[1] Опис лиценци Креативне заједнице доступан је на адреси creativecommons.org.rs/?page_id=74.

UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA MATEMATIKU I INFORMATIKU

Miloš Stojaković
Mirjana Mikalački

UVOD U PROGRAMIRANJE ZA MATEMATIČARE

Novi Sad, 2017.

Naziv: UVOD U PROGRAMIRANJE ZA MATEMATIČARE

Autori: DR MILOŠ STOJAKOVIĆ
redovni profesor PMF-a u Novom Sadu
DR MIRJANA MIKALAČKI
docent PMF-a u Novom Sadu

Recenzenti: DR DRAGAN MAŠULOVIĆ
redovni profesor PMF-a u Novom Sadu
DR DEJAN VUKOBRATOVIĆ
vanredni profesor FTN-a u Novom Sadu

Izdavač: PRIRODNO-MATEMATIČKI FAKULTET U NOVOM SADU

Za izdavača: PROF. DR MILICA PAVKOV HRVOJEVIĆ, dekan Fakulteta

Publikovanje i upotrebu ovog udžbenika je odobrilo Nastavno-naučno veće Prirodno-matematičkog fakulteta na svojoj sednici 23.11.2017.

ISBN: 978-86-7031-358-3

CIP - Каталогизација у публикацији
Библиотека Матице српске, Нови Сад

004.42(075.8)(0.034.2)

СТОЈАКОВИЋ, Милош
Uvod u programiranje za matematičare [Elektronski izvor] / Miloš Stojaković, Mirjana Mikalački. - Нови Сад : Природно-математички факултет, 2017.

Dostupno i na: https://www.pmf.uns.ac.rs/studije/epublikacije/matinf/stojakovic_mikalacki_uvod_u_programiranje_za_matematicare.pdf

ISBN 978-86-7031-358-3
1. Микалачки, Мирјана [аутор]
а) Програмирање
COBISS.SR-ID **319320839**

Sadržaj

Predgovor

Ova knjiga je prvenstveno namenjena matematičarima, ali i svima onima koji su zainteresovani da uče da programiraju kroz rešavanje matematičkih problema. Tekst je napisan tako da u potpunosti prati nastavni plan predmeta *Programiranje 1*, koji se izvodi na Prirodno-matematičkom fakultetu u Novom Sadu. Predstavljeni sadržaji su, prema tome, usklađeni sa nivoom znanja i interesovanjima studenata matematike na prvoj godini osnovnih studija.

Materijal je koncipiran i organizovan kao prvi kurs iz programiranja – pretpostavlja se da čitalac nema programersko predznanje. Pristup rešavanju problema je naglašeno algoritamski, a programski jezik koji se koristi je C#, u današnje vreme popularan i široko dostupan programski jezik.

Knjiga je podeljena u sedam glava, gde se u *prvoj* glavi obrađuju osnovni koncepti vezani za programiranje, kao što su prosti tipovi podataka, dodela, učitavanje i ispis, ali i prepoznavanje i uklanjanje grešaka. U *drugoj* glavi pažnja je usmerena na kontrolne strukture toka programa – grananja i cikluse. U *trećoj* glavi uvode se nizovi i matrice, kao i korišćenje ovih struktura za skladištenje podataka. *Četvrta* glava posvećena je raznim vrstama matematičkih problema, između ostalog i iz teorije brojeva. U *petoj* glavi uvodi se koncept metoda i rekurzije. *Šesta* glava je posvećena rešavanju kombinatornih problema, dok se *sedma* glava bavi primenom jedne od osnovnih statističkih metoda – tačkastom ocenom verovatnoće.

Tekst knjige je prošao kroz više iteracija dok nije dobio oblik u kom je pred vama. Posebnu zahvalnost dugujemo Marku Saviću i Nikoli Trkulji, koji su brojnim komentarima značajno doprineli kvalitetu ovog teksta. Takođe, zahvaljujemo se recenzentima na korisnim primedbama i sugestijama.

Novi Sad, 15.10.2017. *Miloš Stojaković i Mirjana Mikalački*

1 Uvodni pojmovi

U prvoj glavi uvešćemo neke osnovne pojmove na koje se ostatak knjige oslanja. Nakon prvog upoznavanja sa programskim jezikom kroz nekoliko primera, biće reči o elementarnim, a istovremeno i često korišćenim programskim konceptima, kao što su deklarisanje i korišćenje promenljivih, ispis i učitavanje.

1.1 Programski jezik

Svi programi u ovoj knjizi napisani su u programskom jeziku C#, pomoću softvera *Microsoft Visual Studio Express 2013*. Pritom, pisaćemo isključivo tzv. konzolne aplikacije, *ConsoleApplication*. Nećemo ulaziti u specifičnosti softvera koji koristimo, imajući u vidu da se svi programi koje ćemo ovde predstaviti mogu izvršavati i na svim ostalim (do sada objavljenim) verzijama *Visual Studio*-a.

Kada se kreira novi projekat tipa *ConsoleApplication* u editoru se otvara datoteka *Program.cs* u koju ćemo unositi programski kôd. Ta datoteka nije prazna, već sadrži sledeći tekst.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrviProjekat
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Da bismo razumeli značenje i namenu onoga što ovde vidimo, potrebno nam je određeno znanje o programskom jeziku C#. Deo tog znanja ćemo kasnije usvojiti, a deo prevazilazi okvire ove knjige, a zainteresovani čitalac može pročitati više u [1; 2; 3]. U ovom momentu, na samom početku, daćemo samo kratka i ne previše formalna objašnjenja.

Naredbe u prvih pet redova (koje počinju sa `using`) omogućavaju korišćenje navedenih pet standardnih biblioteka u kojima se nalaze neki osnovni metodi, tipovi i operacije (sa ovim konceptima ćemo se detaljnije upoznati kasnije). Na kraju svakog reda stoji tačka-zarez koja označava kraj naredbe.

U redu koji počinje sa `namespace` navodi se ime projekta, u našem slučaju `PrviProjekat`. Početak projekta je označen otvorenom vitičastom zagradom a kraj zatvorenom vitičastom zagradom, i sve što je navedeno između ovih zagrada pripada tom projektu. Unutar projekta se nalazi ime programa iza rezervisane reči `class`, a zatim sledi ono što pripada tom programu, opet unutar vitičastih

zagrada. U našem novootvorenom projektu program sadrži samo prazan metod `Main`, čije zaglavlje vidimo u redu koji počinje sa `static`.

Kada se program pokrene, izvršava se kôd u `Main` metodu, tj. ono što napišemo unutar vitičastih zagrada koje slede nakon zaglavlja metoda `Main`. S obzirom da će svi programi u ovoj knjizi imati potpuno istu formu **izvan** klase `class Program`, u daljem tekstu navodićemo samo tu klasu, pretpostavljajući da je sve izvan nje isto kao i u (novootvorenom praznom) projektu koji smo iznad naveli.

1.2 Prvi program

Prvi program koji ćemo napisati biće veoma jednostavan.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Prvi program!");
    }
}
```

Naredba `Console.WriteLine` služi za ispis onoga što je kao parametar navedeno u zagradama, te nakon izvršavanja ovog programa dobijamo sledeći ispis.

```
Prvi program!
```

Treba imati u vidu da se velika i mala slova razlikuju. Na kraju reda stoji tačka-zarez, označavajući kraj naredbe. Više naredbi unutar vitičastih zagrada zovemo *blok naredbi*. Razmak je separator, a više razmaka tretira se isto kao jedan razmak. Samim tim, iako nam „nazubljenost“ programa (uvlačenje linija uzastopnim razmacima) nije bitna za funkcionalnost, veoma je bitna za preglednost.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Jedan!");
        Console.WriteLine("Dva!");
        Console.WriteLine("Tri...");
    }
}
```

Nakon izvršavanja, dobijamo sledeći ispis.

```
Jedan!
Dva!
Tri...
```

1.3 Tipovi podataka

U jeziku C#, svaki podatak sa kojim radimo mora imati svoj *tip*. Mi ćemo koristiti samo tzv. *ugrađene tipove*. Za svaki tip je karakteristična veličina koju vrednost tog tipa zauzima u memoriji, kao i opseg vrednosti koje on obuhvata. Neki od osnovnih ugrađenih tipova su prikazani u tabeli 1, a za više informacija o ostalim standardnim tipovima upućujemo čitaoca da pogleda [2; 4].

Tip u C#	Veličina (u bajtovima)	Opseg vrednosti
`short`	2	celi brojevi iz intervala $[-32768, 32767]$
`int`	4	celi brojevi iz intervala $[-2^{31}, 2^{31}-1]$
`long`	8	celi brojevi iz intervala $[-2^{63}, 2^{63}-1]$
`float`	4	realni brojevi koji obuhvataju vrednosti (približno) od $-3{,}4 \cdot 10^{38}$ do $-1{,}5 \cdot 10^{-45}$, i vrednosti od $1{,}5 \cdot 10^{-45}$ do $3{,}4 \cdot 10^{38}$, i to sa 7 značajnih cifara
`double`	8	realni brojevi koji obuhvataju vrednosti (približno) od $-1{,}8 \cdot 10^{308}$ do $-5{,}0 \cdot 10^{-324}$, i vrednosti od oko $5{,}0 \cdot 10^{-324}$ do $1{,}8 \cdot 10^{308}$, i to sa 15 značajnih cifara
`char`	2	znakovni tip
`bool`	1	logički tip

Tabela 1. Neki osnovni tipovi podataka u C#

Mi ćemo za predstavljanje celih brojeva koristiti tip `int`, dok ćemo za rad sa realnim brojevima koristiti tip `double`.

1.4 Promenljive i dodela

Promenljive u programiranju nisu isto što i promenljive u matematici, štaviše – suštinski su drugačije, jer u programiranju promenljiva ima svoju brojnu vrednost. Svaka promenljiva ima svoj tip koji određuje koji tip podataka će promenljiva sadržati. Pre prvog korišćenja promenljive potrebno je *deklarisati* je, čime joj je dodeljen tip podataka.

```
class Program
{
    static void Main(string[] args)
    {
        int x;
        x = 3;
        Console.WriteLine(x);
    }
}
```

U prethodnom programu, deklarisali smo promenljivu `x` tipa `int`, dodelili smo joj vrednost 3, a zatim smo je ispisali. Sintaksa deklaracije promenljive ima više varijanti, navešćemo nekoliko:

```
<tip> <ime>;
<tip> <ime1>, <ime2>, <ime3>;
<tip> <ime> = <izraz>;
```

U trećem slučaju, promenljiva odmah po deklaraciji dobija i vrednost datu navedenim izrazom, koja mora biti istog tipa kao i promenljiva. *Dodela* vrednosti promenljivoj podrazumeva promenu vrednosti koju promenljiva ima. Sintaksa dodele je sledeća:

```
<ime> = <izraz>;
```

gde je `<ime>` ime promenljive kojoj se dodeljuje vrednost, dok `<izraz>` ima vrednost istog tipa kao promenljiva. Generalno gledano, različiti tipovi se ne mogu „mešati“, mada postoje izuzeci od ovog pravila – više o tome kasnije.

```
class Program
{
    static void Main(string[] args)
    {
        int p;
        double x, y;
        x = 3.141;
        y = 7.5;
        int z = 1;
        p = 11;
        Console.WriteLine(x);
        Console.WriteLine(y);
        Console.WriteLine(z);
        Console.WriteLine(p);
    }
}
```

Nakon što se ovaj program izvrši, na ekranu se ispisuje

```
3.141
7.5
1
11
```

Ime promenljive može biti bilo koja kombinacija slova, cifara i podvlake, koja počinje slovom ili podvlakom, a da pritom nije rezervisana reč (npr. promenljiva se ne može zvati `double` ili `using`). Takođe, ne smeju se koristiti ni imena nekih standardnih metoda (npr. `WriteLine`).

Kod deskriptivnog nazivanja promenljivih držaćemo se kodeksa tzv. *Kamel (Camel) notacije* – ime počinjemo malim slovom, a svaku (eventualnu) sledeću reč počinjemo velikim slovom (bez razmaka), što ilustrujemo sledećim primerom.

```
class Program
{
    static void Main(string[] args)
    {
        double približnaVrednostBrojaPi = 3.14159;
        double približnaVrednostBrojaE = 2.718;
        int brojPrstijuNaRukama = 10;
        Console.WriteLine(približnaVrednostBrojaPi);
        Console.WriteLine(približnaVrednostBrojaE);
```

```
            Console.WriteLine(brojPrstijuNaRukama);
        }
    }
```

1.5 Operatori

U prethodnim primerima videli smo kako se promenljivoj nekog tipa dodeljuje vrednost, korišćenjem operatora dodele =. Za osnovne matematičke operacije koristimo proste matematičke operatore: za sabiranje +, za oduzimanje -, za množenje *, za deljenje /. Operatori + i - se ponašaju na način na koji smo navikli u matematici u radu i sa celim i sa realnim brojevima. Isto je i sa množenjem. Operator / u radu sa celim brojevima daje količnik pri celobrojnom deljenju, a ukoliko radimo sa realnim brojevima, onda služi za realno deljenje. Npr. izraz `18 / 4` ima vrednost 4, dok izraz `18.0 / 4.0` ima vrednost `4.5`. Operator % se koristi samo u radu sa celim brojevima da izračuna ostatak pri celobrojnom deljenju, npr. izraz `18 % 4` ima vrednost 2. Zagrade (...) se koriste u brojnim izrazima za promenu prioriteta izračunavanja, slično kao u matematičkim izrazima. Više o operatorima može se naći u [4].

Treba imati u vidu da se brojne vrednosti izračunavaju i skladište u *približnoj* aritmetici, što znači da može doći do izvesnih nepreciznosti, npr. izraz `2.0/3.0` ima vrednost koja odstupa od $2/3$, zbog ograničene preciznosti zapisa promenljive u memoriji računara.

```
class Program
{
    static void Main(string[] args)
    {
        int a = 17, b = 3;
        double x = 17.0, y = 3.0;
        Console.WriteLine(a % b);
        Console.WriteLine(a / b);
        Console.WriteLine(x / y);
    }
}
```

Nakon izvršavanja ovog programa, na ekranu imamo sledeći ispis.

```
2
5
5.66666666666667
```

Kao što smo već rekli, vrednost koja se dodeljuje promenljivoj mora biti istog tipa kao i promenljiva. Izuzetak od ovog pravila je tip `int` koji se automatski konvertuje u tip `double` kada za to postoji potreba.

```
class Program
{
    static void Main(string[] args)
    {
        int i = 3;
```

```
            double x;
            x = i + 1.5;
            Console.WriteLine(x);
        }
    }
```

Prethodni program nakon izvršavanja ispisuje na ekran vrednost `4.5`.

U slučaju da želimo da zamenimo vrednosti dve promenljive istog tipa, koristimo treću pomoćnu promenljivu, takođe istog tog tipa. U sledećem programu vrši se zamena vrednosti promenljivih x i y, pa prilikom ispisa x ima vrednost 7, a y vrednost 5.

```
class Program
{
    static void Main(string[] args)
    {
        int x = 5, y = 7;
        int pomocna;

        pomocna = x;
        x = y;
        y = pomocna;

        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}
```

1.6 Relativna promena vrednosti promenljive, inkrementacija i dekrementacija

Kod dodele vrednosti promenljivoj najpre se izračunava vrednost izraza sa desne strane znaka =, da bi se potom ta vrednost smestila u promenljivu navedenu sa leve strane. Samim tim, pojavljivanje iste promenljive na obe strane nije besmisleno, štaviše, često je korisno.

```
class Program
{
    static void Main(string[] args)
    {
        int i = 3;
        i = i + 3;
        i = i * 2;
        i = i + 1;
        Console.WriteLine(i);
    }
}
```

Nakon izvršavanja ovog programa na ekranu se ispisuje vrednost `13`.

Dodavanje vrednosti na postojeću vrednost promenljive kraće se zapisuje `+=`, a ako je vrednost koja se dodaje jedan, onda koristimo `++`. Slično se obavlja i umanjivanje, pomoću `-=` i `--`. I za sve ostale navedene operacije može se koristiti skraćeni zapis, `*=`, `/=`, `%=`. Sledeći program radi isto kao i prethodni, ali se u njemu koristi skraćeni zapis.

```
class Program
{
    static void Main(string[] args)
    {
        int i = 3;
        i += 3;
        i *= 2;
        i ++;
        Console.WriteLine(i);
    }
}
```

Drugi primeri korišćenja skraćenog zapisa se mogu naći u [1; 2; 3; 4].

1.7 Ispis

Već smo pomenuli naredbu `WriteLine` za ispis teksta nakon koga dolazi do prelaska u novi red. Naredba `Write`, koja takođe služi za ispis, razlikuje se od naredbe `WriteLine` po tome što nakon ispisanog teksta na ekranu kursor ostane u istom redu u kome je bio i ispis. U naprednijoj formi ovih naredbi, u parametrima naredbe `Write`, odnosno `WriteLine` možemo navesti i neki broj parametara, čije vrednosti unutar teksta koji se ispisuje pozivamo navođenjem referenci `{0}`, `{1}`, `{2}`, itd. Pritom, referenca `{0}` odgovara prvom parametru, referenca `{1}` drugom parametru, itd.

```
class Program
{
   static void Main(string[] args)
    {
        int a = 1, b = 3;
        double x = 7.7, y = 13.0;
        Console.Write("a: {0}, x: {1},", a, x);
        Console.Write(" Tralalala! ");
        Console.WriteLine("b: {0}, y: {1}.", b, y);
        Console.WriteLine("Lalalaa...");
    }
}
```

Nakon izvršavanja, dobijamo sledeći ispis.

```
a: 1, x: 7.7, Tralalala! b: 3, y: 13.
Lalalaa...
```

Fiksna širina polja za ispis dobija se navođenjem dodatne vrednosti uz referencu promenljive, npr. `{1,5}` označava ispis drugonavedene vrednosti u polje širine 5.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("{0,5} {1,5} {2,5} {3,5}", 12, -5, 3334, 0);
        Console.WriteLine("{0,5} {1,5} {2,5} {3,5}", 5, -57, -3, 1234);
        Console.WriteLine("{0,5} {1,5} {2,5} {3,5}", 2, 2, 2, 2);
    }
}
```

Nakon izvršavanja, dobijamo sledeći ispis.

```
   12    -5  3334     0
    5   -57    -3  1234
    2     2     2     2
```

Parametri se ne moraju referencirati u redosledu u kome su navedeni, a jedan parametar se može pojaviti i više puta, što ilustrujemo sledećim primerom.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("{2} {4} {2} {3} {2} {0} {2}", 1, 2, 3, 4, 5);
    }
}
```

Nakon izvršavanja, dobijamo sledeći ispis.

```
3 5 3 4 3 1 3
```

Kod ispisa realnih brojeva moguće je ograničiti broj decimala iza decimalnog zareza navođenjem dvotačke, slova N, a zatim i broja decimala. Npr. ako navedemo `{0:N2}` ili `{0,10:N2}`, u oba slučaja broj će biti ispisan sa dve decimale, s tom razlikom što se u drugom slučaju broj ispisuje u polje širine 10. Postoje i drugi formati zapisa brojeva, o čemu se detaljnije može pročitati u [5].

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("{0:N3} {1:N3} {2:N3}", 2.22222, 0.0001, 10.1);
        Console.WriteLine("{0,15:N2} {1,15:N2} {2,15:N2}", 2.2, 0.0001, 0.1);
    }
}
```

Nakon izvršavanja, dobijamo sledeći ispis.

```
2.222 0.000 10.100
           2.20            0.00            0.10
```

1.8 Učitavanje

Učitavanje odnosno unos sa tastature vrši se pomoću `Console.ReadLine()`. Takav unos se pretvara u broj tipa `int` pomoću `int.Parse`, a u realan broj pomoću `double.Parse`. Pritom, ako unos ne odgovara tipu koji se očekuje, doći će do greške. O učitavanju promenljivih drugih tipova pogledati [1].

```
class Program
{
    static void Main(string[] args)
    {
        int i;
        double f;
        Console.Write("Unesite i=");
        i = int.Parse(Console.ReadLine());
        Console.WriteLine("i+1={0}", i + 1);
        Console.WriteLine();
        Console.Write("Unesite f=");
        f = double.Parse(Console.ReadLine());
        Console.WriteLine("Dvostruka vrednost f je: {0}", 2 * f);
    }
}
```

Navodimo primer jednog izvršavanja ovog programa.

```
Unesite i=6
i+1=7

Unesite f=12.4
Dvostruka vrednost f je: 24.8
```

1.9 Komentari

Komentari se ignorišu prilikom izvršavanja programa, i možemo ih navesti na dva načina – nakon `//` u okviru jednog reda, ili u jednom ili više redova između `/*` i `*/`. Koristimo ih da objasnimo neke delove programa, kao i da privremeno uklonimo (ali ne i obrišemo) delove programa koji nam u datom momentu nisu potrebni.

```
class Program
{
    /* Ovo je program koji
        ne radi
     nista
          posebno... */
    static void Main(string[] args)
    {
        int i; // deklaracija promenljive i
        i = 3; // dodela vrednosti promenljivoj i
```

```
        /* i = 5; ova cela linija se ne izvrsava */
           Console.WriteLine(i); // ispis vrednosti promenljive i
        }
    }
```

Kada se izvrši ovaj program ispisuje se vrednost `3` na ekranu.

1.10 Greške

Ko radi taj i greši – pre ili kasnije funkcija koju unesemo neće se ponašati onako kako želimo. Naravno, računar radi ono što mu kažemo, tako da krivca za učinjenu grešku treba tražiti u nama, a grešku treba pronaći i otkloniti. Postoje dve suštinski različite vrste grešaka – *sintaksne* i *semantičke*.

Kada program u sebi sadrži sintaksnu grešku nije ga moguće izvršiti, jer nije u skladu sa „pravilima pisanja“ odnosno gramatikom programskog jezika. Prostije rečeno, računar nas u tom slučaju ne razume. U najvećem broju slučajeva sintaksne greške su već i pre izvršavanja programa naznačene u editoru.

No, čak i kada program možemo izvršiti to nije garancija da u njemu nema greške – jer i dalje postoji mogućnost da nije urađeno ono što treba. Greške ovog tipa nazivaju se semantičke greške, a u slengu se zovu i bagovi (engl. *bug*). U slučaju da imamo ovakvu grešku, ostaje nam da je ponovnim pregledom programa u editoru uočimo i popravimo program.

Za razliku od sintaksnih grešaka koje i pre izvršavanja lako uočavamo, jedini način da konstatujemo semantičku grešku jeste da testiramo program na primerima i posmatramo njegov rad. U slučaju da uočimo nepravilnosti, vraćamo se u editor gde pokušavamo da ih otklonimo. Nakon toga ponovo testiramo program sve dok nismo zadovoljni njegovim radom. Nažalost, to što program radi dobro za neke test primere nije garancija da će uvek raditi dobro, i taj fenomen muči sve programere – od početnika do profesionalaca.

Pored grešaka u pisanju programa postoje i greške u izvršavanju (engl. *runtime error*), koje čine da program prestane sa radom tokom izvršavanja, iako je sintaksno u redu. Primer takve greške je kada se vrši učitavanje celobrojne vrednosti pomoću `int.Parse(Console.ReadLine())` a korisnik unese vrednost koja nije ceo broj (npr. slovo).

1.11 Neke osnovne matematičke funkcije

Standardne biblioteke jezika C#, u okviru klase `Math` obuhvataju i matematičke funkcije. Kompletan spisak metoda iz ove klase se mogu naći u [6]. Neke od njih, zajedno sa opisom njihovog rada, navodimo u tabeli 2.

C#	Opis	Vraćena vrednost
`Abs(x)`	apsolutna vrednost celog ili realnog broja x	$\|x\|$
`Sin(x)`	sinus ugla zadatog realnim brojem x	$\sin x$
`Cos(x)`	kosinus ugla zadatog realnim brojem x	$\cos x$
`Sqrt(x)`	kvadratni koren realnog broja x	$\sqrt{x}$
`Exp(x)`	stepen broja e na x	e^x
`Log(x)`	prirodni logaritam od x	$\ln x$
`Log10(x)`	logaritam za osnovu 10 od x	$\log_{10} x$

`Log(x,a)`	logaritam za osnovu a od x	$\log_a x$
`Max(x,y)`	maksimum dva broja x i y	$\max\{x, y\}$
`Min(x,y)`	minimum dva broja x i y	$\min\{x, y\}$
`Pow(a,x)`	stepen broja a na x, a i x su realni brojevi	a^x

Tabela 2. Neke matematičke funkcije u C#

U klasi `Math` se takođe nalaze i konstante, videti tabelu 3.

C#	Konstanta
E	e
PI	π

Tabela 3. Matematičke konstante

Sledeći program ilustruje upotrebu funkcija i konstanti.

