno, da je avtor v letih poučevanja prejel prenekatero vprašanje iz klopi in zato dobro ve, kakšne dileme se študentom najpogosteje porajajo. Po principu »manj je več« se namenoma izogne podrobni razlagi programskega jezika ter teoretičnemu uvodu v algoritme z matematičnimi izpeljavami in dokazi. Kot učno orodje izbere jezik JavaScript, ki predstavlja začetniku majhno kognitivno breme in je hkrati dobro izhodišče za učenje mnogih danes prevladujočih programskih jezikov. Razbremenjen nebistvenih podrobnosti, se bo tako bralec lažje uril v algoritmičnemu mišljenju. Prav algoritmično mišljenje pa učbenik postavlja v ospredje, saj je v nasprotju s programskimi jeziki, ki se menjajo, mnogo trajnejše.

doc. dr. Boštjan Slivnik, UL FRI