Programski jezik C#

Metode

Matija Lokar in Srečo Uranič

V 0.81

november 2008

Predgovor

Omenjeno gradivo predstavlja nadaljevanje gradiv, namenjenih predmetu Programiranje 1 na višješolskem študiju Informatika. Gradivo predpostavlja, da je bralec dobro seznanjen z osnovami programskega jezika C#. Pozna spremenljivke, osnovne podatkovne tipe, zna uporabljati prireditveni stavek, vejitve (pogojni stavek) in zanke. Prav tako zna delati s tipom String in tabelami.

To gradivio pokriva (statične) metode v jeziku C#. Ker rekurzija "naravno" spada v ta del, jo omenjava tukaj, čeprav formalno ni sestavni del predmeta Programiranje 1, ampak se pojavi šele pri predmetu Programiranje 2. Glede samega besedila velja tisto, kar je bilo napisano v predgovorih prejšnjih delov gradiv, torej – ne poveva vsega, določene stvari poenostaviva, veliko je zgledov ... .

Gradivo vsekakor ni dokončano in predstavlja delovno različico. V njem so zagotovo napake (upava, da čimmanj), za katere se vnaprej opravičujeva. Da bo lažje spremljati spremembe, obstaja razdelek Zgodovina sprememb, kamor bova vpisovala spremembe med eno in drugo različico. Tako bo nekomu, ki si je prenesel starejšo različico, lažje ugotoviti, kaj je bilo v novi različici spremenjeno.

Matija Lokar in Srečo Uranič

Kranj, november 2008

Zgodovina sprememb

11. 11. 2008: Različica V0.8 – prva, ki je na voljo javno
11. 11. 2008: Različica V0.81 – formalni in dejanski parametri, klic po referenci, pisanje metod s pomočjo čarovnika, manjši popravki in vaje

KAZALO

Metode (funkcije)

Vsi naši dosedanji programi so bili večinoma kratki in enostavni, ter zaradi tega pregledni. Pri zahtevnejših projektih pa postanejo programi daljši, saj zajemajo več kot en izračun, več pogojev, več zank. Če so napisani v enem samem kosu postanejo nepregledni. Poleg tega se zaporedja stavkov se pri daljših programih lahko tudi večkrat ponovijo. Vzdrževanje in popravljanje takih programov je težavno, časovno zahtevno, verjetnost za napake pa precejšnja.

Vse to so razlogi za pisanje metod. S pomočjo metod programe smiselno razbijemo na podnaloge, vsako podnalogo pa izdelamo posebej. Te manjše podnaloge imenujemo podprogrami, rešimo jih z lastnimi metodami, zaženemo pa jih v glavnem programu. Program torej razdelimo v več manjših "neodvisnih" postopkov in nato obravnavamo vsak postopek posebej. Metodam v drugih programskih jezikih rečemo tudi funkcije, procedure ali podprogrami.

Na ta način bo naš progam krajši, bolj pregleden, izognili se bomo večkratnemu ponavljanju istih stavkov, pa še popravljanje in dopolnjevanje bo enostavnejše. Bistvo metode je tudi v tem, da ko je enkrat napisana, moramo vedeti le še kako jo lahko uporabimo, pri tem pa nas večinoma ne zanima več, kako je pravzaprav sestavljena. Napisano metodo lahko uporabimo večkrat, seveda pa jo lahko uporabimo tudi v drugih programih.

Metode smo v bistvu že ves čas uporabljali, a se tega mogoče nismo zavedali. Za izpisovanje na zaslon smo uporabljali metodo `Console.WriteLine()`, pri pretvarjanju niza v število smo uporabljali metodo `int.Parse()`, pri računanju kvadratnega korena metodo `Math.Sqrt()` in številne druge. Vse te metode so torej že napisane, vse kar moramo vedeti o njih pa je, kakšen je njihov namen in kako jih uporabiti .