```
class Program
{
    static void Main(string[] args)
    {
        double f1 = Math.Abs(-2) + Math.Abs(3.5);
        double f2 = Math.Log(Math.E) + Math.Log(27, 3);
        double f3 = Math.Max(Math.Exp(2), Math.Exp(-2));
        double f4 = Math.Pow(Math.Sin(Math.PI / 2), 2) +
                    Math.Pow(Math.Cos(Math.PI / 2), 2);
        double f5 = Math.Min(Math.Log10(100), Math.E);
        Console.WriteLine("{0,7:N4} {1,7:N4} {2,7:N4} {3,7:N4} {4,7:N4}",
                          f1, f2, f3, f4, f5);
    }
}
```

Prilikom izvršavanja programa, dobijamo sledeći ispis:

```
 5.5000  4.0000  7.3891  1.0000  2.0000
```

Za zaokruživanje realnih brojeva na raspolaganju su nam četiri funkcije – `Round`, `Ceiling`, `Floor` i `Truncate`, takođe u klasi `Math`, čiji opisi su dati u tabeli 4.

C#	Opis
`Ceiling(x)`	zaokruživanje realnog broja x na gore
`Floor(x)`	zaokruživanje realnog broja x na dole
`Round(x)`	zaokruživanje realnog broja x na najbliži ceo broj
`Truncate(x)`	odsecanje razlomljenog dela realnog broja x (iza decimalnog zareza)

Tabela 4. Funkcije za zaokruživanje

Treba imati u vidu da sve navedene funkcije vraćaju vrednost tipa `double` iako su, matematički gledano, vraćene vrednosti uvek celi brojevi, te moramo eksplicitno konvertovati tip ako želimo da koristimo takvu vrednost kao vrednost tipa `int`.

```
class Program
{
    static void Main(string[] args)
    {
        int i, j, k, l, m;
        i = (int)14.0;
        j = (int)(Math.Floor(3.44) + Math.Floor(-3.44));
        k = (int)Math.Ceiling(8.1);
        l = (int)(Math.Round(8.1) + Math.Round(8.9));
        m = (int)(Math.Truncate(3.3) + Math.Truncate(-3.3));
        Console.WriteLine("{0} {1} {2} {3} {4}", i, j, k, l, m);
    }
}
```

Izvršavanjem programa dolazi do sledećeg ispisa.

```
14 -1 9 17 0
```

U tabeli 5 date su vrednosti koje ove četiri funkcije uzimaju za nekoliko realnih brojeva.

	5.3	5.5	5.7	6	6.5	-5.3	-5.5	-5.7	-6	-6.5
`Math.Round`	5	6	6	6	6	-5	-6	-6	-6	-6
`Math.Floor`	5	5	5	6	6	-6	-6	-6	-6	-7
`Math.Ceiling`	6	6	6	6	7	-5	-5	-5	-6	-6
`Math.Truncate`	5	5	5	6	6	-5	-5	-5	-6	-6

Tabela 5. Primeri zaokruživanja koristeći četiri funkcije za zaokruživanje

1.12 Relacije i logički operatori, logički izrazi

Binarne relacije `>`, `<`, `>=`, `<=`, `!=`, `==`, služe za proveru da li je brojna vrednost navedena pre relacije, redom, veća, manja, veća ili jednaka, manja ili jednaka, različita, odnosno jednaka, brojnoj vrednosti navedenoj posle relacije. Vraćena vrednost svake od navedenih relacija je logičkog tipa, i može biti `true` ili `false`.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(8 > 4);
        Console.WriteLine(3 <= 2);
        Console.WriteLine(1 != 2);
        int a = 1;
        Console.WriteLine(a + 2 == Math.Exp(3));
    }
}
```

Izvršavanjem programa dolazi do sledećeg ispisa.

```
True
False
True
False
```

Od logičkih operacija izdvajamo u tabeli 6 binarne operacije za konjunkciju i disjunkciju, kao i unarnu operaciju negacije, koje sve operišu nad vrednostima logičkog tipa.

Operacija	Operator	Objašnjenje
konjunkcija	&&	Rezultat je tačan ako i samo ako su oba izraza tačna.
disjunkcija	||	Rezultat je tačan ako i samo ako je bar jedan od izraza tačan.
negacija	!	Rezultat je tačan ako i samo ako je izraz netačan.

Tabela 6. Logičke operacije

```
class Program
{
    static void Main(string[] args)
    {
        int a = 1, b = 3;
        Console.WriteLine(a > 5 && a < 7);
        Console.WriteLine(b < 5 || b > 7);
        Console.WriteLine(!(a >= 12));
        Console.WriteLine((a > 5 && b < 3) || !(a != 3));
    }
}
```

Ispis koji dobijamo izvršavanjem programa sledi.

```
False
True
True
False
```

Kao što smo već pomenuli, moguće je deklarisati i promenljive logičkog tipa, `bool`. Primer korišćenja ovakvih promenljivih ovog tipa sledi, i kada se ovaj program izvrši ispisuje se „True“.

```
class Program
{
    static void Main(string[] args)
    {
        int a = 13, b = 3;
        bool l1, l2;
        l1 = true;
        l2 = (a % b) == 0;
        l2 = l1 && !(l2);
        Console.WriteLine(l2);
    }
}
```

2 Kontrolne strukture

Često nam nije dovoljno da se sve naredbe izvršavaju jedna za drugom, već postoji potreba da se neka naredba (ili naredbeni blok) izvrši samo ukoliko su ispunjeni uslovi za to, ili želimo da ponavljamo više puta neki naredbeni blok. Da bismo to omogućili koristimo kontrolne strukture u programu.

2.1 IF grananje

Kada je potrebno da uslovimo izvršavanje jedne ili više naredbi, koristimo neko od uslovnih grananja. Jedno od njih je IF grananje, koje u najprostijem obliku ima sledeću sintaksu:

```
if (<logicki_izraz>)
{
    <niz_naredbi>
}
```

Pritom, `<niz_naredbi>` se izvršava samo ako je navedeni logički izraz tačan. U suprotnom, preskače se `<niz_naredbi>` i nastavlja se sa izvršavanjem onoga što se nalazi ispod IF grananja. Naredni program ilustruje primenu IF grananja.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite ceo broj: ");
        int a = int.Parse(Console.ReadLine());
        if (a > 0)
        {
            Console.WriteLine("Uneli ste pozitivan broj!");
            Console.WriteLine("Bas lepo... :)");
        }
        Console.WriteLine("a = {0}", a);
    }
}
```

Ispis nakon izvršavanja ovog programa razlikuje se ako je unet pozitivan broj:

```
Unesite ceo broj: 15
Uneli ste pozitivan broj!
Bas lepo... :)
a = 15
```

i ako nije unet pozitivan broj:

```
Unesite ceo broj: -2
a = -2
```

Ukoliko je potrebno da se izvrši jedna ili više naredbi i u slučaju da logički izraz nije tačan, koristimo varijantu IF grananja koja sadrži i tzv. ELSE blok. Tada je sintaksa sledeća.

```
if (<logicki_izraz>)
{
    <niz_naredbi_1>
}
else
{
    <niz_naredbi_2>
}
```

U slučaju da je navedeni logički izraz tačan, izvršava se `<niz_naredbi_1>` (IF blok), a preskače se `<niz_naredbi_2>`. Ako logički izraz nije tačan, preskače se `<niz_naredbi_1>`, a izvršava se `<niz_naredbi_2>` (ELSE blok). Dakle, kada se koristi ovakav oblik IF grananja, uvek će se izvršiti tačno jedan od dva navedena bloka.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write ("Unesite ceo broj: ");
        int a = int.Parse(Console.ReadLine());
        if (a > 0)
        {
            Console.WriteLine("Uneli ste pozitivan broj!");
        }
        else
        {
            Console.WriteLine("Uneli ste negativan broj...");
        }
    }
}
```

Izvršavanjem prethodnog programa možemo dobiti sledeći ispis.

```
Unesite ceo broj: -78
Uneli ste negativan broj...
```

Naredni program računa i ispisuje apsolutnu vrednost unetog celog broja.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite ceo broj: ");
        int a = int.Parse(Console.ReadLine()), b;
        if (a > 0)
        {
            b = a;
        }
```

```
        else
        {
            b = -a;
        }
        Console.WriteLine("Apsolutna vrednost {0} je {1}.", a, b);
    }
}
```

Navođenje novog IF grananja u okviru ELSE bloka daje nam mogućnost da iterativno postavimo više logičkih testova, tzv. ELSE-IF struktura. Samim tim, od svih navedenih dodela vrednosti promenljivoj `b` u narednom primeru, uvek će biti izvršena tačno jedna. Pritom, do svakog IF grananja doći ćemo samo ako su svi logički izrazi u prethodnim IF grananjima bili netačni.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite ceo broj: ");
        int a = int.Parse(Console.ReadLine());
        if (a > 10)
        {
            Console.WriteLine("a je vece od 10");
        }
        else if (a > 4)
        {
            Console.WriteLine("a je vece od 4, ali nije vece od 10");
        }
        else if (a > 0)
        {
            Console.WriteLine("a je vece od 0, ali nije vece od 4");
        }
        else
        {
            Console.WriteLine("a je manje ili jednako 0");
        }
    }
}
```

U nastavku sledi četiri primera ispisa ovog programa prilikom izvršavanja.

```
Unesite ceo broj a: 15
a je vece od 10

Unesite ceo broj a: 8
a je vece od 4, ali nije vece od 10

Unesite ceo broj a: 3
a je vece od 0, ali nije vece od 4

Unesite ceo broj a: -4
a je manje ili jednako 0
```

2.2 SWITCH-CASE grananje

IF grananja sa komplikovanim uslovima mogu biti prilično dugačka kao i teška za čitanje. U slučaju da imamo promenljivu rednog tipa (kao što su npr. `int`, `char`, `bool`), možemo koristiti SWITCH-CASE grananje. SWITCH-CASE naredba nam omogućava da u zavisnosti od vrednosti navedene promenljive bude izvršen određeni niz naredbi. Sintaksa ove naredbe je sledeća:

```
switch (<promenljiva>)
{
    case <vrednost1>:
        <niz_naredbi_1>
        break;
    case <vrednost2>:
        <niz_naredbi_2>
        break;
    ...
    default:
        <niz_naredbi_d>
        break;
}
```

Podsetimo se da u IF grananju ispunjenje nekog uslova znači izvršavanje svih naredbi u tom bloku, kao i da se svi kasniji uslovi (ukoliko postoje) preskaču i nastavlja se sa izvršavanjem toka programa nakon IF grananja. U SWITCH-CASE grananju, to nije slučaj, ukoliko se u svaki `case` blok ne stavi naredba `break`. Ova naredba govori računaru da ukoliko naiđe na nju, prekine sve naredne provere u `case` blokovima koji slede i nastavi izvršavanje naredbi koje slede nakon SWITCH-CASE naredbe.

Ukoliko promenljiva nema nijednu od navedenih vrednosti izvršava se niz naredbi iz `default` bloka, s tim da je `default` blok moguće i izostaviti.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: ");
        char c = char.Parse(Console.ReadLine());
        switch (c)
        {
            case 'p':
                Console.WriteLine("+ + +");
                break;
            case 'm':
                Console.WriteLine("- - -");
                break;
            case 'z':
                Console.WriteLine("* * *");
                break;
            default:
                Console.WriteLine("o o o");
```

```
                break;
            }
        }
    }
```

U nastavku sledi četiri primera ispisa ovog programa prilikom izvršavanja.

```
Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: p
+ + +

Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: m
- - -

Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: z
* * *

Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: w
o o o
```

Ukoliko bismo izostavili `default` blok iz prethodnog programa, nikakve naredbe se ne bi izvršavale kada bi bio unet znak različit od 'p', 'm' ili 'z'.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: ");
        char c = char.Parse(Console.ReadLine());
        switch (c)
        {
            case 'p':
                Console.WriteLine("+ + +");
                break;
            case 'm':
                Console.WriteLine("- - -");
                break;
            case 'z':
                Console.WriteLine("* * *");
                break;
        }
    }
}
```

Slede dva primera izvršavanja programa.

```
Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: m
- - -

Unesite 'p' za plus, 'm' za minus, a 'z' za zvezdicu: d
```

SWITCH-CASE naredba može biti korisna ukoliko želimo da se isti niz naredbi izvršava za neke vrednosti promenljive koje nisu uzastopne, a ne želimo da pravimo komplikovane IF uslove. U tom

slučaju nabrajamo sve vrednosti (sem poslednje) za koje želimo da se niz naredbi izvrši kao prazne case blokove (bez naredbi), dok uz poslednju pišemo niz naredbi koji želimo da se izvrši. Sledeći program ilustruje primenu SWITCH-CASE naredbe u slučaju da želimo da na osnovu unetog rednog broja meseca ispišemo broj dana u tom mesecu.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Unesite redni broj meseca m (ceo broj izmedju 1 i 12): ");
        int m = int.Parse(Console.ReadLine());
        switch (m)
        {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                Console.WriteLine("31 dan");
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                Console.WriteLine("30 dana");
                break;
            case 2:
                Console.WriteLine("Februar: Ako je godina prestupna -
                                   29 dana, inace 28 dana");
                break;
            default:
                Console.WriteLine("Uneli ste nepostojeci mesec.");
                break;
        }
    }
}
```

U nastavku su 4 primera izvršavanja programa:

```
Unesite redni broj meseca m (ceo broj izmedju 1 i 12): 3
31 dan

Unesite redni broj meseca m (ceo broj izmedju 1 i 12): 11
30 dana

Unesite redni broj meseca m (ceo broj izmedju 1 i 12): 2
Februar: Ako je godina prestupna - 29 dana, inace 28 dana

Unesite redni broj meseca m (ceo broj izmedju 1 i 12): 44
Uneli ste nepostojeci mesec.
```

2.3 FOR petlja

Koncept petlje u programiranju omogućava nam da neki blok naredbi izvršimo više puta. FOR petlja ima sledeću sintaksu:

```
for (<inicijalizacija>; <uslov>; <operacija>)
{
   <niz_naredbi>
}
```

Inicijalizacija se izvršava samo jednom, na početku izvršavanja petlje. Uslov se proverava iznova pre svakog prolaska kroz petlju, i samo ako je uslov zadovoljen <niz_naredbi> unutar FOR petlje, koji zovemo *telo petlje*, se izvršava. Ako prilikom te provere uslov nije zadovoljen prestaje se sa izvršavanjem tela petlje. Konačno, navedena operacija se izvršava posle svakog prolaska kroz petlju, a pre provere uslova.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 4; i++)
        {
            Console.Write("Trenutna vrednost i je: ");
            Console.WriteLine(i);
        }
    }
}
```

Prilikom izvršavanja ovog programa, dobijamo sledeći ispis.

```
Trenutna vrednost i je: 1
Trenutna vrednost i je: 2
Trenutna vrednost i je: 3
Trenutna vrednost i je: 4
```

Treba imati u vidu da promenljiva koja se deklariše prilikom inicijalizacije FOR petlje „postoji“ samo unutar petlje – čim se izađe iz petlje, ta promenljiva više nije deklarisana i samim tim nema ni vrednost.

Naredni program ispisuje kvadrate prvih deset prirodnih brojeva, od najvećeg ka najmanjem.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 10; i >= 1; i--)
        {
            Console.WriteLine("{0} na kvadrat je {1}.", i, i*i);
        }
    }
}
```

2.4 Ugnježdene FOR petlje

Ukoliko imamo dve FOR petlje tako da se jedna („unutrašnja“) nalazi unutar druge („spoljašnje“), u pitanju su *ugnježdene* petlje. Ako pretpostavimo da se blok naredbi u spoljašnjoj petlji izvršava n puta, samim tim će i cela unutrašnja petlja biti iznova izvršavana n puta.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 3; i++)
        {
            for (int j = 1; j <= 4; j++)
            {
                Console.WriteLine("i = {0}, j = {1}", i, j);
            }
        }
    }
}
```

Kad se izvrši prethodni program, dobijamo sledeći ispis.

```
i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 1, j = 4
i = 2, j = 1
i = 2, j = 2
i = 2, j = 3
i = 2, j = 4
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
i = 3, j = 4
```

U sledećem programu, koji ispisuje blok znakova „X“ visine 11 i širine 16, naredba `Console.Write("X")` će se izvršiti $11 \cdot 16 = 176$ puta jer se nalazi u unutrašnjoj petlji. S druge strane, naredba `Console.WriteLine()` nalazi se u spoljašnjoj petlji, ne i u unutrašnjoj, te se ona izvršava samo 11 puta – po jednom za svaki prolazak kroz spoljašnju petlju.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 11; i++)
        {
            Console.Write("  ");
            for (int j = 1; j <= 16; j++)
            {
                Console.Write("X");
            }
```

```
            Console.WriteLine();
        }
    }
}
```

Izvršavanjem dobijamo sledeći ispis.

```
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXX
```

Naredni program ispisuje kružiće (slovo „o“) u 13 redova promenljive dužine, gde dužina i-tog reda odgovara vrednosti funkcije $(i-7)^2$. Na kraju svakog reda ispisuje se „x“.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 13; i++)
        {
            for (int j = 1; j <= Math.Pow(i - 7, 2); j++)
            {
                Console.Write("o");
            }
            Console.WriteLine("x");
        }
    }
}
```

Ispis koji dobijamo izvršavanjem sledi.

```
oooooooooooooooooooooooooooooooooooooox
oooooooooooooooooooooooox
ooooooooooooooox
ooooooooox
oooox
ox
x
ox
oooox
ooooooooox
ooooooooooooooox
oooooooooooooooooooooooox
ooooooooooooooooooooooooooooooooooox
```

2.5 Kombinovanje FOR petlje i IF grananja

Kao što smo već videli, pomoću FOR petlje lako možemo da promenljivoj redom dodelimo vrednosti iz nekog intervala, sa konstantnim pozitivnim ili negativnim korakom. Ali, ako niz vrednosti koje želimo da dodelimo nema takvu strukturu, problem nije tako lako pravolinijski rešiti korišćenjem FOR petlje. U takvim slučajevima često nam pomaže IF grananje unutar FOR petlje.

Na primer, sledeći program ispisuje sve prirodne brojeve manje ili jednake od 50 za koje važi da nisu deljivi sa 3, a deljivi su sa 4.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 50; i++)
        {
            if (i % 3 != 0 && i % 4 == 0)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

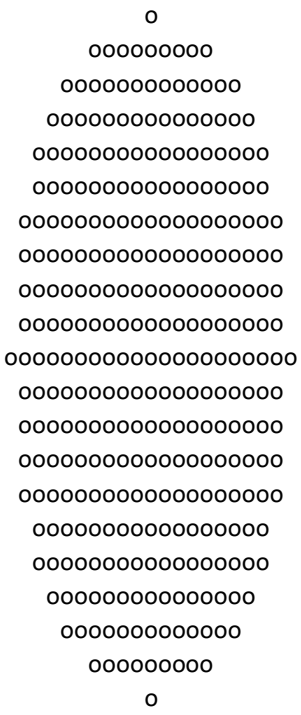
Posmatrajući FOR petlju i IF grananje iz prethodnog programa kao jednu celinu, zapravo smo dobili *„prolazak kroz one brojeve između 1 i 50 za koje je zadovoljen uslov da nisu deljivi sa 3, a jesu sa 4“*, i tačno za te brojeve izvršavaju se naredbe unutar IF-a, u ovom slučaju ispis.

U narednom programu, unutar bloka dimenzija 21×21 biće ispisani kružići koji formiraju krug poluprečnika 10. Ugnježdenim petljama prolazimo kroz sve parove brojeva i i j između 0 i 20, te za svaki par proveravamo da li je zadovoljena jednačina kruga $(i-10)^2 + (j-10)^2 \leq 10^2$. Za one parove za koje je jednačina zadovoljena ispisuje se kružić, dok se u ostalim slučajevima ispisuje razmak.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i <= 20; i++)
        {
            for (int j = 0; j <= 20; j++)
            {
                if (Math.Pow(i - 10, 2) + Math.Pow(j - 10, 2) <= 100)
                {
                    Console.Write("o");
                }
                else
                {
                    Console.Write(" ");
                }
            }
            Console.WriteLine();
```

```
            }
        }
    }
```

Izvršavanjem dobijamo sledeći ispis.



Naredni program ispisuje sve Pitagorine trojke brojeva (tj. trojke brojeva za koje važi $i^2 + j^2 = f^2$) manjih od 20. S obzirom da ne želimo da ispišemo iste trojke više puta, uvodimo dodatni uslov $i \leq j$. Pomoću ugnježdenih petlji proći ćemo kroz sve parove brojeva i i j između 1 i 19, a pobrinućemo se da uslov $i \leq j$ bude zadovoljen tako što ćemo u unutrašnjoj petlji i staviti kao inicijalnu vrednost za j. U slučaju da je $i^2 + j^2$ kvadrat celog broja koji je manji od 20, ispisujemo Pitagorinu trojku.

```
class Program
{
    static void Main(string[] args)
    {
        int d = 20;
        double f;
        for (int i = 1; i < d; i++)
        {
            for (int j = i; j < d; j++)
            {
                f = Math.Sqrt(i * i + j * j);
                if (Math.Round(f) == f && f < d)
                {
                    Console.WriteLine("{0} {1} {2}", i, j, f);
                }
            }
        }
```

```
        }
    }
```

Pomoću `Math.Round(f) == f` proveravamo da li je vrednost promenljive `f` ceo broj. Primetimo da smo pri ispisu mogli i da konvertujemo promenljivu `f` u ceo broj pomoću `(int)f`, ali kada je vrednost promenljive tipa `double` cela svakako se ispisuje bez decimala, te to nije neophodno.

Ispis koji dobijamo izvršavanjem programa sledi.

```
3  4  5
5 12 13
6  8 10
8 15 17
9 12 15
```

2.6 WHILE petlja i DO-WHILE petlja

WHILE i DO-WHILE petlje koristimo kada želimo da ponavljamo jednu ili više naradbi sve dok je dati logički uslov ispunjen. Pritom, broj ponavljanja ne mora biti unapred poznat.

Sintaksa WHILE petlje sledi.

```
while (<logicki_izraz>)
{
   <niz_naredbi>
}
```

Kada WHILE petlja počne da se izvršava, proverava se vrednost koju ima `<logicki_izraz>`. Ako je izraz netačan momentalno se izlazi iz petlje. U slučaju da je izraz tačan izvršava se `<niz_naredbi>`, koji zovemo telo petlje, a nakon toga dolazi do povratka na početak petlje gde se ponavlja isti proces – iznova se proverava vrednost logičkog izraza.

U narednom programu, ispisuju se redom svi stepeni broja 3 koji su manji od datog n. (Broj je stepen broja a ako se može napisati kao a^t za neki prirodan broj t.)

```
class Program
{
    static void Main(string[] args)
    {
        int a = 1, n = 7899;
        while (a < n)
        {
            Console.WriteLine(a);
            a = a * 3;
        }
    }
}
```

Izvršavanjem programa, dobija se sledeći ispis.

```
1
3
```

```
9
27
81
243
729
2187
6561
```

WHILE petlju možemo koristiti i slično petlji FOR, sa promenljivom koja redom uzima vrednosti dok je zadovoljen navedeni uslov. Pritom, inicijalizaciju promenljive vršimo pre petlje, a ažuriranje vrednosti unutar petlje.

Sledeći program ispisuje najmanji prirodan broj i za koji važi da je 9 poslednja cifra polinoma $i^2 + 4i + 3$. Kada se program izvrši, ispisuje se broj 8.

```
class Program
{
    static void Main(string[] args)
    {
        int i = 1;
        while ((i * i + 4 * i + 3) % 10 != 9)
        {
            i++;
        }
        Console.WriteLine(i);
    }
}
```

Kao što smo videli, u WHILE petlji logički uslov se nalazi na početku, te na njega nailazimo pre prvog izvršavanja naredbi koje su navedene unutar petlje. Petlja DO-WHILE je struktura slična WHILE petlji, s tom razlikom što se logički uslov nalazi na kraju, a ne na početku petlje. Sintaksa petlje DO-WHILE je sledeća.

```
do
{
   <niz_naredbi>
}
while (<logicki_izraz>);
```

Kada se DO-WHILE petlja izvršava, najpre se izvrši `<niz_naredbi>`, odnosno telo petlje. Nakon toga se proverava vrednost koju ima `<logicki_izraz>`. Ako je on tačan vraćamo se na početak petlje i ponovo izvršavamo `<niz_naredbi>`, a ako nije tačan izlazimo iz petlje.

Ako `<logicki_izraz>` u uslovu petlje sadrži promenljive onda one moraju biti deklarisane pre petlje.

Treba imati u vidu da se naredbe unutar WHILE petlje u nekim slučajevima neće nijednom izvršiti, dok se naredbe unutar DO-WHILE petlje uvek izvršavaju bar jednom.

Naredni program traži od korisnika da unosi brojeve, sve dok uneti broj ne bude deljiv sa 29.

```
class Program
{
    static void Main(string[] args)
    {
        int a;
        do
        {
            Console.Write("Za zavrsetak, unesite broj deljiv sa 29: ");
            a = int.Parse(Console.ReadLine());
        }
        while (a % 29 != 0);
    }
}
```

U nastavku dajemo jedan mogući ispis prilikom izvršavanja ovog programa.

```
Za zavrsetak, unesite broj deljiv sa 29: 17
Za zavrsetak, unesite broj deljiv sa 29: 789
Za zavrsetak, unesite broj deljiv sa 29: 21
Za zavrsetak, unesite broj deljiv sa 29: 4556
Za zavrsetak, unesite broj deljiv sa 29: 58
```

2.7 Kombinovanje WHILE (ili DO-WHILE) petlje i IF grananja

I WHILE petlja (a isto tako i DO-WHILE petlja) može da se kombinuje sa IF grananjem na sličan način kao i FOR petlja. Tako postižemo da se deo programa u bloku IF grananja izvrši samo onda kada je zadovoljen uslov naveden u IF grananju.

Sledeći program ispisuje vrednosti logaritma (sa prirodnom osnovom) prirodnih brojeva (redom, počevši od broja dva) sve dok ta vrednost ne postane veća od datog broja n, ali samo one vrednosti čija je prva cifra posle decimalnog zareza jednaka nuli.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 5;
        int i = 2;
        do
        {
            if (Math.Log(i) - Math.Truncate(Math.Log(i)) < 0.1)
            {
                Console.WriteLine("ln({0}) = {1:N3}", i, Math.Log(i));
            }
            i++;
        }
        while (Math.Log(i) <= n);
    }
}
```

Generalno govoreći, ako je `x` promenljiva tipa `double` čija je vrednost pozitivna, logičkim izrazom `(x - Math.Truncate(x) < 0.1)` proveravamo da li `x` ima nulu iza decimalnog zareza. Važno je napomenuti i da inkrementacija vrednosti promenljive `i` (unutar tela petlje) mora biti izvan IF grananja, jer se inkrementacija vrši prilikom svakog prolaza kroz petlju, nezavisno od toga da li u tom prolazu dolazi do ispisa ili ne.

Kada se program izvrši, dobijamo sledeći ispis.

```
ln(3) = 1.099
ln(8) = 2.079
ln(21) = 3.045
ln(22) = 3.091
ln(55) = 4.007
ln(56) = 4.025
ln(57) = 4.043
ln(58) = 4.060
ln(59) = 4.078
ln(60) = 4.094
```

2.8 Koju petlju koristiti?

FOR petlju najčešće koristimo da nekoj promenljivoj redom dodeljujemo vrednosti iz unapred poznatog opsega brojeva koji su uzastopni (ili ravnomerno raspoređeni). Samim tim, ova petlja je pogodna za rešavanje problema u kojima je potrebno izvršiti repetitivnu radnju nad unapred poznatim opsegom ili strukturom. S druge strane, WHILE petlja se izvršava sve dok je navedeni logički uslov zadovoljen, te je posebno korisna za obavljanje procesa čije trajanje i obim nisu unapred poznati, ali nam je poznat uslov za završetak procesa. DO-WHILE petlja je po prirodi slična WHILE petlji, a dobija prednost u situacijama u kojima pitanje postavljeno u logičkom uslovu ne želimo da pitamo pre prvog izvršavanja tela petlje.

Treba obratiti pažnju da u telu petlje i kod WHILE i kod DO-WHILE petlje mora postojati naredba ili više njih koje utiču na promenu vrednosti logičkog izraza. U suprotnom, logički izraz će stalno biti tačan i telo petlje će se izvršavati zauvek. To se zove beskonačna petlja.

Iako svaka od petlji sa kojima smo se upoznali ima svoje osobenosti koje je u određenim situacijama čine pogodnijom za korišćenje od ostalih, treba imati u vidu i da najčešće ne postoji samo jedna petlja koja odgovara datom problemu (već je za rešavanje moguće iskoristiti svaku od navedenih petlji, uz manje ili više truda).

Naredna tri programa rade potpuno isto – ispisuju sve parne prirodne brojeve koji nisu veći od 20, u opadajućem redosledu.

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 20; i > 0; i -= 2)
        {
            Console.WriteLine(i);
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        int i = 20;
        while (i > 0)
        {
            Console.WriteLine(i);
            i -= 2;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        int i = 20;
        do
        {
            Console.WriteLine(i);
            i -= 2;
        }
        while (i > 0);
    }
}
```

2.9 Problem „uslova u sredini petlje“

Pretpostavimo da želimo da napišemo program koji rešava sledeći problem – za dati prirodan broj n ispisuje se vrednost funkcije $f(x) = x^2 + x + 1$ redom za sve prirodne brojeve x za koje važi da je $f(x) \leq n$.

Proces kojim pravolinijski dolazimo do rešenja može se opisno dati na sledeći način.

Na početku promenljivoj x dodelimo 1, a zatim ponavljamo sledeći blok akcija:

1. *izračunamo f za trenutnu vrednost promenljive x...*
2. *...i u slučaju da je f još uvek manje ili jednako n...*
3. *...ispišemo f...*
4. *...i uvećamo vrednost promenljive x za jedan.*

Problem nastaje kada probamo da napišemo program koji prati ovaj niz radnji, jer se uslov (2.) ne nalazi ni na početku ni na kraju bloka akcija koji želimo da ponavljamo, pa direktno nije moguće koristiti ni WHILE petlju ni DO-WHILE petlju. Stoga ćemo modifikovati naš opis rešenja, prebacićemo *prvo* izračunavanje funkcije među inicijalne akcije i time ćemo dobiti uslov na početku bloka akcija koje ponavljamo. Ovo možemo da uradimo jer znamo da je $f(1) = 3$.

Na početku promenljivoj x dodelimo 1, promenljivoj f dodelimo 3, a zatim ponavljamo sledeći blok akcija:

1. *U slučaju da je f još uvek manje ili jednako n...*
2. *...ispišemo f...*
3. *...uvećamo vrednost promenljive x za jedan...*
4. *...i izračunamo f za trenutnu vrednost promenljive x.*

Pošto je sada uslov na početku, možemo iskoristiti WHILE petlju. Program koji rešava navedeni problem sledi.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 80;
        int x = 1, f = 3;
        while (f <= n)
        {
            Console.WriteLine(f);
            x++;
            f = x * x + x + 1;
        }
    }
}
```

Izvršavanjem ovog programa dobijamo sledeći ispis.

```
3
7
13
21
31
43
57
73
```

Na kraju vredi napomenuti i da je ovo samo jedna od mogućnosti za prevazilaženje navedenog problema. Generalno govoreći, što je problem koji posmatramo kompleksniji, veća je i paleta pristupa koje možemo iskoristiti za njegovo rešavanje.

2.10 Brojanje, sabiranje, množenje

U programima ćemo često želeti da prebrojimo neke objekte, da saberemo neke brojeve, ili pak da ih pomnožimo. Ta tri problema su srodna, i rešavamo ih na sličan način.

Za brojanje ćemo koristiti promenljivu kojoj na početku dodeljujemo vrednost 0, a nakon toga svaki put kada naiđemo na neki od objekata (koje brojimo) uvećavamo tu promenljivu za jedan.

Naredni program za dati prirodan broj n ispisuje broj prirodnih brojeva između 1 i n koji su deljivi sa 7, a nisu deljivi sa 3.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 425;
        int brojac = 0;
        for (int i = 1; i <= n; i++)
        {
            if (i % 7 == 0 && i % 3 != 0)
            {
                brojac++;
            }
        }
        Console.WriteLine(brojac);
    }
}
```

U slučaju da želimo da saberemo više brojeva, koristimo promenljivu kojoj na početku dodeljujemo vrednost 0, a nakon toga svaki od brojeva koje sabiramo dodajemo na postojeću vrednost te promenljive.

Sledi program koji za dat prirodan broj n ispisuje zbir svih stepena broja 2 koji su manji od n i poslednja cifra im je 4.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 2000;
        int s = 1, zbir = 0;
        while (s < n)
        {
            if (s % 10 == 4)
            {
                zbir += s;
            }
            s *= 2;
        }
        Console.WriteLine(zbir);
    }
}
```

Ako želimo da izračunamo proizvod više brojeva, koristimo promenljivu kojoj na početku dodelimo vrednost 1, a potom vrednost te promenljive ažuriramo množeći je svakim od pomenutih brojeva.

Naredni program ispisuje proizvod svih neparnih brojeva koji su manji od datog prirodnog broja n.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 15;
        int proizvod = 1;
        for (int i = 1; i < n; i += 2)
        {
            proizvod *= i;
        }
        Console.WriteLine(proizvod);
    }
}
```

3 Nizovi i matrice

Do sada smo za čuvanje podataka tokom izvršavanja programa koristili samo promenljive. U slučajevima kada je potrebno sačuvati veću količinu podataka potrebna nam je naprednija struktura u koju takve podatke možemo da smestimo, i da tim podacima kasnije možemo lako da pristupimo. Ovde ćemo se detaljnije upoznati sa dve takve strukture podataka, nizovima i matricama.

3.1 Nizovi

U programiranju, pod pojmom *niz* podrazumevamo konačan niz promenljivih istog tipa. Mi ćemo deklarisati i koristiti nizove na dva načina, zavisno od toga da li nam je dužina niza unapred poznata.

Sintaksa deklaracije niza čiju dužinu znamo u momentu deklaracije ima dve forme:

```
<tip>[] <ime> = new <tip>[<duzina_niza>];
<tip>[] <ime> = { <niz_vrednosti> };
```

Na primer, prva varijanta može izgledati ovako:

```
int[] niz = new int[8];
```

a druga ovako:

```
double[] pera = { 4.5, 4.4, 2.9, 1.11, 6.55, 8.543 };
```

U prvoj varijanti samo smo deklarisali niz, dok smo u drugoj varijanti odmah i upisali navedene vrednosti (redom) u niz. Dužina niza u prvoj varijanti je eksplicitno data, dok je u drugoj varijanti ona jednaka broju elemenata navedenih unutar vitičastih zagrada.

U slučaju da u momentu deklaracije ne znamo dužinu niza, deklarisaćemo niz dovoljno velike dužine u koji ćemo kasnije smeštati elemente niza. S obzirom na to da u ovom slučaju stvarna dužina niza ne odgovara deklarisanoj dužini, pored samog niza potrebna nam je i dodatna celobrojna promenljiva u kojoj ćemo čuvati stvarnu dužinu niza. Sintaksa deklaracije u ovom slučaju je sledeća:

```
<tip>[] <ime> = new <tip>[<gornje_ogranicenje_duzine_niza>];
int <dIme> = 0;
```

Na primer,

```
int[] niz = new int[100];
int dNiz = 0;
```

Pri deklaraciji treba voditi računa da deklarisana dužina niza bude dovoljno velika da niz može da primi sve elemente koje ćemo u njega smeštati tokom izvršavanja programa. Za ime promenljive `<dIme>` uvek ćemo koristiti sledeću konvenciju, dodaćemo slovo „d“ ispred imena niza, pri čemu samo ime niza navodimo velikim slovom. Tako npr, ako je ime niza „pera“, promenljiva u kojoj čuvamo njegovu stvarnu dužinu zvaće se „dPera“.

Deklarisanu dužinu niza `<ime>` dobijamo sa `<ime>.Length`. Treba imati u vidu da su elementi niza indeksirani počevši od nule, tj. brojevima od `0` do `<ime>.Length-1`. Samim tim, prvi element niza ima indeks 0, drugi element ima 1, itd. Da izbegnemo konfuziju, najčešće ćemo eksplicitno naglasiti da govorimo o „elementu niza sa određenim indeksom“. Pristup elementu niza sa indeksom i obavlja se pomoću `<ime>[i]`.

Ako želimo da „prođemo“ redom kroz *ceo* niz, to najčešće obavljamo FOR petljom. Naredni program u datom nizu najpre menja vrednost elementu sa indeksom 2, a zatim redom ispisuje sve elemente niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 4, 4, 7, 1, 6, 8 };
        niz[2] = 333;
        for (int i = 0; i < niz.Length; i++)
        {
            Console.WriteLine(niz[i]);
        }
    }
}
```

Sledi ispis koji dobijamo kad se program izvrši.

```
4
4
333
1
6
8
```

U sledećem programu korisnik unosi elemente sa tastature, gde ukupan broj elemenata koji će biti unet nije unapred poznat, ali pretpostavlja se da korisnik neće uneti više od 100 elemenata. Nakon toga, redom se ispisuju elementi niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = new int[100];
        int dNiz = 0;
        char c;

        do
        {
            Console.Write("Unesite clan niza: ");
            niz[dNiz] = int.Parse(Console.ReadLine());
            dNiz++;
            Console.Write("Da li zelite da zavrsite sa unosom (d/n)? ");
            c = char.Parse(Console.ReadLine());
        }
```

```
            while (c == 'n');

            for (int i = 0; i < dNiz; i++)
            {
                Console.WriteLine(niz[i]);
            }
        }
    }
```

Pošto u promenljivoj `dNiz` čuvamo stvarnu dužinu niza, čim upišemo novi element u niz odmah uvećavamo vrednost promenljive `dNiz` – tako smo sigurni da će njena vrednost uvek odgovarati broju elemenata u nizu. Treba primetiti i da se ispis vrši samo do elementa sa indeksom `dNiz-1` (što je poslednji *stvarni* element niza), iako niz ima 100 deklarisanih elemenata.

Jedna varijanta ispisa ovog programa sledi.

```
Unesite clan niza: 4
Da li zelite da zavrsite sa unosom (d/n)? n
Unesite clan niza: 78
Da li zelite da zavrsite sa unosom (d/n)? n
Unesite clan niza: -3
Da li zelite da zavrsite sa unosom (d/n)? n
Unesite clan niza: 99
Da li zelite da zavrsite sa unosom (d/n)? d
4
78
-3
99
```

U narednom programu, za dati niz ispisuju se najpre redom svi parni elementi niza, a nakon toga redom svi neparni elementi niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 1, 124, 21, 1, -7, 44, 1000, 3, 12, 111 };
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] % 2 == 0)
            {
                Console.Write("{0} ", niz[i]);
            }
        }
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] % 2 != 0)
            {
                Console.Write("{0} ", niz[i]);
            }
        }
```

```
            Console.WriteLine();
        }
    }
```

Ispis ovog programa sledi.

```
    2 124 44 1000 12 1 21 1 -7 3 111
```

U sledećem programu kreiramo novi niz u koji smeštamo elemente datog niza u obrnutom redosledu. Nakon toga redom ispisujemo elemente novog niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 6, 1, 4, 7, 9, 3, 5 };
        int[] novi = new int[niz.Length];
        for (int i = 0; i < niz.Length; i++ )
        {
            novi[novi.Length - i - 1] = niz[i];
        }

        for (int i = 0; i < novi.Length; i++)
        {
            Console.Write("{0} ", novi[i]);
        }
        Console.WriteLine();
    }
}
```

S obzirom da smo niz `novi` kreirali nakon deklaracije datog niza `niz`, u momentu deklaracije niza `novi` već je poznata dužina koju ovaj niz treba da ima – ona je ista kao i dužina niza `niz`, te je čitamo preko `niz.Length` i koristimo za deklaraciju niza `novi`.

Ispis ovog programa sledi.

```
    5 3 9 7 4 1 6
```

Naredni program za dati niz ispisuje redom sve njegove elemente do prvog negativnog elementa. U slučaju da u nizu nema negativnih elemenata, ispisuju se svi elementi niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 3, 1, 12, 22, 0, 8, 44, -2, 1, -8 };
        int i = 0;
        while (i < niz.Length && niz[i] >= 0)
        {
            Console.WriteLine(niz[i]);
            i++;
```

```
        }
    }
}
```

Ispis programa sledi.

```
3
1
12
22
0
8
44
```

U prethodnom programu, uslov WHILE petlje sadrži konjunkciju dva uslova – drugi uslov zaustavlja dalje izvršavanje petlje kada naiđemo na negativan broj, dok prvi zaustavlja izvršavanje kada stignemo do kraja niza (u slučaju da su svi elementi niza nenegativni). Pritom, zaustavljanje na kraju niza će se dogoditi kada vrednost promenljive `i` postane `niz.Length`. Primetimo da za tu vrednost promenljive `i` postavljanje drugog uslova u konjunkciji, `niz[i] >= 0`, nema smisla, jer `niz` nema element sa tim indeksom. Ipak, u tom slučaju neće doći do greške, jer se u programskom jeziku C# drugi logički izraz u konjunkciji uopšte ne posmatra ako je prvi logički izraz netačan (na osnovu te informacije je već jasno da je cela konjunkcija netačna). Naravno, ukoliko bismo logičkim izrazima u programu zamenili mesta došlo bi do greške (u slučaju kada dati niz nema negativnih elemenata).

3.2 Ekstremni elementi

Generalno gledano, u ovom delu bavimo se nalaženjem ekstremnih elementa, po nekom kriterijumu, u datom skupu vrednosti. Da bismo to uradili prolazimo kroz elemente skupa, jedan po jedan, sve vreme čuvajući vrednost ekstremnog elementa skupa elemenata kroz koje smo do tada prošli. Za svaki novi element koji posmatramo poredimo njegovu vrednost sa trenutno sačuvanom ekstremnom vrednošću, koju ažuriramo ako je to potrebno.
Za cele i realne brojeve standardni poredak dat je relacijom $\leq$, i u tom slučaju ekstremne elemente konačnog skupa nazivamo minimum i maksimum.
Naredni program nalazi i ispisuje minimalni element datog niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 9, 2, 12, -4, 1, -7, 4 };
        int m = niz[0];
        for (int i = 1; i < niz.Length; i++)
        {
            if (niz[i] < m)
            {
                m = niz[i];
            }
        }
```

```
            Console.WriteLine("Najmanji element datog niza je {0}.", m);
        }
    }
```

U predhodnom programu imali smo poseban tretman za element sa indeksom 0 – najpre smo njegovu vrednost prepisali u promenljivu `m`, da bismo nakon toga, u potrazi za minimumom, prošli kroz sve ostale elemente niza.

U naprednijoj varijanti ovog pristupa, u promenljivu `m` inicijalno možemo upisati neki „broj za koji znamo da je veći od svih elemenata niza“ i onda izvršiti isti proces redom za sve elemente niza, uključujući i element sa indeksom 0. U tom slučaju, u promenljivu `m` će već u prvom prolasku kroz petlju biti smeštena vrednost elementa sa indeksom 0 (jer je taj element sigurno manji od inicijalne vrednosti promenljive `m`), a ostatak procesa odvijaće se potpuno isto kao u prethodnom programu.

Ostaje nam da odaberemo pogodan „broj za koji znamo da je veći od svih elemenata niza“. Za tip `int`, najveći i najmanji broj koji se može smestiti u promenljivu tog tipa dobijaju se pomoću `int.MaxValue` i `int.MinValue`. Samim tim, `int.MaxValue` je sigurno veći ili jednak svim elementima svakog niza koji sadrži brojeve tipa `int`. Za tip `double` imamo `double.PositiveInfinity` i `double.NegativeInfinity`, i svi brojevi tipa `double` nalaze se između ove dve vrednosti.

Korišćenjem opisanog pristupa, na drugi način nalazimo minimum niza i ispisujemo ga.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 9, 2, 12, -4, 1, -7, 4 };
        int m = int.MaxValue;
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] < m)
            {
                m = niz[i];
            }
        }
        Console.WriteLine("Najmanji element datog niza je {0}.", m);
    }
}
```

Maksimum i minimum niza `<ime>` mogu se dobiti direktno pomoću `<ime>.Max()` i `<ime>.Min()`. Treba imati u vidu da ukoliko je stvarna dužina niza manja od deklarisane, ovi metode ne smemo koristiti jer mogu dati pogrešne vrednosti.

Naredni program nalazi i ispisuje najveći *neparan* element datog niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 6, 1, 4, 7, 9, 3, 5, 11, 8, 13, 42 };
        int m = int.MinValue;
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] % 2 == 1 && niz[i] > m)
```

```
            {
                m = niz[i];
            }
        }
        Console.WriteLine(m);
    }
}
```

Primetimo da rešavajući prethodni zadatak nismo mogli da na početku programa promenljivoj `m` jednostavno dodelimo vrednost prvog neparnog elementa, jer njegova pozicija zavisi od parnosti elemenata datog niza i nije unapred poznata. S druge strane, korišćenjem već pomenutog naprednijeg pristupa i `int.MinValue` lako i efikasno prevazilazimo ovaj problem.

U narednom programu za dati niz koji sadrži realne brojeve nalazimo i ispisujemo najveći element niza i element niza koji je drugi po veličini.

```
class Program
{
    static void Main(string[] args)
    {
        double[] niz = { 2.3, 9.1, 2.7, 12.4, -4.44, 1.6, -7.3, 4.0 };
        double max1 = double.NegativeInfinity, max2 = double.NegativeInfinity;
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] > max1)
            {
                max2 = max1;
                max1 = niz[i];
            }
            else if (niz[i] > max2)
            {
                max2 = niz[i];
            }
        }
        Console.WriteLine("Dva najveca elementa datog niza su {0} i {1}.",
                    max1, max2);
    }
}
```

3.3 Sortiranje niza

Sortiranje niza brojeva je jedan od najvažnijih problema u računarstvu. Za dati niz, želimo da uredimo njegove elemente u neopadajućem poretku. Na primer, kada tako uredimo niz `{ 5, 3, -1, 2, 7, 3 }` dobijamo `{ -1, 2, 3, 3, 5, 7 }`. Postoji mnogo različitih pristupa sortiranju, mi ćemo ovde navesti dva relativno jednostavna algoritma – tzv. *bubble-sort* („sortiranje isplivavanjem“) i *insertion-sort* („sortiranje umetanjem“). Postoje i drugi algoritmi sortiranja od kojih su neki opisani u [7; 8].

Kod bubble-sort algoritma vrši se veći broj uzastopnih poređenja susednih elemenata u nizu koji sortiramo. Konkretno, algoritam obavlja sortiranje niza sa n elemenata u $n - 1$ „prolaza“.

U prvom prolazu, najpre poredimo prva dva elementa niza. U slučaju da je prvi veći od drugog, menjamo im mesta. Nakon toga, poredimo drugi i treći element, menjajući im mesta ako je drugi veći

od trećeg. Isti proces ponavljamo i za treći i četvrti element, pa četvrti i peti, i redom za sve parove susednih elemenata do kraja niza, završavajući time prvi prolaz. Primetimo da se nakon prvog prolaza najveći element niza sigurno nalazi na poslednjoj poziciji – on je kroz ovaj proces „isplivao“ na poslednju poziciju (otud potiče ime algoritma).

U drugom prolazu ponavljamo isti proces, redom poredeći susedne elemente, s tim da završavamo prolaz jedan korak ranije (imajući u vidu da se na poslednjoj poziciji već nalazi najveći element niza). Slično kao u prethodnom slučaju, nakon drugog prolaza element niza koji je drugi po veličini „isplivao“ je na pretposlednju poziciju.

Dalje nastavljamo sa prolazima, skraćujući svaki sledeći prolaz za jedno poređenje. Nakon $n - 1$ prolaza, niz je sortiran.

Ovaj proces ilustrovaćemo primerom, sortirajući niz koji redom sadrži elemente 3, 9, 5, 6 i 1.

prvi prolaz:

3 9 5 6 1 →	**3 9** 5 6 1	(3 ≤ 9, pa ne dolazi do zamene)
3 **9 5** 6 1 →	3 **5 9** 6 1	(9 > 5, pa dolazi do zamene)
3 5 **9 6** 1 →	3 5 **6 9** 1	(9 > 6, pa dolazi do zamene)
3 5 6 **9 1** →	3 5 6 **1 9**	(9 > 1, pa dolazi do zamene)

drugi prolaz:

3 5 6 1 9 →	**3 5** 6 1 9
3 **5 6** 1 9 →	3 **5 6** 1 9
3 5 **6 1** 9 →	3 5 **1 6** 9

treći prolaz:

3 5 1 6 9 →	**3 5** 1 6 9
3 **5 1** 6 9 →	3 **1 5** 6 9

četvrti prolaz:

3 1 5 6 9 →	**1 3** 5 6 9

U nastavku sledi program koji sortira niz navedenim algoritmom, i nakon toga ga ispisuje.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 9, 2, 12, -4, 1, -7, 4, 77, 0 };
        for (int i = 0; i < niz.Length - 1; i++)
        {
            for (int j = 0; j < niz.Length - 1 - i; j++)
            {
                if (niz[j] > niz[j + 1])
                {
                    int pom = niz[j + 1];
                    niz[j + 1] = niz[j];
                    niz[j] = pom;
                }
            }
        }
```

```
            for (int i = 0; i < niz.Length; i++)
            {
                Console.WriteLine(niz[i]);
            }
        }
    }
```

Kada se program izvrši, dobijamo sledeći ispis.

```
-7
-4
0
1
2
2
4
9
12
77
```

Prelazimo na opis *insertion-sort* algoritma za sortiranje niza.