Seveda pa lahko metode pišemo tudi sami. Pravzaprav smo vsaj eno od metod že ves čas pisali – to je bila metoda `Main()`. Ogrodje zanjo nam je zgeneriralo že razvojno okolje, mi pa smo doslej pisali le njeno vsebino oz. kot temu rečemo strokovno – **glavo** metode nam je zgeneriralo razvojno okolje, **telo** metode pa smo napisali sami. Metoda `Main()` predstavlja izhodišče našega programa (t.i. **glavni program**) in ima zaradi tega že vnaprej točno predpisano ime, obliko in namen. Metode, ki se jih bomo naučili pisati, bodo imele prav tako specifičen namen, obliko in ime, vse te lastnosti pa bomo določili mi sami pred in med njihovim pisanjem. Držati pa se moramo pravila, da morajo biti metode, ki jih napišemo, pregledne, če se la da uporabne tudi v drugih programih, biti morajo samozadostne, prav tako jih tudi ne napišemo izrecno za izpisovanje ali branje podatkov, razen če to ni prav izrecni namen te metode.

Delitev metod

V jeziku C# metode v splošnem delimo na **statične** in **objektne**. V tem poglavju bomo obravnavali le statične metode. Zaenkrat povejmo le, da statične metode kličemo nad razredom, medtem, ko objektne nad objektom. Metode ločimo tudi po načinu dostopa na **javne** (public), **privatne** (private) in **zaščitene** (protected). Ker bomo uporabljali le javne metode, se v način dostopa ne bomo spuščali.

Definicija metode

Metodo v program vpeljemo z definicijo. Definicija metode je sestavljena iz **glave** oziroma **deklaracije** metode in **telesa** metode, napišemo pa jo v celoti pred ali pa **za** glavno metodo `Main` našega programa.
V deklaracijo zapišemo najprej **dostop** do metode (torej ali je metoda javna (*public*), privatna (*private*) ali zaščitena (*protected*)), nato **vrsto** metode s katero povemo ali je metoda statična (*static*) ali objektna. V tem

razdelku bodo vse naše metode vrste *public static*. Sledi ji **tip** rezultata metode. Tip rezultata metode je poljuben podatkovni tip (npr. *int*, *double[]*, *string*) in pomeni tip vrednosti, ki ga metoda vrača. Nato z **imenom metode** povemo kako se imenuje metoda. Velja dogovor, da se imena metod začnejo z malo črko. Metodam damo smiselna imena, ki povedo, kaj metoda dela. Ime metode je poljubno, ne smemo pa za ime metode pa uporabiti rezerviranih besed (ime metode **ne sme** biti npr. *string*, *do*, *return*, …). V imenih metod **ne sme** biti presledkov in nekaterih posebnih znakov (npr. znakov '/', ')', '.', …), prav tako pa v imenih metod niso zaželjeni šumniki. Za imenom metode zapišemo okrogle oklepaje, ki so obvezni del deklaracije. Znotraj oklepajev zapišemo **argumente metode**, ki jih metoda sprejme (ime argumentov) in kakšnega tipa so. Več argumentov med sabo ločimo z vejico. Če metoda ne sprejme nobenih argumentov, napišemo le prazne okrogle oklepaje *()*. Sledi **telo metode**, to je del, ki je znotraj bloka *{}*. V telo metode zapišemo stavke, ki izvršijo nalogo metode.

Posebne so metode, ki ne vračajo nobenega rezultata, ampak le opravijo določene delo (na primer nekaj izpišejo, narišejo sliko, pošljejo datoteko na tiskalnik, ...). Pri takih metodah kot tip rezultata navedemo tip ***void***. Metode tipa void v svojem telesu nimajo stavka return.

Javna metoda – metoda je dostopna vsem delm našega programa (do nadalnjega bodo vse metode javne)

Statična metoda (do nadanjega bodo vse metode statične – kaj pravzaprav to pomeni bo jasno šele pri poglavju o razredih in objektih)

tip metode: določa tip rezultata, ki ga metoda vrača.

```
public static   tip metode   ime_metode  (parametri metode)  //Glava metode
{                                                                          
   ... stavki                                                              
  return ...//vračanje vrednosti.                                          } Telo metode
           //Stavka return ni, če je metoda tipa void                      
}                                                                          
```

Poglejmo si nekaj primerov deklaracij metod.