U datom nizu najpre izdvajamo element sa indeksom jedan. Zavisno od toga da li je izdvojeni element veći od elementa sa indeksom nula, umećemo ga pre ili posle elementa sa indeksom nula, tako da prva dva elementa niza budu sortirana (tj. poređana u neopadajućem redosledu). Nakon toga izdvajamo element sa indeksom dva, te ga umećemo u već sortirani početni dvoelementni segment na odgovarajuće mesto tako da sada prva tri elementa niza budu sortirana. Isti proces nastavljamo redom sa svim preostalim elementima niza.

U opštem slučaju, pretpostavimo da je prvih k elemenata niza (elementi sa indeksima $0, 1, 2, \dots, k-1$) već sortirano i da želimo da u sortirani deo umetnemo izdvojeni element sa indeksom k. Pomeramo za jedno mesto udesno elemente u sortiranom delu koji su veći od izdvojenog elementa, jedan po jedan – najpre element sa indeksom $k-1$ pomerimo na poziciju k, potom element sa indeksom $k-2$ pomerimo na poziciju $k-1$, itd, sve dok „upražnjeno mesto" ne bude na poziciji na koju želimo da umetnemo izdvojeni element.

U tabeli 7 dat je primer primene ovog algoritma na niz koji sadrži elemente 6, 3, 5, 1, 8 i 4. Sortiran početni deo niza označen je masnim slovima, a trenutno izdvojen element ima žutu pozadinu.

Kada pišemo program koji sortira niz ovim algoritmom, prilikom umetanja moramo voditi računa o tome da sve elemente sortiranog bloka koji se nalaze desno od pozicije umetanja pomerimo za jedno mesto udesno (redom zdesna na levo). Izdvojeni element pre umetanja moramo sačuvati u pomoćnoj promenljivoj, jer će već prilikom prvog prolaza kroz WHILE petlju biti prebrisan.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 9, 2, 12, -4, 1, -7, 4, 0, 77 };
        for (int i = 1; i < niz.Length; i++)
        {
            int pom = niz[i];
            int j = i;
```

```
                while (j > 0 && niz[j - 1] > pom)
                {
                    niz[j] = niz[j - 1];
                    j--;
                }
                niz[j] = pom;
            }

            for (int i = 0; i < niz.Length; i++)
            {
                Console.WriteLine(niz[i]);
            }
        }
    }
```

```
6 3 5 1 8 4

                3           3
6 3 5 1 8 4   6 _ 5 1 8 4   _ 6 5 1 8 4   3 6 5 1 8 4

                  5           5
3 6 5 1 8 4   3 6 _ 1 8 4   3 _ 6 1 8 4   3 5 6 1 8 4

                    1           1           1           1
3 5 6 1 8 4   3 5 6 _ 8 4   3 5 _ 6 8 4   3 _ 5 6 8 4   _ 3 5 6 8 4   1 3 5 6 8 4

                      8
1 3 5 6 8 4   1 3 5 6 _ 4   1 3 5 6 8 4

                        4           4           4           4
1 3 5 6 8 4   1 3 5 6 8 _   1 3 5 6 _ 8   1 3 5 _ 6 8   1 3 _ 5 6 8   1 3 4 5 6 8
```

Tabela 7. Primer sortiranja algoritmom insertion-sort

Sortiranje niza `<niz>` može se izvršiti i pomoću metoda `Array.Sort(<niz>)`. Treba voditi računa da ovaj metod sortira sve elemente niza, što nije ono što želimo ako stvarna dužina niza nije jednaka deklarisanoj.

3.4 Skupovi i multiskupovi

U matematici, skup predstavlja neuređenu kolekciju različitih elemenata, dok je multiskup neuređena kolekcija elemenata koji se mogu i ponavljati (više od jednom). Jedan očigledan način predstavljanja skupova i multiskupova je preko niza koji (u proizvoljnom redosledu, koji sami biramo) sadrži elemente skupa (odnosno multiskupa). Takvo predstavljanje je zgodno za skupove i multiskupove koji se ne menjaju ili im se samo dodaju elementi, no nije praktično kada želimo da uklanjamo elemente, jer je uklanjanje elementa iz sredine niza komplikovan i spor proces.

U slučaju da znamo da se svi elementi skupa (ili multiskupa) nalaze u nekom unapred utvrđenom skupu vrednosti (koji ćemo zvati *univerzum*), predstaljvanje se može uraditi i na drugi način, tako da se kasnije dodavanje i uklanjanje elemenata može uraditi brzo i elegantno. U tom slučaju skupove ćemo deklarisati kao niz tipa `bool` čija je dužina jednaka kardinalnosti univerzuma, a multiskupove kao niz tipa `int` čija je dužina jednaka kardinalnosti univerzuma.

Za skup predstavljen kao niz tipa `bool` čija je dužina jednaka kardinalnosti univerzuma interpretacija je sledeća – element niza ima vrednost `true` ako i samo ako se odgovarajući element univerzuma nalazi u skupu. Na primer, ako je univerzum $\{0, 1, 2, 3, 4\}$, a skup koji želimo da predstavimo je $\{1, 4\}$, predstavili bismo ga logičkim nizom `{ false, true, false, false, true }`.

U sledećem programu dat je univerzum (kao niz različitih celih brojeva), a zatim se generišu skupovi A i B koji sadrže, respektivno, sve elemente univerzuma koji su deljivi sa tri, i sve parne elemente univerzuma. Nakon toga se iz skupa A izbacuju oni elementi koji pripadaju skupu B (drugim rečima, nakon tog procesa u skupu A nalazi se razlika skupova $A \setminus B$), da bi tako ažuriran skup A na kraju bio ispisan.

```
class Program
{
    static void Main(string[] args)
    {
        int[] univerzum = { -9, -8, -7, -6, -5, -4, -3, -2, -1,
                             0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        bool[] skupA = new bool[univerzum.Length];
        bool[] skupB = new bool[univerzum.Length];
        for (int i = 0; i < univerzum.Length; i++)
        {
            if (univerzum[i] % 3 == 0)
            {
                skupA[i] = true;
            }
            else
            {
                skupA[i] = false;
            }
            if (univerzum[i] % 2 == 0)
            {
                skupB[i] = true;
            }
            else
            {
                skupB[i] = false;
            }
        }

        for (int i = 0; i < univerzum.Length; i++)
        {
            if (skupA[i] && skupB[i])
            {
                skupA[i] = false;
            }
        }

        bool pocetak = true;
        Console.Write("{");
        for (int i = 0; i < univerzum.Length; i++)
        {
            if (skupA[i])
            {
```

```
                if (pocetak)
                {
                    pocetak = false;
                }
                else
                {
                    Console.Write(", ");
                }
                Console.Write("{0}", univerzum[i]);
            }
        }
        Console.WriteLine("}");
    }
}
```

Kada se program izvrši, dobijamo sledeći ispis.

```
{-9, -3, 3, 9}
```

Kada multiskup predstavljamo kao niz tipa `int` čija je dužina jednaka kardinalnosti univerzuma, interpretacija je sledeća – svaki element tog niza ima vrednost jednaku *multiplicitetu* (broju pojavljivanja u multiskupu) odgovarajućeg elementa univerzuma. Ovakvo predstavljanje multiskupa naziva se i *histogram*. Na primer, ako je univerzum $\{0, 1, 2, 3, 4\}$, a multiskup koji želimo da predstavimo $\{1, 1, 1, 1, 1, 2, 4, 4\}$, predstavili bismo ga celobrojnim nizom `{ 0, 5, 1, 0, 2 }`.

U sledećem programu, dat je niz jednocifrenih celih brojeva, a zatim se formira multiskup (histogram) koji sadrži multiplicitet (u datom nizu) svakog od brojeva univerzuma. Pošto su svi elementi niza jednocifreni, univerzum je skup brojeva $\{0, 1, \ldots, 9\}$ koji tačno odgovara indeksima niza kojim je predstavljen multiskup, te nije potrebno posebno deklarisati niz koji sadrži univerzum.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 2, 3, 4, 3, 2, 1, 5, 9, 0, 2, 3, 3, 3, 4, 9, 2 };
        int[] multiskup = new int[10];
        for (int i = 0; i < multiskup.Length; i++)
        {
            multiskup[i] = 0;
        }
        for (int i = 0; i < niz.Length; i++)
        {
            multiskup[niz[i]]++;
        }
        Console.WriteLine("Najfrekventniji element datog multiskupa pojavljuje
          se u nizu {0} puta.", multiskup.Max());
    }
}
```

Prilikom izvršavanja programa, ispisuje se poruka da se najfrekventniji element niza `niz` pojavljuje 5 puta, što je u nizu datom u primeru element 3.

3.5 Matrice

Neformalno govoreći, *matricu* možemo posmatrati kao tabelu u koju su smeštene vrednosti istog tipa podataka. Za razliku od nizova koji su jednodimenzionalni, matrice imaju dve dimenzije – podaci su organizovani u redove (vrste) i kolone.

Sintaksa deklaracije matrice čije dimenzije znamo u momentu deklaracije ima dve forme:

```
<tip>[,] <ime> = new <tip>[<broj_vrsta>, <broj_kolona>];
<tip>[,] <ime> = { <niz_nizova_vrednosti> };
```

Na primer, prva varijanta može izgledati ovako:

```
int[,] mat = new int[8, 3];
```

a druga ovako:

```
int[,] nova = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };
```

U prvoj varijanti smo samo deklarisali matricu, dok smo u drugoj varijanti odmah i upisali navedene vrednosti, tako što smo redom naveli vrste matrice (gde je svaka vrsta data kao niz vrednosti koje se nalaze u toj vrsti, i sve vrste moraju imati isti broj elemenata). Dimenzije matrice u prvoj varijanti su eksplicitno date, dok je u drugoj varijanti broj vrsta i broj kolona određen datim podacima. U navedenom primeru, matrica `nova` ima tri vrste i četiri kolone, a struktura joj je sledeća:

1	2	3	4
2	3	4	5
3	4	5	6

U slučaju da u momentu deklaracije ne znamo tačne dimenzije matrice, deklarisaćemo dovoljno veliku matricu da primi sve podatke koje u nju planiramo da smestimo. S obzirom na to da u ovom slučaju broj redova i kolona matrice ne odgovara deklarisanim dimenzijama, pored same matrice potrebne su nam i dve dodatne celobrojne promenljive u kojima ćemo čuvati stvarne dimenzije matrice. Sintaksa deklaracije u ovom slučaju je sledeća:

```
<tip>[,] <ime> = new <tip>[<gornje_ogr_broja_vrsta>, <gornje_ogr_broja_kolona>];
int <nIme> = 0, <mIme> = 0;
```

Na primer,

```
int[,] mat = new int[50, 50];
int nMat = 0, mMat = 0;
```

Za imena promenljivih `<nIme>` i `<mIme>` primenjujemo sličnu konvenciju kao kod nizova – dodaćemo slovo „n“, odnosno „m“, ispred imena matrice, pri čemu samo ime matrice navodimo velikim slovom. Deklarisani broj redova matrice `<mat>` dobijamo pomoću `<mat>.GetLength(0)`, a broj kolona sa `<mat>.GetLength(1)`. Elementu matrice `<mat>` koji se nalazi u vrsti sa indeksom `i` i koloni sa indeksom `j` pristupamo pomoću `<mat>[i, j]`. Ovaj element zvaćemo i *element na poziciji* (i, j).

Kao i kod nizova, indeksi vrsta i kolona matrice počinju od 0, pa npr. matrica sa tri vrste ima vrste sa indeksima 0, 1 i 2.

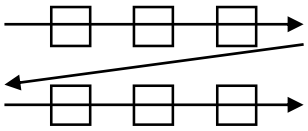
Ako želimo da „prođemo“ kroz sve elemente matrice, za to najčešće koristimo dve ugnježdene FOR petlje. Naredni program najpre menja vrednost elementa date matrice na poziciji $(0,1)$, a zatim ispisuje sve elemente matrice.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 3 }, { 5, 7, 9 } };
        mat[0, 1] = 555;
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                Console.WriteLine(mat[i, j]);
            }
        }
    }
}
```

Prilikom izvršavanja programa dobijamo sledeći ispis.

```
1
555
3
5
7
9
```

Kao što vidimo, ovakvim prolazom kroz matricu (pomoću dve ugnježdene FOR petlje) najpre su redom ispisani svi elementi prve vrste, a zatim redom svi elementi druge vrste. Drugim rečima, redosled prolaska je sledeći:



U narednom programu korisnik najpre unosi broj vrsta i kolona matrice, uz pretpostavku da nijedan od ta dva broja neće biti veći od 20. Nakon toga, od korisnika se očekuje da redom unese elemente matrice. Naposletku, uneta matrica se ispisuje.

```
class Program
{
    static void Main(string[] args)
    {
        double[,] mat = new double[20, 20];
        int nMat = 0, mMat = 0;
```

```
            Console.Write("Unesite broj redova matrice: ");
            nMat = int.Parse(Console.ReadLine());
            Console.Write("Unesite broj kolona matrice: ");
            mMat = int.Parse(Console.ReadLine());
            for (int i = 0; i < nMat; i++)
            {
                for (int j = 0; j < mMat; j++)
                {
                    Console.Write("Unesite element na poziciji ({0}, {1}): ", i, j);
                    mat[i, j] = double.Parse(Console.ReadLine());
                }
            }
            Console.WriteLine();
            Console.WriteLine("Vasa matrica:");
            for (int i = 0; i < nMat; i++)
            {
                for (int j = 0; j < mMat; j++)
                {
                    Console.Write("{0,9:N2}", mat[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
```

Mogući ispis prilikom izvršavanja prethodnog programa sledi.

```
Unesite broj redova matrice: 3
Unesite broj kolona matrice: 2
Unesite element na poziciji (0, 0): 2.2
Unesite element na poziciji (0, 1): 333
Unesite element na poziciji (1, 0): -18
Unesite element na poziciji (1, 1): 3.14159
Unesite element na poziciji (2, 0): 100.1
Unesite element na poziciji (2, 1): 123

Vasa matrica:
     2.20   333.00
   -18.00     3.14
   100.10   123.00
```

U sledećem programu, od korisnika se traži da unese broj vrsta i broj kolona matrice. Nakon toga, kreira se matrica tih dimenzija koja na poziciji (i, j) ima element čija je vrednost $i \cdot j$, a zatim se ta matrica i ispisuje.

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Broj vrsta: ");
        int nMat = int.Parse(Console.ReadLine());
        Console.Write("Broj kolona: ");
        int mMat = int.Parse(Console.ReadLine());
```

```
            int[,] mat = new int[nMat, mMat];
            for (int i = 0; i < nMat; i++)
            {
                for (int j = 0; j < mMat; j++)
                {
                    mat[i, j] = i * j;
                }
            }
            for (int i = 0; i < nMat; i++)
            {
                for (int j = 0; j < mMat; j++)
                {
                    Console.Write("{0,6}", mat[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
```

Mogući ispis prilikom izvršavanja ovog programa sledi.

```
    Broj vrsta: 8
    Broj kolona: 6
         0     0     0     0     0     0
         0     1     2     3     4     5
         0     2     4     6     8    10
         0     3     6     9    12    15
         0     4     8    12    16    20
         0     5    10    15    20    25
         0     6    12    18    24    30
         0     7    14    21    28    35
```

U narednom programu, kreira se niz čiji je broj elemenata jednak broju kolona date matrice, zatim se u svaki element tog niza smešta zbir elemenata koji se nalaze u odgovarajućoj koloni date matrice, a na kraju se taj niz i ispisuje.

```
    class Program
    {
        static void Main(string[] args)
        {
            int[,] mat = { { 1, 2, 3, 4 }, { 2, 3, 4, 5 }, { 3, 4, 5, 6 } };
            int[] niz = new int[mat.GetLength(1)];
            for (int l = 0; l < niz.Length; l++)
            {
                int s = 0;
                for (int k = 0; k < mat.GetLength(0); k++)
                {
                    s += mat[k, l];
                }
                niz[l] = s;
            }

            for (int i = 0; i < niz.Length; i++)
```

```
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }
}
```

Prilikom izvršavanja programa, dobijamo sledeći ispis.

```
6 9 12 15
```

U narednom programu kreiramo kvadratnu matricu koja na glavnoj dijagonali redom ima elemente datog niza, a izvan glavne dijagonale ima jedinice. Zatim tu matricu i ispisujemo.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 3, -1, 3, 4, 12, 21 };
        int n = niz.Length;
        int[,] mat = new int[n, n];
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (i == j)
                {
                    mat[i, j] = niz[i];
                }
                else
                {
                    mat[i, j] = 1;
                }
            }
        }
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                Console.Write("{0, 4}", mat[i, j]);
            }
            Console.WriteLine();
        }
    }
}
```

Kako je dužina glavne dijagonale matrice koju kreiramo jednaka dužini datog niza, i broj vrsta i broj kolona te matrice jednak je dužini datog niza. S obzirom da u programu više puta želimo da iskoristimo vrednost dužine datog niza, praktično je uvesti novu promenljivu `n` čija je vrednost `niz.Length`.

Prilikom izvršavanja programa, dobijamo sledeći ispis.

```
   3   1   1   1   1   1
   1  -1   1   1   1   1
   1   1   3   1   1   1
   1   1   1   4   1   1
   1   1   1   1  12   1
   1   1   1   1   1  21
```

Naredni program kreira i ispisuje kvadratnu matricu u kojoj se dati niz nalazi neposredno ispod glavne dijagonale, sa elementima u obrnutom redosledu, ispod njega su dvojke, a iznad njega su trojke. Glavnu dijagonalu čine elementi čiji su indeksi vrste i kolone jednaki.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 17, -1, 77, 44, 55, -9 };
        int[,] mat = new int[niz.Length + 1, niz.Length + 1];
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                if (i == j + 1)
                {
                    mat[i, j] = niz[niz.Length - i];
                }
                else if (i > j + 1)
                {
                    mat[i, j] = 2;
                }
                else
                {
                    mat[i, j] = 3;
                }
            }
        }
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                Console.Write("{0,4}", mat[i, j]);
            }
            Console.WriteLine();
        }
    }
}
```

Kada se program izvrši, dobijamo sledeći ispis.

```
   3   3   3   3   3   3   3
  -9   3   3   3   3   3   3
   2  55   3   3   3   3   3
```

```
2   2  44   3   3   3   3
2   2   2  77   3   3   3
2   2   2   2  -1   3   3
2   2   2   2   2  17   3
```

Sledeći program ispisuje matricu koja se dobija od date kvadratne matrice kada se elementi na sporednoj dijagonali zamene redom (od donjeg levog ugla ka gornjem desnom uglu) prirodnim brojevima, počevši od broja jedan.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 7, 8 },
                       { 2, 3, 4, 5 },
                       { 9, 1, 5, 6 } };
        int[,] nova = new int[mat.GetLength(1), mat.GetLength(0)];
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                nova[j, i] = mat[i, j];
            }
        }
        for (int i = 0; i < nova.GetLength(0); i++)
        {
            for (int j = 0; j < nova.GetLength(1); j++)
            {
                Console.Write(("{0,4}", nova[i, j]);
            }
            Console.WriteLine();
        }
    }
}
```

Iako je data matrica dvodimenzionalna struktura, vrednosti se menjaju samo na sporednoj dijagonali (koja je jednodimenzionalna), pa te promene možemo izvršiti jednom FOR petljom.

Izvršavanjem programa dobijamo sledeći ispis.

```
1   3   4   4
6   2   3   9
2   2   4   1
1   7   9   9
```

U narednom programu kreira se i ispisuje matrica koja se dobija od date matrice *transponovanjem*, tj. simetričnim preslikavanjem preko glavne dijagonale.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 7, 8 },
                       { 2, 3, 4, 5 },
                       { 9, 1, 5, 6 } };
        int[,] nova = new int[mat.GetLength(1), mat.GetLength(0)];
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                nova[j, i] = mat[i, j];
            }
        }
        for (int i = 0; i < nova.GetLength(0); i++)
        {
            for (int j = 0; j < nova.GetLength(1); j++)
            {
                Console.Write("{0,4}", nova[i, j]);
            }
            Console.WriteLine();
        }
    }
}
```

Nakon što se izvrši program ispisuje se sledeća matrica.

```
1   2   9
2   3   1
7   4   5
8   5   6
```

Ako želimo da kreiramo i ispišemo matricu koja se dobija od date matrice rotacijom za $90°$ u smeru kazaljke na satu, to možemo uraditi koristeći prethodni program u kome je linija

```
nova[j, i] = mat[i, j];
```

zamenjena sledećom linijom

```
nova[j, nova.GetLength(1) - i - 1] = mat[i, j];
```

U tom slučaju, izvršavanjem dobijamo ispis:

```
9   2   1
1   3   2
5   4   7
6   5   8
```

U narednom programu kreira se i ispisuje matrica koja sadrži redom kolone čiji su indeksi elementi datog niza.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 3, 4 },
                       { 2, 3, 4, 5 },
                       { 3, 4, 5, 6 } };
        int[] niz = { 1, 1, 0, 3, 3, 2, 0 };
        int[,] nova = new int[mat.GetLength(0), niz.Length];
        for (int j = 0; j < niz.Length; j++)
        {
            for (int i = 0; i < mat.GetLength(0); i++)
            {
                nova[i, j] = mat[i, niz[j]];
            }
        }
        for (int i = 0; i < nova.GetLength(0); i++)
        {
            for (int j = 0; j < nova.GetLength(1); j++)
            {
                Console.Write("{0,4}", nova[i, j]);
            }
            Console.WriteLine();
        }
    }
}
```

Kada se program izvrši, dobijamo sledeći ispis.

```
   2   2   1   4   4   3   1
   3   3   2   5   5   4   2
   4   4   3   6   6   5   3
```

4 Matematički problemi

U rešavanju matematičkih problema, posebno ukoliko je problem konačne prirode, korišćenje računara može biti od velike pomoći. U ovoj glavi bavićemo se traženjem odgovora na pitanja koja su konjunkcija ili disjunkcija većeg broja uslova. Takođe, razmotrićemo načine da se uhvatimo u koštac sa problemima iz teorije brojeva, kao i problemima u kojima se posmatraju cifre u zapisu broja.

4.1 Konjunkcije i disjunkcije većeg broja uslova

U nekim situacijama potrebno je proveriti da li je istovremeno zadovoljen veći broj uslova, odnosno, preciznije govoreći – proveriti da li je konjunkcija skupa iskaza tačna. Kao što je poznato, konjunkcija skupa iskaza je tačna ako i samo ako su svi iskazi tačni.

Probleme tog tipa generički rešavamo uvođenjem logičke promenljive kojoj najpre dajemo vrednost „tačno“, a zatim prolazimo redom kroz pomenute iskaze i (svaki put) kada naiđemo na netačan iskaz postavljamo vrednost promenljive na „netačno“. Nakon prolaska kroz sve iskaze, promenljiva ima vrednost „tačno“ ako i samo ako su svi iskazi tačni.

Na sličan način proveravamo i disjunkcije većeg broja iskaza, imajući u vidu da je disjunkcija iskaza zapravo ekvivalentna negaciji konjunkcije negacija istih iskaza,

$$x_1 \vee x_2 \vee \ldots \vee x_n = \neg(\neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_n).$$

Kada proveravamo da li je disjunkcija skupa iskaza tačna, logičkoj promenljivoj najpre dajemo vrednost „netačno“, a zatim prolazimo redom kroz pomenute iskaze i (svaki put) kada naiđemo na tačan iskaz postavljamo vrednost promenljive na „tačno“. Nakon prolaska kroz sve iskaze, promenljiva ima vrednost „tačno“ ako i samo ako je bar jedan iskaz tačan.

Sledeći program ispisuje „True“ ako su svi elementi datog niza parni brojevi, a inače ispisuje „False“.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 3, 2, 5, 8, 3, 19, 22 };
        bool sviParni = true;
        for (int i = 0; i < niz.Length; i++)
        {
            if (niz[i] % 2 != 0)
            {
                sviParni = false;
            }
        }
        Console.WriteLine(sviParni);
    }
}
```

Kada se izvrši program, na ekranu će se ispisati `False`.

Primetimo da se u prethodnom programu uvek prolazi kroz sve elemente niza, iako je odgovor na pitanje „Da li su svi elementi datog niza parni brojevi?“ poznat čim naiđemo na prvi neparan element niza. Sledi efikasnija verzija istog programa u kojoj umesto FOR petlje koristimo WHILE petlju, što nam omogućava da ranije završimo prolazak kroz niz ukoliko naiđemo na neparan element.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz = { 3, 2, 5, 8, 3, 19, 22 };
        bool sviParni = true;
        int i = 0;
        while (i < niz.Length && sviParni)
        {
            if (niz[i] % 2 != 0)
            {
                sviParni = false;
            }
            i++;
        }
        Console.WriteLine(sviParni);
    }
}
```

U narednom programu ispisuje se „True“ ako je bar jedan element date matrice jednak nuli, a inače se ispisuje „False“.

Jasno je da je uslov koji treba proveriti zapravo disjunkcija većeg broja uslova.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 3, 4 },
                       { 3, 3, 4, 5 },
                       { 3, 4, 5, 6 } };
        bool jedanNula = false;
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                if (mat[i, j] == 0)
                {
                    jedanNula = true;
                }
            }
        }
        Console.WriteLine(jedanNula);
    }
}
```

Nakon izvršavanja programa na ekranu će se ispisati `False`.

I kod disjunkcije većeg broja uslova proveru možemo izvršiti efikasnije, koristeći WHILE petlju umesto FOR petlje, što nam omogućava da ranije završimo prolazak (čim prvi put naiđemo na zadovoljen uslov). Imajući to u vidu, ponovo rešavamo prethodni problem.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1, 2, 3, 4 },
                       { 3, 8, 4, 5 },
                       { 3, 4, 5, 6 } };
        bool jedanNula = false;
        int i = 0;
        while (!jedanNula && i < mat.GetLength(0))
        {
            int j = 0;
            while (!jedanNula && j < mat.GetLength(1))
            {
                if (mat[i, j] == 0)
                {
                    jedanNula = true;
                }
                j++;
            }
            i++;
        }
        Console.WriteLine(jedanNula);
    }
}
```

Sledeći program ispisuje sve vrste date matrice u kojima se svi elementi završavaju cifrom 3.

```
class Program
{
    static void Main(string[] args)
    {
        int[,] mat = { { 1,  2,  3,   4 },
                       { 3, 33, 43, 503 },
                       { 3,  4,  5,   6 },
                       { 3, 43,  3,   3 } };
        for (int i = 0; i < mat.GetLength(0); i++)
        {
            bool uslov = true;
            for (int j = 0; j < mat.GetLength(1); j++)
            {
                if (mat[i, j] % 10 != 3)
                {
                    uslov = false;
                }
            }
            if (uslov)
            {
                for (int j = 0; j < mat.GetLength(1); j++)
```

```
                {
                    Console.Write("{0, 6} ", mat[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Ispis koji se dobija prilikom izvršavanja programa sledi.

```
     3    33    43   503
     3    43     3     3
```

Naredni program za dva data niza ispisuje „True“ ako prvi niz sadrži drugi niz kao podniz (na uzastopnim pozicijama), a inače ispisuje „False“.

```
class Program
{
    static void Main(string[] args)
    {
        int[] niz1 = { 3, 2, 5, 8, 3, 19, 22, 3, 5, 7, 6, 3, 4 };
        int[] niz2 = { 3, 5, 7 };
        bool jePodniz = false;
        for (int i = 0; i <= niz1.Length - niz2.Length; i++)
        {
            bool poklapaSe = true;
            for (int j = 0; j < niz2.Length; j++)
            {
                if (niz1[i + j] != niz2[j])
                {
                    poklapaSe = false;
                }
            }
            if (poklapaSe)
            {
                jePodniz = true;
            }
        }
        Console.WriteLine(jePodniz);
    }
}
```

Primetimo da se nakon izvršavanje sledećeg bloka naredbi u promenljivoj `poklapaSe` zapravo nalazi odgovor na pitanje *„Da li se* `niz2` *nalazi kao podniz u* `niz1` *počevši od pozicije* `i`*?“*

```
            bool poklapaSe = true;
            for (int j = 0; j < niz2.Length; j++)
            {
                if (niz1[i + j] != niz2[j])
                {
                    poklapaSe = false;
```

```
            }
        }
```

4.2 Teorija brojeva

Naredni odeljak je posvećen elementarnim problemima iz teorije brojeva, odnosno problemima koji se bave celim brojevima, sa posebnim osvrtom na deljivost i proste faktore. Za druge probleme iz ove oblasti, upućujemo zainteresovanog čitaoca na [9].

Kao što smo već videli, deljivost brojeva proveravamo preko ostatka pri (celobrojnom) deljenju, tj. operatora `%`.

Naredni program ispisuje koliko puta je dati prirodan broj deljiv sa dva. Preciznije govoreći, za dat prirodan broj n određuje se i ispisuje najveći prirodan broj k takav da $2^k | n$.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 1248;
        int k = 0;
        while (n % 2 == 0)
        {
            k++;
            n /= 2;
        }
        Console.WriteLine(k);
    }
}
```

Sledeći program računa i ispisuje NZD (najveći zajednički delitelj) dva data prirodna broja. Za nalaženje NZD-a koristimo Euklidov algoritam, koji se oslanja na činjenicu da ako sa k označimo ostatak pri (celobrojnom) deljenju broja n brojem m, važi $\mathrm{NZD}(n, m) = \mathrm{NZD}(m, k)$.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 1248;
        int m = 1692;
        while (m != 0)
        {
            int r = n % m;
            n = m;
            m = r;
        }
        Console.WriteLine(n);
    }
}
```

Prirodan broj veći od jedan je prost ukoliko nema drugih delitelja sem 1 i njega samog. Naredni program ispisuje „True“ ako je dati prirodan broj veći od jedan prost.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 1201;
        bool prost = true;
        for (int i = 2; i < n; i++)
        {
            if (n % i == 0)
            {
                prost = false;
            }
        }
        Console.WriteLine(prost);
    }
}
```

Imajući u vidu da je svaki složen broj n deljiv nekim brojem između 2 i $\sqrt{n}$, prethodni program možemo unaprediti tako da proverava samo brojeve u tom intervalu, kao i da prekine proveru čim se pronađe prvi delitelj broja n.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 1201;
        bool prost = true;
        int i = 2;
        while (prost && i <= Math.Sqrt(n))
        {
            if (n % i == 0)
            {
                prost = false;
            }
            i++;
        }
        Console.WriteLine(prost);
    }
}
```

Svaki prirodan broj veći od jedan može se na jedinstven način predstaviti kao proizvod prostih brojeva, što nazivamo *razvoj na proste faktore* tog broja. Tako, na primer, $720 = 2^4 3^2 5^1$.

Naredni program za dat prirodan broj ispisuje razvoj na proste faktore tog broja. Prilikom izračunavanja, proste faktore ćemo redom smeštati u jedan niz, a odgovarajuće stepene u drugi niz.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 60214000;
        int[] fakt = new int[100];
        int[] step = new int[100];
        int d = 0;
        for (int i = 2; i <= n; i++)
        {
            int s = 0;
            while (n % i == 0)
            {
                s++;
                n /= i;
            }
            if (s > 0)
            {
                fakt[d] = i;
                step[d] = s;
                d++;
            }
        }
        for (int i = 0; i < d; i++)
        {
            Console.WriteLine("{0} {1}", fakt[i], step[i]);
        }
    }
}
```

Vredi napomenuti da se vrednost promenljive `n` (koja figurira u uslovu `i <= n` u prvoj FOR petlji) smanjuje tokom izvršavanja tela petlje.

Izvršavanjem programa dobijamo sledeći ispis.

```
 2 4
 5 3
 7 1
11 1
17 1
23 1
```

Prirodan broj je *savršen* ako je jednak zbiru svih svojih pravih delitelja (delitelja koji su manji od tog broja). Na primer, broj 28 je savršen, jer je $28 = 1 + 2 + 4 + 7 + 14$. Naredni program ispisuje sve savršene brojeve koji su manji ili jednaki datom prirodnom broju n.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 10000;
        for (int i = 1; i <= n; i++)
        {
```

```
            int s = 0;
            for (int j = 1; j <= i / 2; j++)
            {
                if (i % j == 0)
                {
                    s += j;
                }
            }
            if (i == s)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Kada se program izvrši, ispisuje se:

```
6
28
496
8128
```

Naredni program ispisuje redom sve proste brojeve manje od datog prirodnog broja n. Za realizaciju programa koristićemo tzv. *Eratostenovo sito*, algoritam čiji opis sledi.

Počinjemo sa nizom svih prirodnih brojeva od 2 do $n-1$. Najpre posmatramo prvi broj u nizu, tj. 2, te „precrtamo“ sve ostale brojeve u nizu koji su deljivi sa 2. Zatim posmatramo sledeći neprecrtan broj, tj. 3, pa opet precrtamo sve ostale brojeve koji su deljivi sa 3. Proces ponavljamo redom i sa narednim neprecrtanim brojevima, sve dok ne dođemo do kraja niza. U tom momentu u nizu su ostali neprecrtani svi prosti brojevi manji od n, te nam ostaje samo da ih ispišemo.

Informaciju o tome da li je broj i precrtan čuvaćemo u nizu `precrtan` (koji je `bool` tipa) na poziciji sa indeksom i – vrednost `true` znači da je broj i precrtan, a vrednost `false` da nije. S obzirom da se brojevi 0 i 1 ne pojavljuju u algoritmu (u inicijalnom nizu najmanji broj je 2), prva dva člana niza `precrtan` nećemo koristiti.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 41;
        bool[] precrtan = new bool[n];
        for (int i = 2; i < n; i++)
        {
            precrtan[i] = false;
        }
        for (int i = 2; i < n; i++)
        {
            if (!precrtan[i])
            {
                for (int j = 2 * i; j < n; j += i)
                {
```

```
                    precrtan[j] = true;
                }
            }
        }
        for (int i = 2; i < n; i++)
        {
            if (!precrtan[i])
            {
                Console.Write("{0} ", i);
            }
        }
        Console.WriteLine();
    }
}
```

Kada se program izvrši, dobija se sledeći ispis.

```
2 3 5 7 11 13 17 19 23 29 31 37
```

4.3 Cifre u zapisu broja

Ako posmatramo broj 31706, odmah nam je jasno da su cifre ovog broja 3, 1, 7, 0 i 6. Ali ako želimo da napišemo program koji kreira niz cifara datog broja, taj posao nije trivijalan. Za dat prirodan broj n, njegovu cifru jedinica (prvu cifru zdesna) dobijamo kao ostatak pri deljenju sa 10, tj. `n % 10`. S druge strane, broj koji se dobija kao rezultat celobrojnog deljenja n sa 10, dakle `n / 10`, u zapisu ima redom iste cifre kao n, osim što mu poslednja cifra nedostaje. Na primer, ako promenljiva `n` ima vrednost 31706, vrednost izraza `n % 10` je 6, a vrednost izraza `n / 10` je 3170. Prema tome, uzastopnim primenama ove dve operacije možemo dobiti sve cifre datog broja.

Naredni program kreira i ispisuje niz koji redom sadrži cifre datog broja. U programu se koristi formula $\lfloor \log_{10} n \rfloor + 1$ koja daje broj cifara u zapisu prirodnog broja n.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 415809;
        int l = (int)Math.Floor(Math.Log(n, 10)) + 1;
        int[] cifre = new int[l];
        for (int i = l - 1; i >= 0; i--)
        {
            cifre[i] = n % 10;
            n /= 10;
        }
        for (int i = 0; i < l; i++)
        {
            Console.Write("{0} ", cifre[i]);
        }
        Console.WriteLine();
    }
}
```

Kad se program pokrene, dobijamo sledeći ispis.

```
4 1 5 8 0 9
```

Za neke prostije manipulacije ciframa nije neophodno kreirati niz koji sadrži cifre, te u mnogim takvim slučajevima možemo koristiti i WHILE petlju za „prolaz“ kroz cifre. Kod ovog pristupa nije nam unapred potreban broj cifara broja.

U narednom programu ispisuju se svi prirodni brojevi manji ili jednaki n čiji je zbir cifara deljiv sa 18.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 400;
        for (int i = 1; i <= n; i++)
        {
            int broj = i;
            int zbirCifara = 0;
            while (broj != 0)
            {
                zbirCifara += broj % 10;
                broj /= 10;
            }
            if (zbirCifara % 18 == 0)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Sledi ispis koji se dobija izvršavanjem programa.

```
99
189
198
279
288
297
369
378
387
396
```

Kada prolazimo kroz cifre (na bilo koji od goreopisanih načina) moramo imati u vidu da se vrednost promenljive kroz čije cifre prolazimo gubi, tj. na kraju tog procesa promenljiva će imati vrednost 0. Ukoliko ne želimo i/ili ne smemo da menjamo vrednost te promenljive (kao što je slučaj u prethodnom primeru, gde je potrebno proći kroz cifre promenljive koja je brojač FOR petlje, i koju posle treba ispisati) onda najpre tu vrednost moramo iskopirati u pomoćnu promenljivu, a potom ceo

postupak nalaženja cifara uraditi sa tom pomoćnom promenljivom – baš kao što smo uradili u prethodnom primeru.

Prelazimo sada na obrnut proces – u narednom programu izračunava se i ispisuje broj čije su cifre redom navedene u datom nizu.

```
class Program
{
    static void Main(string[] args)
    {
        int[] cifre = { 5, 3, 0, 4, 9, 2, 3 };
        int n = 0;
        for (int i = 0; i < cifre.Length; i++)
        {
            n = 10 * n + cifre[i];
        }
        Console.WriteLine(n);
    }
}
```

Prilikom izvršavanja programa dobija se sledeći ispis.

```
5304923
```

Sledeći program ispisuje sve prirodne brojeve i između datog prirodnog broja n i datog prirodnog broja m, koji su deljivi brojem koji se dobija od i izbacivanjem svih cifara na parnim pozicijama (gledano sleva) – druge, četvrte, itd.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 88000;
        int m = 94000;
        for (int i = n + 1; i < m; i++)
        {
            int broj = i;
            int l = (int)Math.Floor(Math.Log(broj, 10)) + 1;
            int[] cifre = new int[l];
            for (int j = l - 1; j >= 0; j--)
            {
                cifre[j] = broj % 10;
                broj /= 10;
            }

            int novi = 0;
            for (int j = 0; j < cifre.Length; j += 2)
            {
                novi = 10 * novi + cifre[j];
            }

            if (i % novi == 0)
            {
```

```
                    Console.WriteLine("{0} / {1} = {2}", i, novi, i / novi);
                }
            }
        }
    }
```

Prilikom izvršavanja programa dobija se sledeći ispis.

```
88275 / 825 = 107
88740 / 870 = 102
88779 / 879 = 101
88880 / 880 = 101
89022 / 802 = 111
89100 / 810 = 110
89345 / 835 = 107
89464 / 844 = 106
89991 / 891 = 101
90000 / 900 = 100
90432 / 942 = 96
90909 / 999 = 91
91001 / 901 = 101
91584 / 954 = 96
91675 / 965 = 95
92112 / 912 = 101
93223 / 923 = 101
93605 / 965 = 97
```

4.4 Brojni sistemi

Uobičajeni prikaz brojeva ciframa od 0 do 9 nazivamo dekadni sistem, ili sistem sa bazom 10. Kao što smo već videli, svaka cifra u zapisu ima svoju „težinu“ – prva cifra zdesna je cifra jedinica, zatim sledi cifra desetica, pa cifra stotina, itd. Imajući to u vidu, interpretiramo zapis broja, i tako je, na primer, $3176 = 3 \cdot 10^3 + 1 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0$.

Umesto baze 10, za bazu možemo uzeti i bilo koji drugi prirodan broj $b > 1$. U brojnom sistemu sa bazom b za zapis koristimo b različitih „cifara“, čije su vrednosti od 0 do $b - 1$. Ako broj nije u dekadnom sistemu, navešćemo bazu b u indeksu broja da naglasimo u kom sistemu je zapis. Tako, na primer, imamo $2143_5 = 2 \cdot 5^3 + 1 \cdot 5^2 + 4 \cdot 5^1 + 3 \cdot 5^0 = 298$, ili $10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 22$.

Algoritam za kreiranje niza cifara u zapisu datog broja u sistemu sa datom bazom i algoritam za izračunavanje broja čije su cifre u zapisu u sistemu sa datom bazom date u nizu dobijamo kao generalizaciju već pomenutih odgovarajućih algoritama za dekadni sistem.

Naredni program za date prirodne brojeve n i b kreira i ispisuje niz koji redom sadrži cifre broja n u zapisu u sistemu sa bazom b. U programu se koristi formula $\lfloor \log_b n \rfloor + 1$ koja daje broj cifara u zapisu prirodnog broja n u sistemu sa bazom b.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 415809, b = 8;
        int l = (int)Math.Floor(Math.Log(n, b)) + 1;
        int[] cifre = new int[l];
        for (int i = l - 1; i >= 0; i--)
        {
            cifre[i] = n % b;
            n /= b;
        }
        for (int i = 0; i < l; i++)
        {
            Console.Write("{0} ", cifre[i]);
        }
        Console.WriteLine();
    }
}
```

Sledi ispis koji se dobija izvršavanjem programa.

```
1 4 5 4 1 0 1
```

Naredni program ispisuje sve prirodne brojeve između dva data broja čije cifre u zapisu u sistemu sa datom bazom čine rastući niz, a pored svakog takvog broja ispisuje se i njegov zapis u sistemu sa datom bazom.

```
class Program
{
    static void Main(string[] args)
    {
        int n = 50, m = 310;
        int b = 6;
        for (int i = n; i <= m; i++)
        {
            int broj = i;
            int l = (int)Math.Floor(Math.Log(broj, b)) + 1;
            int[] cifre = new int[l];
            for (int j = l - 1; j >= 0; j--)
            {
                cifre[j] = broj % b;
                broj /= b;
            }
            bool rastuci = true;
            for (int j = 0; j < l - 1; j++)
            {
                if (cifre[j] >= cifre[j + 1])
                {
                    rastuci = false;
                }
            }
```

```
            if (rastuci)
            {
                Console.Write("{0}, ", i);
                for (int j = 0; j < l; j++)
                {
                    Console.Write("{0}", cifre[j]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Kada se program pokrene, dobija se sledeći ispis.

```
51, 123
52, 124
53, 125
58, 134
59, 135
65, 145
94, 234
95, 235
101, 245
137, 345
310, 1234
```

Vredi napomenuti da se, kao i za dekadni sistem, prolazak kroz cifre datog broja može uraditi i WHILE petljom.

I kod računanja vrednosti broja čije cifre znamo, možemo generalizovati pristup koji smo koristili za sistem sa bazom 10. U narednom programu izračunava se i ispisuje broj čije su cifre u zapisu u sistemu sa datom bazom redom navedene u datom nizu.

```
class Program
{
    static void Main(string[] args)
    {
        int b = 2;
        int[] cifre = { 1, 0, 0, 1, 0, 1, 1 };
        int n = 0;
        for (int i = 0; i < cifre.Length; i++)
        {
            n = b * n + cifre[i];
        }
        Console.WriteLine(n);
    }
}
```

Prilikom izvršavanja programa dobija se sledeći ispis.

```
75
```

U narednom programu ispisuje se broj čiji se binarni zapis (zapis u sistemu sa bazom dva) dobija kada se cifre datog neparnog broja n u binarnom zapisu poređaju naopako (od poslednje ka prvoj).

```
class Program
{
    static void Main(string[] args)
    {
        int n = 2017, b = 2;
        int l = (int)Math.Floor(Math.Log(n, b)) + 1;
        int[] cifre = new int[l];
        for (int i = l - 1; i >= 0; i--)
        {
            cifre[i] = n % b;
            n /= b;
        }

        int obrnut = 0;
        for (int i = cifre.Length - 1; i >= 0; i--)
        {
            obrnut = b * obrnut + cifre[i];
        }
        Console.WriteLine(obrnut);
    }
}
```

Prilikom izvršavanja programa ispisuje se broj 1087, jer 2017 je u binarnom zapisu 11111100001, a binarni zapis 1087 je 10000111111.

5 Metodi i rekurzija

Kako programi koje pišemo postaju duži i kompleksniji, snalaženje u kôdu i sagledavanje programa kao celine prestaje da bude rutinska radnja. U takvim situacijama uputno je organizovati delove kôda u celine, koje onda po potrebi možemo izvršavati. Ovakav koncept organizacije programa naziva se *modularno programiranje,* a mi ćemo za takvo programiranje koristiti tzv. *metode.*

5.1 Metodi

Najprostije govoreći, metod je imenovani deo kôda. Korišćenjem metoda možemo značajno smanjiti strukturalnu kompleksnost programa i povećati njegovu čitljivost.

Svi programi koje smo do sada pisali imali su samo jedan metod – `Main` metod, koji se izvršava kada pokrenemo program, a sada ćemo videti kako se mogu uvesti i koristiti drugi metodi. Pre nego što pređemo na formalniji opis ovog koncepta, daćemo par jednostavnih primera.

Sledeći program pored metoda `Main` sadrži i metod `Prvi`, koji prilikom izvršavanja ispisuje navedeni string.

```
class Program
{
    static void Prvi()
    {
        Console.WriteLine("Tralalalaaa...!");
    }

    static void Main(string[] args)
    {
        for (int i = 1; i <= 6; i++)
        {
            Prvi();
        }
    }
}
```

Kada se pokrene program izvršava se metod `Main`, a s obzirom na to da se poziv metoda `Prvi` nalazi unutar FOR petlje u metodu `Main`, on biva pozvan šest puta i dobijamo sledeći ispis.

```
Tralalalaaa...!
Tralalalaaa...!
Tralalalaaa...!
Tralalalaaa...!
Tralalalaaa...!
Tralalalaaa...!
```

U narednom programu uvodi se metod `Zbir`, koji sabira dva cela broja. Za razliku od metoda `Prvi` iz prethodnog programa, ovaj metod vraća vrednost i ima ulazne parametre – vraćena vrednost je tipa `int`, a dva ulazna parametra su `a` i `b`, oba takođe tipa `int`. U telu metoda `Zbir` deklariše se promenljiva `zbir` koja dobija vrednost jednaku zbiru `a` i `b`, a zatim ta vrednost biva vraćena.

```
class Program
{
    static int Zbir(int a, int b)
    {
        int zbir = a + b;
        return zbir;
    }

    static void Main(string[] args)
    {
        int a = 2;
        Console.WriteLine(Zbir(3, 4));
        Console.WriteLine(Zbir(a - 1, 2 * a + 1));
        Console.WriteLine(Zbir(Zbir(3, 1), Zbir(2, 3)));
    }
}
```

Prilikom izvršavanja programa više puta se poziva metod `Zbir`, i dobijamo sledeći ispis.

```
7
6
9
```

Naredni program pored metoda `Main` sadrži još dva metoda – metod `Polinom` koji vraća vrednost polinoma $x^3 + 3x^2 - 4x + 1$ za onu realnu vrednost x koja je prosleđena kao parametar, i metod `IspisiVrednost` koji ispisuje rečenicu u kojoj je navedena vrednost prosleđenog parametra, kao i vrednost pomenutog polinoma za taj parametar.

```
class Program
{
    static double Polinom(double x)
    {
        return (Math.Pow(x, 3) + 3 * Math.Pow(x, 2) - 4 * x + 1);
    }

    static void IspisiVrednost(double y)
    {
        Console.WriteLine("Vrednost polinoma u {0,4:N2} je {1,5:N2}.", y,
                          Polinom(y));
    }

    static void Main(string[] args)
    {
        IspisiVrednost(Math.PI);
        IspisiVrednost(2);
    }
}
```

Primetimo da se metod `Polinom` ne poziva direktno iz metoda `Main`, već samo iz metoda `IspisiVrednost`. Kada se program izvrši, dolazi do sledećeg ispisa.

```
Vrednost polinoma u 3.14 je 49.05.
Vrednost polinoma u 2.00 je 13.00.
```

Generalno gledano, svi metodi koje uvodimo biće u skladu sa narednim opisom sintakse.

```
static <tip> <ime>(<parametri>)
{
       <niz_naredbi>
}
```

Prvi red predstavlja *deklaraciju metoda*, dok se u vitičastim zagradama nalazi *telo metoda*. Pritom, `<tip>` određuje tip vrednosti koju metod vraća. Ako je taj tip `void` (kao što je slučaj u metodu `IspisiVrednost` u prethodnom programu, ili metodu `Main`), metod ne vraća vrednost. Potom se navodi ime metoda, koje ćemo birati tako da počinje velikim slovom (vodeći se pri tome ostalim pravilima koja smo pominjali prilikom imenovanja promenljivih). Naposletku, u zagradama se navode prosleđeni parametri odvojeni zarezom – za svaki od njih prvo tip pa ime. Ako želimo da metod nema nijedan prosleđeni parametar, ostavljamo prazne zagrade.

Ukoliko tip vrednosti koju metod vraća nije `void`, u telu metoda mora da se nađe naredba `return` posle koje sledi izraz čiju vrednost metod vraća. Tip tog izraza mora odgovarati tipu podataka koji je naveden u deklaraciji metoda.

Metod se poziva navođenjem imena i odgovarajućeg broja izraza u zagradama. Pritom, broj i tipovi izraza moraju da odgovaraju broju i tipovima parametara metoda. Poziv metoda možemo izvršiti iz bilo kog metoda programa.

U narednom programu ispisuje se „True“ ako su u datoj matrici (koja sadrži prirodne brojeve) broj vrsta, broj kolona i svi elementi prosti brojevi, a inače se ispisuje „False“.

S obzirom na to da više puta treba izvršiti proveru da li je prirodan broj prost, zgodno je uvesti metod koji vrši tu proveru.

```
class Program
{
    static bool Prost(int n)
    {
        if (n < 2)
        {
            return false;
        }
        else
        {
            bool prost = true;
            int i = 2;
            while (prost && i <= Math.Sqrt(n))
            {
                if (n % i == 0)
                {
                    prost = false;
                }
                i++;
            }
            return prost;
        }
```

```
    }

    static void Main(string[] args)
    {
        int[,] mat = { { 3, 2, 7, 5, 11},
                       { 2, 2, 2, 2, 17},
                       { 3, 7, 2, 3, 37} };
        bool odgovor = true;
        if (!Prost(mat.GetLength(0)) || !Prost(mat.GetLength(1)))
        {
            odgovor = false;
        }
        else
        {
            for (int i = 0; i < mat.GetLength(0); i++)
            {
                for (int j = 0; j < mat.GetLength(1); j++)
                {
                    if (!Prost(mat[i, j]))
                    {
                        odgovor = false;
                    }
                }
            }
        }
        Console.WriteLine(odgovor);
    }
}
```

Naredni program ispisuje sve nizove koji se dobijaju od datog niza tako što se jedan blok elemenata, koji se sastoji od dva ili više uzastopnih elemenata, obrne naopako. Primetimo da je svaki takav blok jedinstveno određen indeksom a prvog elementa bloka, i indeksom b poslednjeg elementa bloka, gde važi $a < b$. Prema tome, potrebno je „proći“ kroz sve takve parove indeksa, za svaki par izvršiti obrtanje bloka, ispisati dobijeni niz, a zatim ponovnim obrtanjem istog bloka vratiti niz u prvobitno stanje (kako bi bio spreman za naredno obrtanje). Obrtanje bloka i ispis niza izvršićemo pomoću posebnih metoda.

```
class Program
{
    static void ObrtanjeBloka(int[] niz, int a, int b)
    {
        for (int i = a; i <= (a + b) / 2; i++)
        {
            int pomocna = niz[i];
            niz[i] = niz[b - (i - a)];
            niz[b - (i - a)] = pomocna;
        }
    }

    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
```

```
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void Main(string[] args)
    {
        int[] niz = { 1, 2, 3, 4, 5, 6};
        for (int a = 0; a < niz.Length - 1; a++ )
        {
            for (int b = a + 1; b < niz.Length; b++)
            {
                ObrtanjeBloka(niz, a, b);
                IspisiNiz(niz);
                ObrtanjeBloka(niz, a, b);
            }
        }
    }
}
```

Kada se program izvrši, dolazi do sledećeg ispisa.

```
2 1 3 4 5 6
3 2 1 4 5 6
4 3 2 1 5 6
5 4 3 2 1 6
6 5 4 3 2 1
1 3 2 4 5 6
1 4 3 2 5 6
1 5 4 3 2 6
1 6 5 4 3 2
1 2 4 3 5 6
1 2 5 4 3 6
1 2 6 5 4 3
1 2 3 5 4 6
1 2 3 6 5 4
1 2 3 4 6 5
```

5.2 Rekurzija

U programiranju, *rekurzija* je koncept zgodan za primenu kod problema čije se rešenje može opisati rekurzivno, odnosno onih problema čije se rešavanje prirodno može svesti na rešavanje jedne ili više instanci istog problema, ali manjeg obima. S obzirom na to da je u pitanju apstraktan koncept, sa njim ćemo se upoznavati postepeno, uz mnoštvo primera.

Kod korišćenja rekurzije najpre je potrebno formulisati rešenje problema u rekurzivnom obliku. Počećemo jednostavnim zadatkom, koristeći rekurziju da napišemo program koji za dat prirodan broj n redom ispisuje brojeve od n do 1.

Rešavanje ovog problema svakako možemo započeti ispisivanjem broja n. U slučaju da je $n = 1$, uradili smo sve što treba. Ako je $n > 1$, preostaje nam još da redom ispišemo brojeve od $n - 1$ do 1.

No, primetimo da je problem „ispisa brojeva od $n-1$ do 1“ zapravo isti kao početni problem, za $n-1$ umesto n.

Ovakvu (rekurzivnu) formulaciju koristimo da rešimo problem korišćenjem metoda koji će imati *rekurzivne pozive*, odnosno – pozivaće sam sebe.

```
class Program
{
    static void Ispisi(int n)
    {
        Console.WriteLine(n);
        if (n > 1)
        {
            Ispisi(n - 1);
        }
    }

    static void Main(string[] args)
    {
        int n = 4;
        Ispisi(n);
    }
}
```

Kada se program pokrene, izvršava se metod `Main` i dobijamo sledeći ispis.

```
4
3
2
1
```

Pritom, metod `Main` poziva metod `Ispisi` sa parametrom 4. Tokom njegovog izvršavanja ponovo se poziva `Ispisi`, ovaj put sa parametrom 3. Taj metod dalje poziva `Ispisi` sa parametrom 2, iz koga se poziva `Ispisi` sa parametrom 1. U ovom slučaju, s obzirom na to da je parametar 1, nema rekurzivnog poziva, te se pozvani metodi završavaju, jedan za drugim, u obrnutom redosledu od redosleda pozivanja.

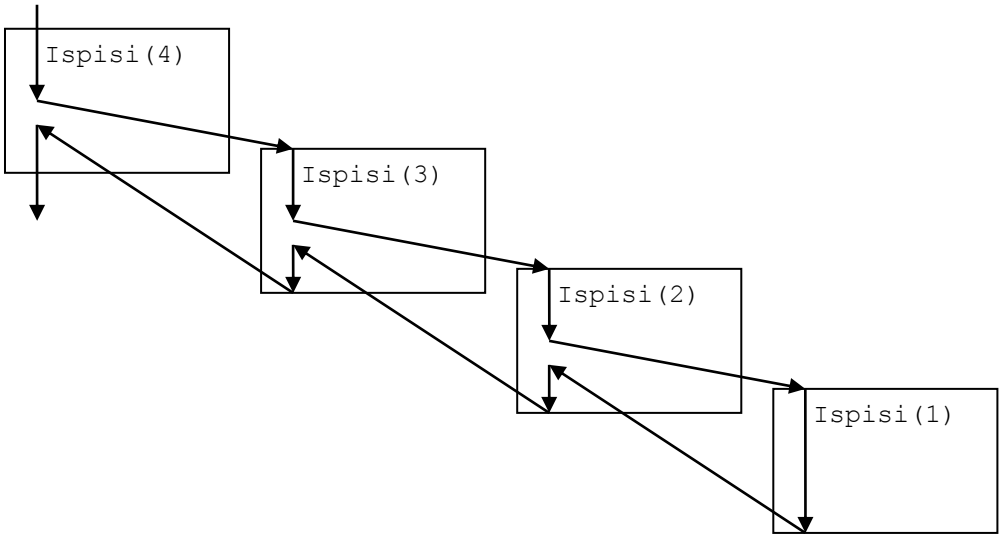
Iako je u pitanju isti metod, ova četiri poziva zapravo treba posmatrati kao četiri *kopije* metoda `Ispisi`. Redosled izvršavanja prikazan je na slici 1.

Treba imati u vidu da se tokom izvršavanja poziva `Ispisi(1)` još nije završilo izvršavanje nijednog od tri poziva tog metoda koji su prethodno počeli da se izvršavaju – izvršavanje svakog od njih je zaustavljeno u momentu rekurzivnog poziva, i nastavlja se tek kada se završi izvršavanje tog rekurzivnog poziva. Prema tome, tek nakon izvršavanja poziva `Ispisi(1)` privodi se kraju i izvršavanje poziva `Ispisi(2)`, potom `Ispisi(3)`, i tek na kraju `Ispisi(4)`.

Kod rekurzivnih poziva metoda, parametri metoda i promenljive unutar njega u svakom od poziva su nezavisni od parametara i promenljivih u ostalim pozivima, iako imaju ista imena – za svaki poziv prave se nove kopije ovih parametara. Konkretno, u prethodnom primeru, svaki poziv metoda `Ispisi` ima po jedan parametar n, i svi ti parametri su nezavisni.

Da dodatno ilustrujemo redosled izvršavanja u rekurzivnim pozivima metoda, uradićemo jedan eksperiment – u programu koji smo napisali premestićemo naredbu `Console.WriteLine(n)` sa

početka na kraj metoda `Ispisi`, a sve ostalo će ostati isto kao i pre. Novi metod ćemo nazvati `Ispisi2`.



Slika 1. Redosled izvršavanja metoda Ispisi

```
class Program
{
    static void Ispisi2(int n)
    {
        if (n > 1)
        {
            Ispisi2(n - 1);
        }
        Console.WriteLine(n);
    }

    static void Main(string[] args)
    {
        int n = 4;
        Ispisi2(n);
    }
}
```

Ova promena na prvi pogled može da deluje beznačajno, ali izvršavanje programa neće biti isto. Naime, u svakom od poziva metoda `Ispisi2` u kojima je došlo do rekurzivnog poziva unutar IF-a, parametar `n` se ispisuje tek nakon završetka tog rekurzivnog poziva. Samim tim, iako je struktura rekurzivnih poziva potpuno ista kao kod metoda `Ispisi` (slika iznad), parametri se ispisuju onim redom kojim pozvani metodi završavaju izvršavanje – najpre se završava `Ispisi2(1)`, zatim `Ispisi2(2)`, pa `Ispisi2(3)`, i konačno `Ispisi2(4)`, što znači da će brojevi ovaj put biti ispisani u rastućem poretku:

```
1
2
3
4
```

U narednom programu se za dati prirodan broj ispisuje vrednost dvostrukog faktorijela.

Dvostruki faktorijel prirodnog broja n je proizvod svih prirodnih brojeva manjih ili jednakih n koji su iste parnosti kao n, i označava se sa $n!!$. Tako je, na primer, $9!! = 9 \cdot 7 \cdot 5 \cdot 3 \cdot 1 = 945$. Ovaj problem ćemo rešiti preko metoda sa rekurzivnim pozivom, koristeći rekurzivnu definiciju dvostrukog faktorijela,

$$n!! = \begin{cases} n \cdot (n-2)!!, & n > 2 \\ n, & n = 1, n = 2 \end{cases}.$$

```
class Program
{
    static int DvostrukiFaktorijel(int n)
    {
        if (n <= 2)
        {
            return n;
        }
        else
        {
            return (n * DvostrukiFaktorijel(n - 2));
        }
    }

    static void Main(string[] args)
    {
        int n = 15;
        Console.WriteLine(DvostrukiFaktorijel(n));
    }
}
```

Kada se program izvrši, dobijamo sledeći ispis.

```
2027025
```

Fibonačijev niz je niz prirodnih brojeva čija su prva dva člana jedinice, a svaki sledeći je jednak zbiru prethodna dva. Prema tome, ako sa $f(n)$ označimo n-ti član niza, imamo $f(1) = 1$, $f(2) = 1$, $f(3) = 2$, $f(4) = 3$, $f(5) = 5$, $f(6) = 8$, $f(7) = 13$, itd. Rekurzivni zapis definicije niza je sledeći:

$$f(n) = \begin{cases} f(n-1) + f(n-2), & n > 2 \\ 1, & n = 1, n = 2 \end{cases},$$

pa imajući ovo u vidu možemo da napišemo program koji korišćenjem rekurzije za dati prirodan broj n vraća n-ti član Fibonačijevog niza.

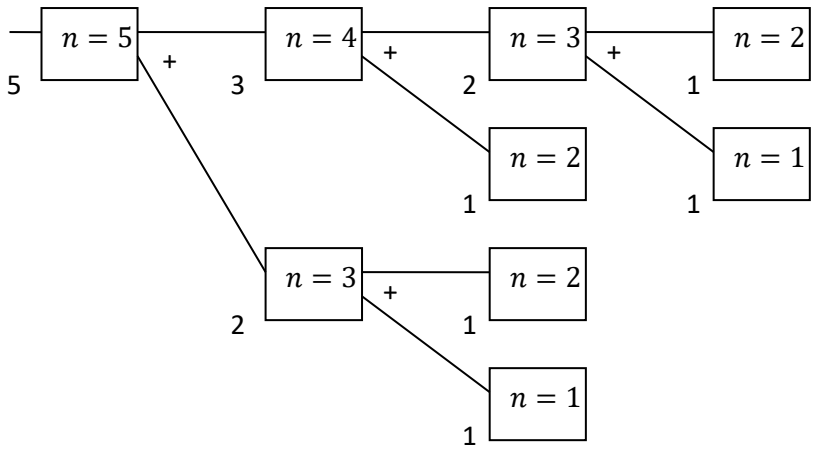
```
class Program
{
    static int Fibonaci(int n)
    {
        if (n <= 2)
        {
            return 1;
        }
        else
        {
            return (Fibonaci(n - 1) + Fibonaci(n - 2));
        }
    }

    static void Main(string[] args)
    {
        int n = 18;
        Console.WriteLine(Fibonaci(n));
    }
}
```

Ispis programa prilikom izvršavanja sledi.

```
2584
```

S obzirom na to da se iz svakog poziva metoda `Fibonaci` (kada je vrednost prosleđenog parametra bar tri) dva puta rekurzivno poziva metod `Fibonaci`, dijagram poziva funkcija ima razgranatu strukturu. Na slici 2 prikazan je proces izračunavanja petog člana niza. Svaki poziv funkcije označićemo vrednošću parametra n za koju je funkcija pozvana.



Slika 2. Rekurzivno izračunavanje petog člana Fibonačijevog niza

U opštem slučaju, kada želimo da proverimo da li neki niz sadrži element koji ima odgovarajuća svojstva, ne preostaje nam ništa drugo nego da redom prođemo kroz sve elemente niza i za svaki ponaosob izvršimo proveru. Ali ako znamo da je niz *sortiran* u neopadajućem poretku i zanima nas da li se u nizu nalazi element koji ima određenu vrednost – nazovimo je x, pretragu možemo izvršiti znatno brže. Pogledaćemo najpre element na srednjoj poziciji (ako niz ima paran broj elemenata,

pogledaćemo neki od dva elementa u sredini). Ako je taj element jednak x, potraga je završena. Ako je veći od x, jasno je da se traženi element, ako postoji, nalazi u prvoj polovini niza. Ako je manji od x, jasno je da se traženi element, ako postoji, nalazi u drugoj polovini niza. Sada isti postupak možemo ponoviti za polovinu niza u kojoj smo „locirali“ traženi element, posmatrajući element na srednjoj poziciji te polovine. Na taj način, pretražićemo ceo niz polovljenjem u najviše (reda veličine) $\log_2 n$ poteza[1], i dobićemo odgovor na pitanje da li se x nalazi u nizu ili ne. Ovakav algoritam pretrage kroz sortiran niz naziva se *binarna pretraga* ili *pretraga polovljenjem*.

Neka je, na primer, dat sortiran niz `{1, 4, 8, 12, 47, 100, 203, 342, 420, 560, 630, 750}`, a u njemu se traži element $x = 47$. U tabeli 8 prikazaćemo po koracima kako izgleda pretraga polovljenjem na ovom primeru, kao i deo niza koji se posmatra u svakom koraku. Žutim je označen element niza sa kojim se poredi traženi element, kao i relacija između posmatranog i traženog elementa.

Traženi element	Deo niza koji se posmatra	Relacija
$x = 47$	{1, 4, 8, 12, 47, **100**, 203, 342, 420, 560, 630, 750}	$x < 100$
$x = 47$	{1, 4, **8**, 12, 47}	$x > 8$
$x = 47$	{**12**, 47}	$x > 12$
$x = 47$	{**47**}	$x = 47$

Tabela 8. Primer pretrage polovljenjem

U narednom programu napisaćemo metod koji binarnom pretragom pretražuje prosleđeni *sortirani* niz u potrazi za prosleđenim prirodnim brojem, i vraća (kao logičku vrednost) odgovor na pitanje da li je broj pronađen u nizu ili ne. Pored niza i broja koji tražimo, metodu se prosleđuju još dva parametra, a i b, početni i krajnji indeks (zatvorenog) intervala elemenata niza u kome se broj može nalaziti.

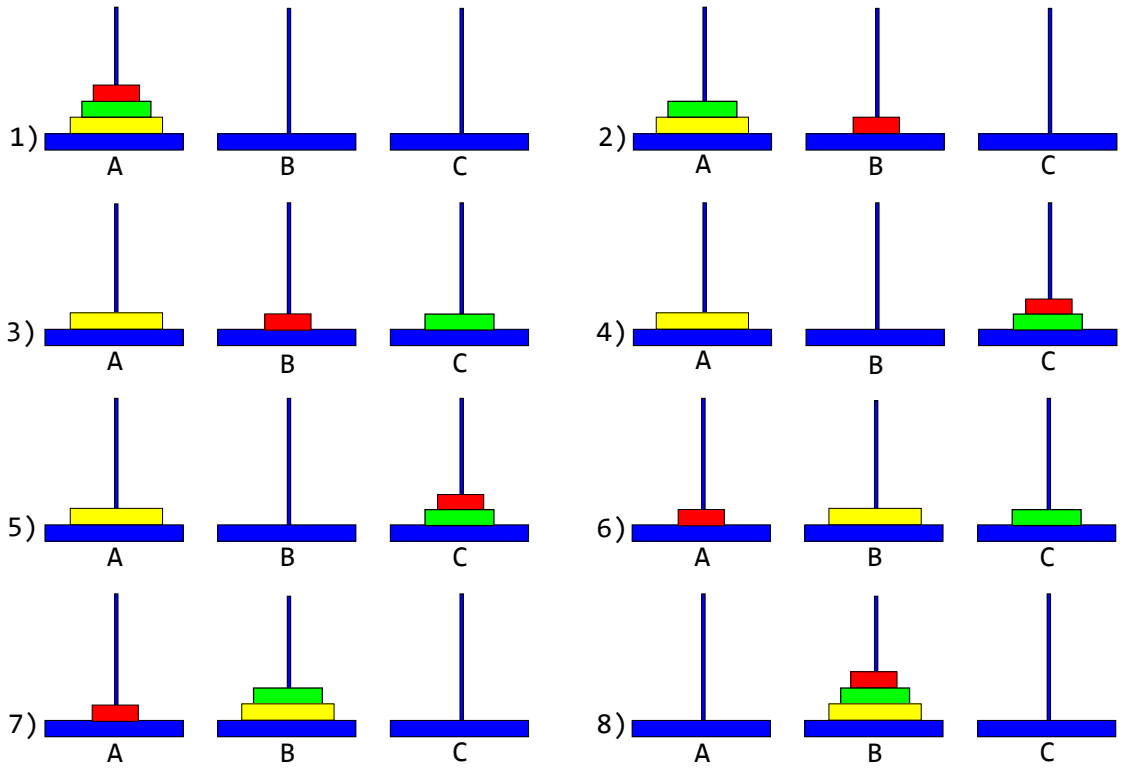
```
class Program
{
    static bool Nadjen(int[] niz, int a, int b, int x)
    {
        if (b < a)
        {
            return false;
        }
        else
        {
            int s = (a + b) / 2;
            if (niz[s] == x)
            {
                return true;
            }
            else if (niz[s] > x)
            {
                return Nadjen(niz, a, s - 1, x);
            }
            else
            {
                return Nadjen(niz, s + 1, b, x);
```

[1] Najveći mogući broj poteza je $\lceil \log_2(n+1) \rceil$, ali to nam nije previše bitno...

```
            }
        }
    }

    static void Main(string[] args)
    {
        int[] niz = { 1, 21, 13, 4, 8, 7, -3, 12, 5, 4, 67, 0 };
        int x = 12;
        Array.Sort(niz);
        Console.WriteLine(Nadjen(niz, 0, niz.Length - 1, x));
    }
}
```

Zadatak po imenu *Hanojska kula* sastavio je 1883. godine francuski matematičar Edvard Lukas [10]. Postoje tri stuba, na prvi stub je redom naslagano n diskova različite veličine, od najvećeg ka najmanjem, dok na ostalim stubovima nema diskova. Potrebno je sve diskove prebaciti sa prvog na drugi stub, poštujući sledeća dva pravila – diskovi se prebacuju jedan po jedan, i nije dozvoljeno staviti veći disk iznad manjeg[2]. Na slici 3, za $n = 3$, prikazan je postupak prebacivanja diskova sa prvog stuba (označenog sa A) na drugi stub (označen sa B), koristeći treći stub C kao pomoćni.



Slika 3. Primer Hanojske kule sa 3 diska

[2] *Zadatak je inspirisan legendom, koja kaže da je u hindu hramu Kaši Višvanat postojala prostorija sa tri stuba, i na levom stubu 64 zlatna diska. Po drevnom proročanstvu, sveštenici su imali zadatak da (prateći opisana pravila) prebace sve diskove sa prvog stuba na drugi, a nakon izvršenja ovog zadatka nastupio bi kraj sveta,* ☹.

Primetimo da se zadatak može opisati rekurzivno – za $n > 1$, prebacivanje n diskova sa prvog na drugi stub može se izvršiti tako što se najpre $n - 1$ disk prebaci sa prvog na treći stub, zatim se poslednji disk prebaci sa prvog na drugi stub, a na kraju se $n - 1$ disk prebaci sa trećeg na drugi stub. Naravno, ako je $n = 1$, zadatak se trivijalno izvršava prebacivanjem samo jednog diska.

Imajući ovaj rekurzivni opis u vidu, pišemo metod kome, pored broja diskova, redom prosleđujemo i redne brojeve stubova – stub sa kog se prebacuje, stub na koji se prebacuje, i preostali stub.

Naredni program za dat prirodni broj n ispisuje prebacivanja diskova u rešenju zadatka Hanojske kule sa n diskova.

```
class Program
{
    static void Hanoi(int n, int a, int b, int c)
    {
        if (n == 1)
        {
            Console.WriteLine("{0} -> {1}", a, b);
        }
        else
        {
            Hanoi(n - 1, a, c, b);
            Console.WriteLine("{0} -> {1}", a, b);
            Hanoi(n - 1, c, b, a);
        }
    }

    static void Main(string[] args)
    {
        int n = 4;
        Hanoi(n, 1, 2, 3);
    }
}
```

Kada izvršimo program, dobijamo sledeći ispis.

```
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
1 -> 2
3 -> 2
3 -> 1
2 -> 1
3 -> 2
1 -> 3
1 -> 2
3 -> 2
```

U programiranju povremeno postoji potreba da dve ili više rekurzivnih funkcija tokom svog rada pozivaju jedna drugu, što se naziva *uzajamna rekurzija.* Kao primer navodimo igru koju igraju dva igrača – Alisa i Bob. Oni sa stola, na kome na početku ima n kamenčića, naizmenično uklanjaju 1 ili 2 kamenčića. Pobeđuje onaj igrač koji ukloni poslednji kamenčić sa stola. Pretpostavimo da igrači igraju tako da Alisa u svakom potezu ukloni tačno 1 kamenčić, a Bob ukloni 1 ako je na stolu neparan broj kamenčića, a u suprotnom ukloni 2 kamenčića. Pitanje koje se postavlja je ko pobeđuje u ovoj igri ako je dat prirodan broj n i ako se zna da Alisa prva igra. Naredni program za dat prirodni broj n ispisuje pobednika u ovoj igri.

```
class Program
{
    static void AlisaIgra(int n)
    {
        if (n == 0)
        {
            Console.WriteLine("Bob pobedjuje");
        }
        else
        {
            BobIgra(n - 1);
        }
    }

    static void BobIgra(int n)
    {
        if (n == 0)
        {
            Console.WriteLine("Alisa pobedjuje");
        }
        else
        {
            if (n % 2 == 0)
            {
                AlisaIgra(n - 2);
            }
            else
            {
                AlisaIgra(n - 1);
            }
        }
    }
    static void Main(string[] args)
    {
        int n = 8;
        AlisaIgra(n);
    }
}
```

Kada izvršimo program, ispisuje se informacija da Bob pobeđuje, što ne treba da čudi jer se teorijskom analizom toka igre može utvrditi da Bob pobeđuje za svako $n \geq 2$.

6 Kombinatorni problemi

Ukratko govoreći, kombinatorika je grana matematike koja se bavi diskretnim strukturama i prebrajanjem. Samim tim, problemi iz ove oblasti pogodni su za rešavanje upotrebom računara, posebno ako su konačne prirode. Ovde ćemo predstaviti načine da u programu obradimo neke od standardnih kombinatornih struktura, kao što su permutacije, kombinacije, varijacije, partitivni skup, i sl. Više o kombinatorici i diskretnoj matematici može se naći u [11].

6.1 Partitivni skup

U narednom programu iskoristićemo rekurziju da „prođemo“ kroz partitivni skup (skup svih podskupova) skupa $\{0, 1, \ldots, n-1\}$, gde je n dat prirodan broj.

Podskup skupa $\{0, 1, \ldots, n-1\}$ predstavićemo preko logičkog niza dužine n, tako da element niza na poziciji i ima vrednost `true` ako i samo ako je i element podskupa. Ključni metod koji preko rekurzivnih poziva generiše sve podskupove zvaće se `PartitivniSkup`. Ispis skupa obavićemo metodom `IspisiSkup`, koji ispisuje skup u matematičkom zapisu – elementi su nabrojani unutar vitičastih zagrada i odvojeni su zarezima.

```
class Program
{
    static void IspisiSkup(bool[] jeElement)
    {
        bool pocetak = true;
        Console.Write("{");
        for (int i = 0; i < jeElement.Length; i++)
        {
            if (jeElement[i])
            {
                if (pocetak)
                {
                    pocetak = false;
                }
                else
                {
                    Console.Write(", ");
                }
                Console.Write("{0}", i);
            }
        }
        Console.WriteLine("}");
    }

    static void PartitivniSkup(bool[] jeElement, int n)
    {
        if (n == 0)
        {
            IspisiSkup(jeElement);
        }
        else
        {
```

```
            jeElement[n - 1] = true;
            PartitivniSkup(jeElement, n - 1);
            jeElement[n - 1] = false;
            PartitivniSkup(jeElement, n - 1);
        }
    }

    static void Main(string[] args)
    {
        int n = 3;
        bool[] jeElement = new bool[n];
        PartitivniSkup(jeElement, n);
    }
}
```

Prilikom izvršavanja, dobijamo sledeći ispis.

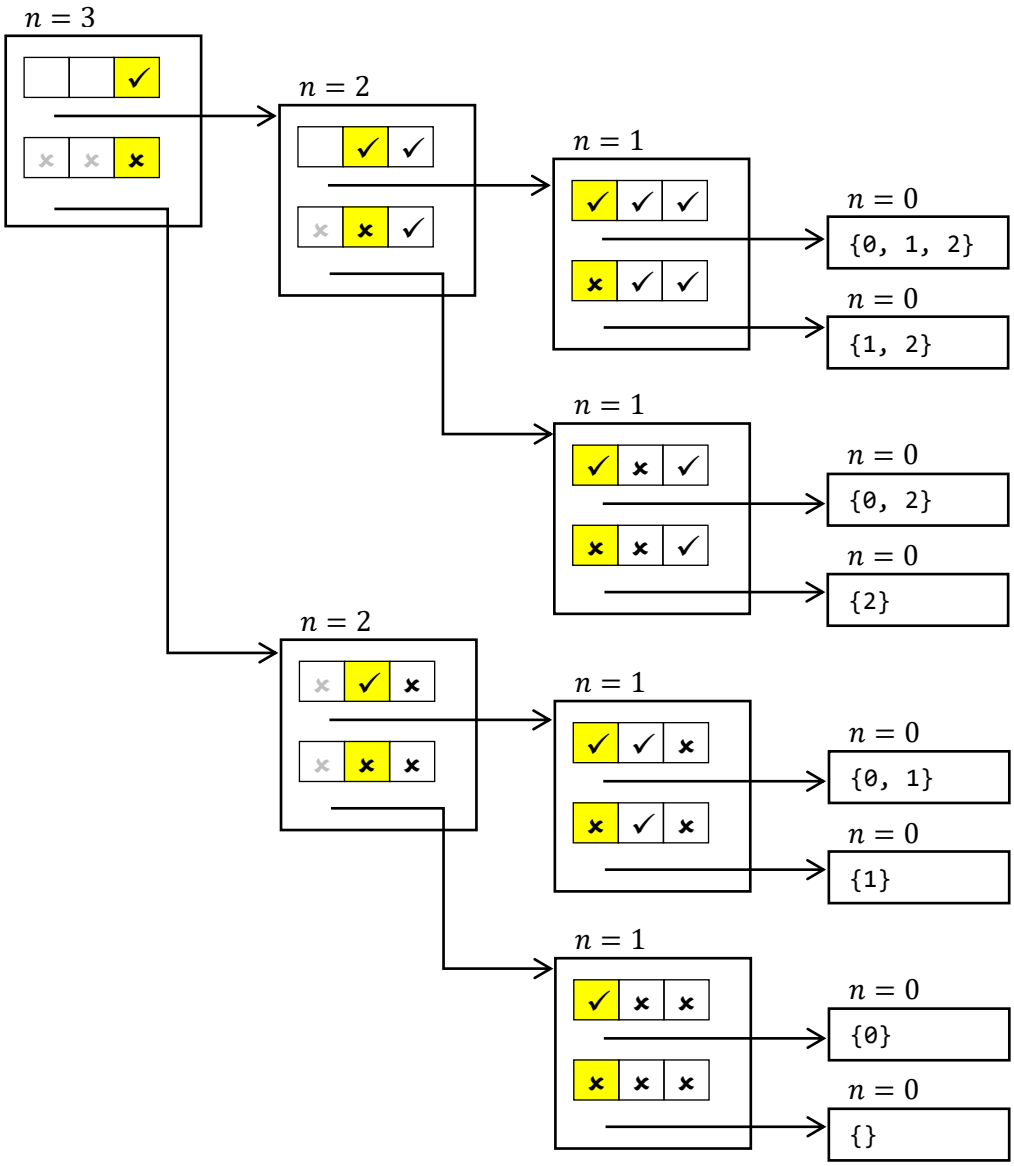
```
{0, 1, 2}
{1, 2}
{0, 2}
{2}
{0, 1}
{1}
{0}
{}
```

Bitno je napomenuti da se niz prosleđen kao parametar metoda ne kopira (za razliku od promenljive), već se svaka promena na nizu unutar metoda odražava i na prosleđeni niz (koji je deklarisan izvan metoda).

Imajući to u vidu, u celom prethodnom programu postoji samo jedan niz – logički niz `jeElement` koji je deklarisan u metodu `Main`. Kod svih poziva metoda `PartitivniSkup` i `IspisiSkup` koji imaju niz kao parametar, prosleđen je taj **isti** niz i sve dodele vrednosti elementima vrše se na njemu.

Šematski prikaz strukture rekurzivnih poziva prilikom izvršavanja prethodnog programa je prikazan na slici 4. Iznad svakog poziva metoda navedena je vrednost parametra n sa kojom je metod pozvan, a strelice prikazuju redosled izvršavanja. Iako u programu (i svim pozivima metoda) postoji samo jedan niz, zbog lakšeg praćenja na svakom mestu na kome se dodeljuje vrednost nekom njegovom elementu naveden je ceo niz vrednosti koje on sadrži u tom momentu, s tim da je žutim označena vrednost koja je upravo dodeljena. Pritom, znak „✓“ označava vrednost true, a znak „✗“ vrednost false.

Niz povremeno sadrži i „zaostale“ vrednosti (koje se nalaze levo od elementa čija se vrednost trenutno upisuje) – te vrednosti su nebitne jer će tokom izvršavanja programa preko njih biti upisane nove vrednosti, i one su označene sivim znacima.



Slika 4. Struktura rekurzivnih poziva

Prethodni program bavio se podskupovima skupa $\{0, 1, \ldots, n-1\}$. No, uz male modifikacije možemo dobiti i prolazak kroz sve podskupove *proizvoljnog* skupa. Sledi program koji za dati niz (čiji su svi elementi različiti) ispisuje sve podskupove skupa koji sadrži elemente tog niza.

Kao što ćemo videti, pomenuti niz ćemo nepromenjen proslediti kao dodatni parametar svim pozivima metoda `PartitivniSkup` i `IspisiSkup`, a ključna razlika je to što će se prilikom ispisa umesto indeksa `i` ispisivati element datog niza sa indeksom `i`.

```
class Program
{
    static void IspisiSkup(bool[] jeElement, int[] skup)
    {
        bool pocetak = true;
        Console.Write("{");
        for (int i = 0; i < jeElement.Length; i++)
        {
```

```
            if (jeElement[i])
            {
                if (pocetak)
                {
                    pocetak = false;
                }
                else
                {
                    Console.Write(", ");
                }
                Console.Write("{0}", skup[i]);
            }
        }
        Console.WriteLine("}");
    }

    static void PartitivniSkup(bool[] jeElement, int[] skup, int n)
    {
        if (n == 0)
        {
            IspisiSkup(jeElement, skup);
        }
        else
        {
            jeElement[n - 1] = true;
            PartitivniSkup(jeElement, skup, n - 1);
            jeElement[n - 1] = false;
            PartitivniSkup(jeElement, skup, n - 1);
        }
    }

    static void Main(string[] args)
    {
        int[] skup = { 2, 6, 7, 9 };
        int n = skup.Length;
        bool[] jeElement = new bool[n];
        PartitivniSkup(jeElement, skup, n);
    }
}
```

Prilikom izvršavanja, dolazi do sledećeg ispisa.

```
{2, 6, 7, 9}
{6, 7, 9}
{2, 7, 9}
{7, 9}
{2, 6, 9}
{6, 9}
{2, 9}
{9}
{2, 6, 7}
{6, 7}
{2, 7}
{7}
```

```
{2, 6}
{6}
{2}
{}
```

Metodi koje smo kreirali u prvom programu omogućavaju nam da prođemo kroz sve podskupove skupa $\{0, 1, \ldots, n-1\}$, kao i da ispišemo svaki od njih. U slučaju da želimo da ispišemo samo *neke* od podskupova, to možemo uraditi modifikacijom ovog kôda.

Konkretno, ako za dat prirodni broj n želimo da ispišemo sve podskupove skupa $\{0, 1, \ldots, n-1\}$ koji ne sadrže susedne (uzastopne) brojeve, u prethodnom programu treba da dodamo metod koji proverava ovaj uslov (metod ćemo nazvati `DobarSkup`), uz modifikaciju IF-bloka u već napisanom metodu `PartitivniSkup`. Metod `IspisiSkup` možemo da iskoristimo u identičnoj formi, kompletan program sledi.

```
class Program
{
    static void IspisiSkup(bool[] jeElement)
    {
        bool pocetak = true;
        Console.Write("{");
        for (int i = 0; i < jeElement.Length; i++)
        {
            if (jeElement[i])
            {
                if (pocetak)
                {
                    pocetak = false;
                }
                else
                {
                    Console.Write(", ");
                }
                Console.Write("{0}", i);
            }
        }
        Console.WriteLine("}");
    }

    static bool DobarSkup(bool[] jeElement)
    {
        bool neSadrziSusedne = true;
        for (int i = 1; i < jeElement.Length; i++)
        {
            if (jeElement[i - 1] && jeElement[i])
            {
                neSadrziSusedne = false;
            }
        }
        return neSadrziSusedne;
    }

    static void PartitivniSkup(bool[] jeElement, int n)
    {
```

```
        if (n == 0)
        {
            if (DobarSkup(jeElement))
            {
                IspisiSkup(jeElement);
            }
        }
        else
        {
            jeElement[n - 1] = true;
            PartitivniSkup(jeElement, n - 1);
            jeElement[n - 1] = false;
            PartitivniSkup(jeElement, n - 1);
        }
    }

    static void Main(string[] args)
    {
        int n = 5;
        bool[] jeElement = new bool[n];
        PartitivniSkup(jeElement, n);
    }
}
```

Kada se program izvrši, dolazi do sledećeg ispisa.

```
{0, 2, 4}
{2, 4}
{1, 4}
{0, 4}
{4}
{1, 3}
{0, 3}
{3}
{0, 2}
{2}
{1}
{0}
{}
```

Sledi program koji za dati niz (čiji su svi elementi različiti) ispisuje sve podskupove skupa koji sadrži elemente niza, takve da je zbir elemenata podskupa jednak datom celom broju a. Kao osnovu koristimo program koji smo već napisali, a koji prolazi kroz sve podskupove datog skupa.

```
class Program
{
    static void IspisiSkup(bool[] jeElement, int[] skup)
    {
        bool pocetak = true;
        Console.Write("{");
        for (int i = 0; i < jeElement.Length; i++)
        {
            if (jeElement[i])
```

```
        {
            if (pocetak)
            {
                pocetak = false;
            }
            else
            {
                Console.Write(", ");
            }
            Console.Write("{0}", skup[i]);
        }
    }
    Console.WriteLine("}");
}

static bool DobarZbir(bool[] jeElement, int[] skup, int a)
{
    int zbir = 0;
    for (int i = 0; i < jeElement.Length; i++)
    {
        if (jeElement[i])
        {
            zbir += skup[i];
        }
    }
    if (zbir == a)
    {
        return true;
    }
    else
    {
        return false;
    }
}

static void PartitivniSkup(bool[] jeElement, int[] skup, int n, int a)
{
    if (n == 0)
    {
        if (DobarZbir(jeElement, skup, a))
        {
            IspisiSkup(jeElement, skup);
        }
    }
    else
    {
        jeElement[n - 1] = true;
        PartitivniSkup(jeElement, skup, n - 1, a);
        jeElement[n - 1] = false;
        PartitivniSkup(jeElement, skup, n - 1, a);
    }
}
```

```
    static void Main(string[] args)
    {
        int[] skup = { -1, -2, -3, -7, 1, 2, 6, 9 };
        int a = 1;
        int n = skup.Length;
        bool[] jeElement = new bool[n];
        PartitivniSkup(jeElement, skup, n, a);
    }
}
```

Kada se program izvrši, dolazi do sledećeg ispisa. Vidimo da je zbir elemenata u svakom od skupova jednak jedan.

```
{-1, -3, -7, 1, 2, 9}
{-3, -7, 2, 9}
{-1, -2, -7, 2, 9}
{-2, -7, 1, 9}
{-1, -7, 9}
{-1, -7, 1, 2, 6}
{-7, 2, 6}
{-1, -2, -3, 1, 6}
{-2, -3, 6}
{-2, 1, 2}
{-1, 2}
{1}
```

6.2 Razbijanje broja u zbir

Za prirodne brojeve s i k, pod *razbijanjem* broja s u zbir k nenegativnih celih brojeva podrazumevamo uređenu k-torku brojeva $(x_0, x_1, \dots, x_{k-1})$ tako da važi $x_0 + x_1 + \cdots + x_{k-1} = s$.

Za pisanje programa koji prolazi kroz sva ovakva razbijanja koristimo rekurzivni opis ove strukture. Naime, ako je $k > 1$, za x_{k-1}, tj. poslednji broj u razbijanju, možemo uzeti bilo koji od brojeva $0, 1, 2, \dots, s$, dok preostali brojevi predstavljaju razbijanje broja $s - x_{k-1}$ u zbir $k - 1$ brojeva. Ako je $k = 1$, jasno je da imamo samo jednu mogućnost da kreiramo razbijanje.

```
class Program
{
    static void IspisiSabirke(int[] sabirci)
    {
        Console.Write(sabirci[0]);
        for (int i = 1; i < sabirci.Length; i++)
        {
            Console.Write(" + {0}", sabirci[i]);
        }
        Console.WriteLine();
    }

    static void RazbijUZbir(int[] sabirci, int s, int k)
    {
        if (k == 1)
        {
```

```
            sabirci[0] = s;
            IspisiSabirke(sabirci);
        }
        else
        {
            for (int i = 0; i <= s; i++)
            {
                sabirci[k - 1] = i;
                RazbijUZbir(sabirci, s - i, k - 1);
            }
        }
    }

    static void Main(string[] args)
    {
        int k = 3;
        int s = 4;
        int[] sabirci = new int[k];
        RazbijUZbir(sabirci, s, k);
    }
}
```

Kada pokrenemo program, dobijamo sledeći ispis.

```
4 + 0 + 0
3 + 1 + 0
2 + 2 + 0
1 + 3 + 0
0 + 4 + 0
3 + 0 + 1
2 + 1 + 1
1 + 2 + 1
0 + 3 + 1
2 + 0 + 2
1 + 1 + 2
0 + 2 + 2
1 + 0 + 3
0 + 1 + 3
0 + 0 + 4
```

Ako želimo da *prebrojimo* sva razbijanja broja s u zbir k nenegativnih celih brojeva, uvodimo nekoliko modifikacija u prethodni program.

Metod `RazbijUZbir` će vraćati vrednost celobrojnog tipa (i to će biti broj načina da se broj s razbije u zbir k nenegativnih celih brojeva). Iz IF-bloka metoda vratićemo 1 (jer smo u tom slučaju našli jedno razbijanje u zbir), dok ćemo iz ELSE-bloka vratiti *zbir* svih vraćenih vrednosti rekurzivnih poziva unutar tog bloka. Na taj način, vrednost vraćena iz inicijalnog poziva metoda `RazbijUZbir` biće jednaka zbiru, pa samim tim i broju, svih vraćenih jedinica, a taj broj je jednak broju traženih razbijanja u zbir.

```
class Program
{
    static int RazbijUZbir(int[] sabirci, int s, int k)
    {
        if (k == 1)
        {
            sabirci[0] = s;
            return 1;
        }
        else
        {
            int zbir = 0;
            for (int i = 0; i <= s; i++)
            {
                sabirci[k - 1] = i;
                zbir += RazbijUZbir(sabirci, s - i, k - 1);
            }
            return zbir;
        }
    }

    static void Main(string[] args)
    {
        int k = 3;
        int s = 4;
        int[] sabirci = new int[k];
        Console.WriteLine(RazbijUZbir(sabirci, s, k));
    }
}
```

Prilikom izvršavanja programa, ispisuje se broj 15.

U opštem slučaju, ako koristimo rekurziju za prolazak kroz predstavnike neke kombinatorne strukture, ali umesto ispisa želimo da ih *prebrojimo*, generički pristup koji koristimo je sledeći. Najpre, rekurzivni metod modifikujemo tako da vraća celobrojnu vrednost. Dalje, kad god naiđemo na predstavnika strukture koju prebrajamo, vraćamo vrednost 1. Ako unutar metoda imamo rekurzivne pozive, tada vraćamo zbir svih vraćenih vrednosti rekurzivnih poziva unutar tog metoda. Na taj način vraćena vrednost inicijalno pozvanog metoda biće jednaka zbiru vraćenih jedinica, a samim tim i broju takvih jedinica, tj. broju traženih kombinatornih objekata.

Ako želimo da brojimo samo one predstavnike strukture koji zadovoljavaju određeni uslov, pristup je sličan – svaki put kad naiđemo na predstavnika strukture vršimo proveru uslova i u slučaju pozitivnog ishoda vraćamo 1, a inače vraćamo 0.

U nastavku dajemo program koji ispisuje broj razbijanja broja s u zbir k nenegativnih celih brojeva koja ne sadrže nijednu nulu (tj. svi brojevi u razbijanju su prirodni brojevi).

```
class Program
{
    static bool DobroRazbijanje(int[] sabirci)
    {
        bool nemaNulu = true;
        for (int i = 0; i < sabirci.Length; i++)
```

```
            {
                if (sabirci[i] == 0)
                {
                    nemaNulu = false;
                }
            }
            return nemaNulu;
        }

        static int RazbijUZbir(int[] sabirci, int s, int k)
        {
            if (k == 1)
            {
                sabirci[0] = s;
                if (DobroRazbijanje(sabirci))
                {
                    return 1;
                }
                else
                {
                    return 0;
                }
            }
            else
            {
                int zbir = 0;
                for (int i = 0; i <= s; i++)
                {
                    sabirci[k - 1] = i;
                    zbir += RazbijUZbir(sabirci, s - i, k - 1);
                }
                return zbir;
            }
        }

        static void Main(string[] args)
        {
            int k = 3;
            int s = 4;
            int[] sabirci = new int[k];
            Console.WriteLine(RazbijUZbir(sabirci, s, k));
        }
    }
```

Prilikom izvršavanja programa, ispisuje se broj tri.

Napominjemo da se prolazak kroz sva razbijanja broja s u zbir k *prirodnih* brojeva može izvršiti i prolaskom kroz razbijanja broja $s - k$ u zbir k nenegativnih celih brojeva, uz uvećavanje svakog od dobijenih brojeva za jedan.

Prelazimo na srodan problem – razbijanje prirodnog broja s u zbir prirodnih brojeva, ali ovaj put bez ograničenja na broj brojeva u razbijanju. Sledeći program prolazi kroz sva takva razbijanja datog

prirodnog broja. Pošto broj sabiraka nije uvek isti, za smeštanje sabiraka koristićemo dovoljno dugačak niz čiju trenutnu dužinu čuvamo u posebnoj promenljivoj.

```
class Program
{
    static void IspisiSabirke(int[] sabirci, int dSabirci)
    {
        Console.Write(sabirci[0]);
        for (int i = 1; i < dSabirci; i++)
        {
            Console.Write(" + {0}", sabirci[i]);
        }
        Console.WriteLine();
    }

    static void Razbijanje(int[] sabirci, int dSabirci, int s)
    {
        if (s == 0)
        {
            IspisiSabirke(sabirci, dSabirci);
        }
        else
        {
            for (int i = 1; i <= s; i++)
            {
                sabirci[dSabirci] = i;
                Razbijanje(sabirci, dSabirci + 1, s - i);
            }
        }
    }

    static void Main(string[] args)
    {
        int s = 4;
        int[] sabirci = new int[100];
        int dSabirci = 0;
        Razbijanje(sabirci, dSabirci, s);
    }
}
```

Prilikom izvršavanja programa, dolazi do sledećeg ispisa.

```
1 + 1 + 1 + 1
1 + 1 + 2
1 + 2 + 1
1 + 3
2 + 1 + 1
2 + 2
3 + 1
4
```

6.3 Varijacije sa ponavljanjem

Varijacija sa ponavljanjem dužine k, datog skupa sa n elemenata, S_n, je uređena k-torka elemenata iz tog skupa. Naredni program ispisuje sve varijacije sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine k.

Za prolazak kroz sve varijacije sa ponavljanjem koristimo rekurzivni opis ove strukture – za poslednji element varijacije možemo izabrati bilo koji element navedenog skupa, a ispred njega možemo staviti bilo koju varijaciju sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine $k-1$.

```
class Program
{
    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void VSP(int[] niz, int n, int k)
    {
        if (k == 0)
        {
            IspisiNiz(niz);
        }
        else
        {
            for (int i = 0; i < n; i++)
            {
                niz[k - 1] = i;
                VSP(niz, n, k - 1);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 2;
        int k = 4;
        int[] niz = new int[k];
        VSP(niz, n, k);
    }
}
```

Kada se program izvrši, dobija se sledeći ispis.

```
0 0 0 0
1 0 0 0
0 1 0 0
1 1 0 0
0 0 1 0
```

```
1 0 1 0
0 1 1 0
1 1 1 0
0 0 0 1
1 0 0 1
0 1 0 1
1 1 0 1
0 0 1 1
1 0 1 1
0 1 1 1
1 1 1 1
```

U narednom programu, ispisuju se sve varijacije sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine k u kojima se nijedan element ne pojavljuje tačno jednom. Ovaj uslov proverićemo u posebnom metodu `DobraVarijacija`.

```
class Program
{
    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static bool DobraVarijacija(int[] niz, int n)
    {
        int[] brojac = new int[n];
        for (int i = 0; i < n; i++)
        {
            brojac[i] = 0;
        }
        for (int j = 0; j < niz.Length; j++)
        {
            brojac[niz[j]]++;
        }
        bool test = true;
        for (int i = 0; i < n; i++)
        {
            if (brojac[i] == 1)
            {
                test = false;
            }
        }
        return test;
    }

    static void VSP(int[] niz, int n, int k)
    {
        if (k == 0)
        {
```

```
            if (DobraVarijacija(niz, n))
            {
                IspisiNiz(niz);
            }
        }
        else
        {
            for (int i = 0; i < n; i++)
            {
                niz[k - 1] = i;
                VSP(niz, n, k - 1);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 3;
        int k = 4;
        int[] niz = new int[k];
        VSP(niz, n, k);
    }
}
```

Proveru da li se u varijaciji neki element pojavljuje tačno jednom izvršavamo tako što za svaki element skupa $\{0, 1, \ldots, n-1\}$ nalazimo broj njegovih pojavljivanja u varijaciji.

Kada se program izvrši, dobija se sledeći ispis.

```
0 0 0 0
1 1 0 0
2 2 0 0
1 0 1 0
0 1 1 0
2 0 2 0
0 2 2 0
1 0 0 1
0 1 0 1
0 0 1 1
1 1 1 1
2 2 1 1
2 1 2 1
1 2 2 1
2 0 0 2
0 2 0 2
2 1 1 2
1 2 1 2
0 0 2 2
1 1 2 2
2 2 2 2
```

6.4 Permutacije

Permutacija datog skupa sa n elemenata, S_n, predstavlja bilo koju uređenu n-torku različitih elemenata iz tog skupa. U narednom programu ispisuju se sve permutacije skupa $\{0, 1, \ldots, n-1\}$.

Za prolazak kroz sve permutacije koristićemo rekurzivni opis ove strukture – za prvi element permutacije možemo uzeti bilo koji element navedenog skupa, dok posle njega možemo staviti bilo koju permutaciju svih preostalih elemenata.

U ovom slučaju program će biti nešto drugačiji od prethodnih, proces ćemo početi sa već popunjenim nizom u kome se redom nalaze svi elementi skupa – taj niz generisaćemo već u metodu `Main`.

Jedan po jedan, svaki od elemenata će biti doveden na početno mesto u nizu, zamenom mesta sa elementom sa indeksom nula. Posle svake od zamena rekurzivno ćemo generisati sve permutacije preostalih elemenata, da bismo nakon toga vratili niz u prvobitno stanje – ponovo menjajući mesto ista dva elementa niza.

Pored niza i parametra `m` koji predstavlja redni broj pozicije koju trenutno obrađujemo, svim rekurzivnim pozivima prosleđivaće se i parametar `n`, koji će u svakom pozivu biti jednak dužini permutacije. Zamena dva elementa u nizu biće vršena metodom `Zameni`.

```
class Program
{
    static void Zameni(int[] niz, int a, int b)
    {
        int pomocna = niz[a];
        niz[a] = niz[b];
        niz[b] = pomocna;
    }

    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void Per(int[] niz, int n, int m)
    {
        if (m == n)
        {
            IspisiNiz(niz);
        }
        else
        {
            for (int i = m; i < n; i++)
            {
                Zameni(niz, m, i);
                Per(niz, n, m + 1);
                Zameni(niz, m, i);
            }
```

```
        }
    }

    static void Main(string[] args)
    {
        int n = 4;
        int[] niz = new int[n];
        for (int i = 0; i < n; i++)
        {
            niz[i] = i;
        }
        Per(niz, n, 0);
    }
}
```

Prilikom izvršavanja programa dobija se sledeći ispis.

```
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0
3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

Primetimo da se metod `Per` iz prethodnog programa može koristiti i za prolazak kroz permutacije proizvoljnog skupa – samo je potrebno elemente tog skupa smestiti u `niz` u metodu `Main`.

Za datu permutaciju π, ciklus je skup $\{a_1, a_2, \dots, a_s\} \subseteq \{0,1, \dots, n-1\}$ takav da se u π na poziciji a_1 nalazi element a_2, na poziciji a_2 nalazi se a_3, i tako dalje. Konačno, na poziciji a_s nalazi se a_1. Za svaku permutaciju važi da su njeni ciklusi disjunktni, i da im je unija jednaka skupu $\{0,1, \dots, n-1\}$. Na primer, ciklusi permutacije $(2,3,0,4,1)$ su $\{0,2\}$ i $\{1,3,4\}$, a permutacija $(1,2,3,4,0)$ ima samo jedan ciklus, $\{0,1,2,3,4\}$.

Naredni program ispisuje sve permutacije skupa $\{0, 1, \dots, n-1\}$ koje imaju tačno jedan ciklus. Za proveru uslova koristićemo poseban metod, `JedanCiklus`.

```
class Program
{
    static void Zameni(int[] niz, int a, int b)
    {
        int pomocna = niz[a];
        niz[a] = niz[b];
        niz[b] = pomocna;
    }

    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static bool JedanCiklus(int[] niz, int n)
    {
        int pozicija = 0;
        int brojac = 0;

        do
        {
            pozicija = niz[pozicija];
            brojac++;
        }
        while (pozicija != 0);

        if (brojac == n)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    static void Per(int[] niz, int n, int m)
    {
        if (m == n)
        {
            if (JedanCiklus(niz, n))
            {
                IspisiNiz(niz);
            }
        }
        else
        {
            for (int i = m; i < n; i++)
            {
```

```
                Zameni(niz, m, i);
                Per(niz, n, m + 1);
                Zameni(niz, m, i);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 4;
        int[] niz = new int[n];
        for (int i = 0; i < n; i++)
        {
            niz[i] = i;
        }
        Per(niz, n, 0);
    }
}
```

Prilikom izvršavanja programa, dobija se sledeći ispis.

```
1 2 3 0
1 3 0 2
2 0 3 1
2 3 1 0
3 2 0 1
3 0 1 2
```

6.5 Varijacije bez ponavljanja

Varijacija dužina k bez ponavljanja datog skupa sa n elemenata, S_n, je bilo koja uređena k-torka različitih elemenata iz tog skupa. Naredni program ispisuje sve varijacije bez ponavljanja skupa $\{0, 1, \ldots, n-1\}$ dužine k.

Algoritam koji ćemo koristiti vrlo je sličan onom koji smo koristili za generisanje permutacija, s tim da ćemo samo prvih k elemenata niza menjati sa svim ostalima, nakon čega će ne generisana varijacija naći na prvih k pozicija niza. Za ispis će nam trebati poseban metod za ispis *prvih k elemenata* niza. U svim pozivima metoda prosleđeni parametri n i k imaće svoju inicijalnu vrednost.

```
class Program
{
    static void Zameni(int[] niz, int a, int b)
    {
        int pomocna = niz[a];
        niz[a] = niz[b];
        niz[b] = pomocna;
    }

    static void IspisiK(int[] niz, int k)
    {
        for (int i = 0; i < k; i++)
        {
```

```
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void VBP(int[] niz, int n, int k, int m)
    {
        if (m == k)
        {
            IspisiK(niz, k);
        }
        else
        {
            for (int i = m; i < n; i++)
            {
                Zameni(niz, m, i);
                VBP(niz, n, k, m + 1);
                Zameni(niz, m, i);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 4;
        int k = 2;
        int[] niz = new int[n];
        for (int i = 0; i < n; i++)
        {
            niz[i] = i;
        }
        VBP(niz, n, k, 0);
    }
}
```

Prilikom izvršavanja programa dobija se sledeći ispis.

```
0 1
0 2
0 3
1 0
1 2
1 3
2 1
2 0
2 3
3 1
3 2
3 0
```

6.6 Kombinacije sa ponavljanjem

Kombinacija sa ponavljanjem dužine k, datog skupa sa n elemenata, S_n, je bilo koji multiskup sa tačno k, ne obavezno različitih, elemenata iz tog skupa. Naredni program ispisuje sve kombinacije sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine k.

S obzirom da redosled elemenata u kombinacijama nije bitan, a mi želimo da ih smestimo u niz, opredeljujemo se za smeštanje elemenata redom – u neopadajućem redosledu.

Za prolazak kroz sve predstavnike ove strukture koristićemo njen rekurzivni opis. U slučaju da je $n > 1$, sve kombinacije sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine k generišemo tako što najpre generišemo sve kombinacije sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine $k-1$ te svima njima dodamo element $n-1$, a zatim generišemo i sve kombinacije sa ponavljanjem skupa $\{0, 1, \ldots, n-2\}$ dužine k. Ako je $n = 1$, samo prva opcija dolazi u obzir.

```
class Program
{
    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void KSP(int[] niz, int n, int k)
    {
        if (k == 0)
        {
            IspisiNiz(niz);
        }
        else
        {
            niz[k - 1] = n - 1;
            KSP(niz, n, k - 1);
            if (n > 1)
            {
                KSP(niz, n - 1, k);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 3;
        int k = 4;
        int[] niz = new int[k];
        KSP(niz, n, k);
    }
}
```

Kada se program izvrši, dobija se sledeći ispis.

```
2 2 2 2
1 2 2 2
0 2 2 2
1 1 2 2
0 1 2 2
0 0 2 2
1 1 1 2
0 1 1 2
0 0 1 2
0 0 0 2
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0
```

Naredni program ispisuje *broj* kombinacija sa ponavljanjem skupa $\{0, 1, \ldots, n-1\}$ dužine k u kojima se pojavljuje najviše 2 različita elementa. U programu ćemo koristiti već opisan pristup za brojanje u rekurziji, a za proveru uslova kreiraćemo poseban metod koji koristi činjenicu da su elementi kombinacije u neopadajućem poretku.

```
class Program
{
    static bool DobraKombinacija(int[] niz)
    {
        int brojSuseda = 1;
        for (int i = 1; i < niz.Length; i++)
        {
            if (niz[i - 1] != niz[i])
            {
                brojSuseda++;
            }
        }
        return (brojSuseda <= 2);
    }

    static int KSP(int[] niz, int n, int k)
    {
        if (k == 0)
        {
            if (DobraKombinacija(niz))
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
        else
        {
```

```
            int zbir = 0;
            niz[k - 1] = n - 1;
            zbir += KSP(niz, n, k - 1);
            if (n > 1)
            {
                zbir += KSP(niz, n - 1, k);
            }
            return zbir;
        }
    }

    static void Main(string[] args)
    {
        int n = 3;
        int k = 4;
        int[] niz = new int[k];
        Console.WriteLine(KSP(niz, n, k));
    }
}
```

Kada se program izvrši, ispisuje se broj 12.

6.7 Kombinacije bez ponavljanja

Kombinacija bez ponavljanja dužine k, datog skupa sa n elemenata, S_n, je bilo koji podskup tog skupa sa tačno k elemenata. Naredni program ispisuje sve kombinacije bez ponavljanja skupa $\{0, 1, \ldots, n-1\}$ dužine k.

Kao i kod kombinacija sa ponavljanjem, s obzirom na to da redosled elemenata nije bitan, a mi želimo da ih smestimo u niz, opredeljujemo se za smeštanje elemenata redom – ovaj put u rastućem redosledu (jer u kombinacijama bez ponavljanja nema istih elemenata).

Za prolazak kroz sve predstavnike ove strukture koristićemo njen rekurzivni opis. U slučaju da je $n > k$, sve kombinacije bez ponavljanja skupa $\{0, 1, \ldots, n-1\}$ dužine k generišemo tako što najpre generišemo sve kombinacije bez ponavljanja skupa $\{0, 1, \ldots, n-2\}$ dužine $k-1$ te svima njima dodamo element $n-1$, a zatim generišemo i sve kombinacije bez ponavljanja skupa $\{0, 1, \ldots, n-2\}$ dužine k. Ako je $n = k$, samo prva opcija je moguća, jer se u tom slučaju više nijedan element ne može preskočiti.

```
class Program
{
    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
        }
        Console.WriteLine();
    }

    static void KBP(int[] niz, int n, int k)
    {
```

```
            if (k == 0)
            {
                IspisiNiz(niz);
            }
            else
            {
                niz[k - 1] = n - 1;
                KBP(niz, n - 1, k - 1);
                if (n > k)
                {
                    KBP(niz, n - 1, k);
                }
            }
        }

        static void Main(string[] args)
        {
            int n = 6;
            int k = 4;
            int[] niz = new int[k];
            KBP(niz, n, k);
        }
    }
```

Prilikom izvršavanja programa, dobijamo sledeći ispis.

```
2 3 4 5
1 3 4 5
0 3 4 5
1 2 4 5
0 2 4 5
0 1 4 5
1 2 3 5
0 2 3 5
0 1 3 5
0 1 2 5
1 2 3 4
0 2 3 4
0 1 3 4
0 1 2 4
0 1 2 3
```

U narednom programu ispisuju se sve kombinacije bez ponavljanja skupa $\{0, 1, \ldots, n-1\}$ dužine k u kojima se nijedna dva elementa ne razlikuju za tačno tri. Ovaj uslov proveravaćemo u posebnom metodu.

```
class Program
{
    static void IspisiNiz(int[] niz)
    {
        for (int i = 0; i < niz.Length; i++)
        {
            Console.Write("{0} ", niz[i]);
```

```
        }
        Console.WriteLine();
    }

    static bool DobraKombinacija(int[] niz)
    {
        bool nemaRazlikeTri = true;
        for (int i = 0; i < niz.Length - 1; i++ )
        {
            for (int j = i + 1; j < niz.Length; j++)
            {
                if (niz[j] - niz[i] == 3)
                {
                    nemaRazlikeTri = false;
                }
            }
        }
        return nemaRazlikeTri;
    }

    static void KBP(int[] niz, int n, int k)
    {
        if (k == 0)
        {
            if (DobraKombinacija(niz))
            {
                IspisiNiz(niz);
            }
        }
        else
        {
            niz[k - 1] = n - 1;
            KBP(niz, n - 1, k - 1);
            if (n > k)
            {
                KBP(niz, n - 1, k);
            }
        }
    }

    static void Main(string[] args)
    {
        int n = 8;
        int k = 4;
        int[] niz = new int[k];
        KBP(niz, n, k);
    }
}
```

Kada se program izvrši, dobija se sledeći ispis.

```
1 5 6 7
0 5 6 7
1 2 6 7
```

```
0 2 6 7
0 1 6 7
1 3 5 7
0 1 5 7
1 2 3 7
0 1 2 7
0 4 5 6
0 1 5 6
0 2 4 6
0 1 2 6
```

7 Statističke ocene

Statistika je oblast koja se, uopšteno govoreći, bavi tumačenjem podataka, a mi ćemo se upoznati sa jednom od bazičnih metoda – *tačkastom ocenom verovatnoće*. Naime, ako u okviru eksperimenta posmatramo događaj koji se odigrava sa verovatnoćom koja nam nije poznata, tu verovatnoću možemo da odredimo približno ponavljajući eksperiment više puta. Bez namere da ulazimo dublje u matematičku analizu problema, reći ćemo još samo da što više puta ponovimo eksperiment, veće su šanse da naša procena bude „blizu“ stvarne vrednosti. Više o ovoj metodi, a i mnogim drugim statističkim metodama ocene, može se naći u knjizi [12].

Na primer, bacamo kockicu za igru (koja na stranicama ima ispisane brojeve od jedan do šest) i zanima nas koja je verovatnoća da broj na kockici bude deljiv sa tri. Ako bacimo kockicu sto puta, i u 36 bacanja broj na kockici bude deljiv sa tri, procenjujemo pomenutu verovatnoću sa $36/100 = 0{,}36$. Inače, (tačna) verovatnoća ovog događaja je $1/3$.

Ako za procenu koristimo računar, eksperiment možemo da ponovimo veći broj puta, što daje bolju ocenu. Da bismo simulirali eksperimente sa slučajnim ishodom u programskom jeziku C# trebaju nam „slučajno“ izabrani brojevi, a njih dobijamo pomoću klase `Random`. Novi generator slučajnih brojeva deklarišemo na sledeći način.

```
Random <ime> = new Random();
```

Nakon toga, pomoću `<ime>.Next(a, b)` dobijamo slučajno izabran ceo broj iz poluotvorenog intervala $[a, b)$, tj. iz skupa $\{a, a+1, a+2, \ldots, b-2, b-1\}$. Pritom, svaki broj ima istu verovatnoću da bude izabran.

U narednom programu ispisuje se deset slučajno izabranih brojeva iz skupa $\{0, 1\}$.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        for (int i = 0; i < 10; i++)
        {
            int broj = slucajan.Next(0, 2);
            Console.WriteLine(broj);
        }
    }
}
```

Sledi primer ispisa koji se dobija prilikom izvršavanja ovog programa.

```
0
1
1
0
1
1
0
```

```
1
1
1
```

Generisanje slučajne vrednosti iz prethodnog programa možemo posmatrati i kao simulaciju bacanja novčića, gde se 0 interpretira kao pismo, a 1 kao glava. Ako želimo da simuliramo bacanje kockice za igru, treba da izaberemo slučajan broj iz skupa $\{1,2,\ldots,6\}$. U narednom programu simuliramo 8 bacanja kockice.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        for (int i = 0; i < 8; i++)
        {
            Console.WriteLine(slucajan.Next(1, 7));
        }
    }
}
```

Sledi primer ispisa koji se dobija prilikom izvršavanja ovog programa.

```
1
2
6
1
4
5
5
3
```

Ako želimo slučajno izabran realan broj iz intervala $(0,1)$, koristimo `NextDouble()` metod, što je ilustrovano sledećim programom.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        for (int i = 0; i < 10; i++)
        {
            double realan = slucajan.NextDouble();
            Console.WriteLine(realan);
        }
    }
}
```

Sledi primer ispisa koji se dobija prilikom izvršavanja ovog programa.

```
0.0358527410011053
0.462956475309542
0.729268771935845
```

```
0.875799682864826
0.442899257616559
0.772842418762316
0.572976021828584
0.490151732456941
0.486630610416937
0.134437600678968
```

U narednom programu se za dat prirodan broj n ispisuje procena verovatnoće da broj na bačenoj kockici za igru bude deljiv sa 3, ponavljajući eksperiment bacanja kocke n puta.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        int n = 100000;
        int uspesni = 0;
        for (int i = 1; i <= n; i++)
        {
            if (slucajan.Next(1,7) % 3 == 0)
            {
                uspesni++;
            }
        }
        Console.WriteLine((double)uspesni / n);
    }
}
```

Kao što smo već pomenuli, tačna verovatnoća ovog događaja je $1/3$, a pokretanjem programa dobija se ispisan broj koji je „blizak“ tačnoj verovatnoći.

U sledećem programu se za dat prirodan broj n ispisuje procena verovatnoće da brojevi na šest bačenih kockica za igru budu svi različiti, ponavljajući eksperiment n puta.

Verovatnoća ovog događaja je 0.015432 ...

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        int n = 100000;
        int[] brojac = new int[6];
        int uspesni = 0;
        for (int i = 1; i <= n; i++)
        {
            for (int j = 0; j < 6; j++)
            {
                brojac[j] = 0;
            }
            for (int j = 0; j < 6; j++)
            {
                int bacenBroj = slucajan.Next(1, 7);
                brojac[bacenBroj - 1]++;
```

```
            }
            if (brojac.Min() == 1)
            {
                uspesni++;
            }
        }
        Console.WriteLine((double)uspesni / n);
    }
}
```

Naredni program za date prirodne brojeve n i k ispisuje procenu verovatnoće da se tokom bacanja novčića k puta glava nije pojavila tri puta za redom, ponavljajući eksperiment n puta.

Za $k = 5$, tačna verovatnoća ovog događaja je 0.75.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        int n = 100000;
        int k = 5;
        int[] novcici = new int[k];
        int uspesni = 0;
        for (int i = 1; i <= n; i++)
        {
            for (int j = 0; j < k; j++)
            {
                novcici[j] = slucajan.Next(0, 2);
            }
            bool test = true;
            for (int j = 0; j <= k - 3; j++)
            {
                if (novcici[j] + novcici[j + 1] + novcici[j + 2] == 3)
                {
                    test = false;
                }
            }
            if (test)
            {
                uspesni++;
            }
        }
        Console.WriteLine((double)uspesni / n);
    }
}
```

U sledećem programu se za date prirodne brojeve n, k i l ispisuje procena verovatnoće da se tokom bacanja novčića k puta između svaka dva pisma glava pojavila bar l puta, ponavljajući eksperiment n puta.

Za $k = 4$ i $l = 2$, tačna verovatnoća ovog događaja je 0.375.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        int n = 1000000;
        int k = 4, l = 2;
        int[] novcici = new int[k];
        int uspesni = 0;
        for (int i = 1; i <= n; i++)
        {
            for (int j = 0; j < k; j++)
            {
                novcici[j] = slucajan.Next(0, 2);
            }
            bool test = true;
            int brojGlava = int.MaxValue / 2;
            for (int j = 0; j < k; j++)
            {
                if (novcici[j] == 0)
                {
                    if (brojGlava < l)
                    {
                        test = false;
                    }
                    brojGlava = 0;
                }
                else
                {
                    brojGlava++;
                }
            }
            if (test)
            {
                uspesni++;
            }
        }
        Console.WriteLine((double)uspesni / n);
    }
}
```

U prethodnom programu, želimo da inicijalna vrednost promenljive `brojGlava` bude velika (svakako veća od dva), pa smo odabrali da to bude `int.MaxValue / 2`. Na taj način, prilikom prvog nailaska na pismo, promenljiva `brojGlava` sigurno neće biti manja od dva i samim tim vrednost promenljive `test` će ostati `true`, što i želimo (jer bez obzira na broj glava *pre* prvog pisma, uslov zadatka je u tom momentu još uvek zadovoljen). Primetimo da `int.MaxValue` nije dobar izbor za inicijalnu vrednost promenljive `brojGlava`, jer bi nakon uvećavanja za jedan njena vrednost postala `int.MinValue`[3], što ne želimo.

[3] Uzastopnim uvećavanjem vrednosti promenljive tipa `int` za jedan „vrtimo se u krug“ – posle najvećeg broja tipa `int` sledi najmanji broj tipa `int`.

U ruletu se bacanjem kuglice bira jedan broj iz skupa $\{0,1,2,\ldots,36\}$, i svi brojevi imaju istu verovatnoću izbora.

U sledećem programu se za date prirodne brojeve n i k ispisuje procena verovatnoće da u k bacanja kuglice u ruletu *svi* brojevi dobijeni u neparnim bacanjima (prvom, trećem, petom,...) budu veći od *svih* brojeva dobijenih u parnim bacanjima, ponavljajući eksperiment n puta.

```
class Program
{
    static void Main(string[] args)
    {
        Random slucajan = new Random();
        int n = 2000000;
        int k = 4;
        int[] rulet = new int[k];
        int uspesni = 0;
        for (int i = 1; i <= n; i++)
        {
            for (int j = 0; j < k; j++)
            {
                rulet[j] = slucajan.Next(0, 37);
            }
            bool test = true;
            for (int j = 0; j < k; j += 2)
            {
                for (int s = 1; s < k; s += 2)
                {
                    if (rulet[j] <= rulet[s])
                    {
                        test = false;
                    }
                }
            }
            if (test)
            {
                uspesni++;
            }
        }
        Console.WriteLine((double)uspesni / n);
    }
}
```

Literatura

1. **Albahari, Joseph and Albahari, Ben.** *C# 5.0 in a Nutshell.* 5th Edition. Sebastopol : O'Reilly Media, Inc., 2012.
2. **Ferguson, Jeff, et al.** *C# Bible.* Indianapolis : Wiley, 2002.
3. **Griffiths, Ian.** *Programming C# 5.0.* Sebastopol : O'Reilly Media Inc., 2012.
4. **Liberty, Jesse and MacDonald Beijing, Brian.** *Learning C# 3.0.* Sebastopol : O'Reilly Media Inc., 2009.
5. **Nakov, Svetlin et al.** *Fundamentals of computer programming with C#.* Sofia : s.n., 2013.
6. **Microsoft.** https://msdn.microsoft.com/en-us/library. *https://msdn.microsoft.com.* [Online] Microsoft, 2017. [Cited: 10 20, 2017.] https://msdn.microsoft.com/en-us/library/system.math(v=vs.110).aspx.
7. **McMillan, Michael.** *Data structures and algorithms using C#.* New York : Cambridge University Press, 2007.
8. **Jain, Hemant.** *Problems Solving in Data Structures & Algorithms Using C#.* Bhopal : Hemant Jain, 2017.
9. **Tošić, Ratko and Vukoslavčević, Vanja.** *Elementi teorije brojeva.* Novi Sad : Alef, 1995.
10. **Graham, Ronald L., Knuth, Donald E. and Patashnik, Oren.** *Concrete mathematics (Second Edition).* s.l. : Addison-Wesley Publishing Company Inc., 1994.
11. **Mašulović, Dragan.** *Diskretna matematika za informatičare 1.* Novi Sad : Prirodno-matematički fakultet, Departman za matematiku i informatiku, 2014.
12. **Lozanov Crvenković, Zagorka.** *Statistika.* Novi Sad : Prirodno-matematički fakultet, 2012.