Primeri:

```
public static int beri()
```

Metoda `beri` je javna (*public*), statična (*static*), vrača rezultat tipa *int* in ne sprejme nobenega argumenta (znotraj oklepajev ni argumentov).

```
public static void nekaj(string[] tabNiz)
```

Metoda `nekaj` je javna (*public*), statična (*static*), ne vrača rezultata (tip *void*) in sprejme en argument, ki se imenuje *tabNiz* in je tipa *string[]* (tabela nizov).

```
public static int max(int a, int b)
```

Metoda `max` je javna (*public*), statična (*static*), vrača rezultat tipa *int* in sprejme dva argumenta, ki sta celi števili (tip *int*), prvi argument se imenuje *a*, drugi *b*.

```
public double abs()
```

Metoda `abs` je javna (*public*), objektna (ni določila *static*), vrača rezultat tipa *double* in ne sprejme nobenega argumenta (znotraj oklepajev ni argumentov).

Parametri metod niso obvezni, saj lahko napišemo metodo, ki ne sprejme nobenega parametra, a take metode v splošnem niso preveč uporabne.

Pozdrav uporabniku

Napišimo metodo, ki prebere naše ime in nas pozdravi (npr. za ime "Janez" izpiše Pozdravljen Janez).

```
//ker metoda ne bo vrnila ničesar, je tipa void
public static void pozdravUporabniku()
{
    //preberemo podatek
    Console.Write("Vnesi ime: ");
    string ime = Console.ReadLine();
    //in ga izpisemo
    Console.WriteLine("Pozdravljen " + ime + "!");
}
```

Metodo smo napisali, potrebno pa jo še znati uporabiti, oz. jo poklicati. Metode tipa void (ki ne vračajo ničesar) uporabimo v **samostojnem stavku** tako, da napišemo njihovo ime, v oklepajih pa še parametre, če seveda metoda parametre potrebuje, sicer pa napišemo le oklepaj in zaklepaj. Zgornjo metodo bi torej poklicali (npr. kjerkoli v glavnem programu `Main` takole:

```
pozdravUporabniku();
```

Vrednost metode

Če je tip metode različen od *void*, moramo vrednost, ki jo vrne metoda, določiti s stavkom

```
return izraz;
```

Vrednost metode je vrednost izraza, ki je naveden za ključno besedo *return*. V opisu postopka je stavek *return* lahko na več mestih. Kakor hitro se eden izvede, se izvajanje metode konča z vrednostjo, kot jo določa izraz, naveden pri stavku *return*. Če metoda kaj izpisuje ali riše na ekran, to ni rezultat te metode, ampak samo stranski učinek, ki ga ima metoda. Take metode običajno ne vračajo nobenih rezultatov. V tem primeru je rezultat metode tipa *void*.

Primer:

```
public static double povprecje(double x, double y)
{
    return (x + y) / 2.0;
}
```

Metoda `povprecje` je javna, statična, vrne rezultat tipa *double* in sprejme dva argumenta, ki sta tipa *double*, prvi se imenuje *x* in drugi *y*.

Vsota lihih števil

Napišimo metodo, ki ne sprejme nobenih parametrov, izračuna in vrne pa vsoto vseh dvomestnih števil, ki so deljiva s 5!

```
public static int vsota() //glava funkcije
{
     int vsota = 0; //začetna vsota je 0
     for (int i = 1; i <= 100; i++)
     {
         if (i % 5 == 0) //če ostanek pri deljenju deljiv s 5
         vsota = vsota + i; //potem število prištejemo k vsoti
     }
    //metoda vrne rezultat: vsoto vseh dvomestnih števil, deljivih s 5
    return vsota;
}
```

Še klic metode v glavnem programu: ker metoda `vsota()` vrača rezultat (tip *int*), jo ne moremo klicati samostojno, ampak v nekem **prireditvenem stavku**, lahko pa tudi kot parameter v drugi metodi.

```
int vsota100 = vsota();//klic metode v prireditvenem stavku

Console.WriteLine(vsota());//klic metode kot parametra metode WriteLine
```

Zgornja metoda je sicer povsem legalna, ker pa nima vhodnih parametrov je uporabna le za točno določen, ozek namen. Če želimo narediti metodo res uporabno, uporabljamo parametre.

Argumenti metod - formalni in dejanski parametri

V glavi metode lahko uporabimo tudi **argumente**. Pravimo jim tudi parametri metode. Parametre, ki jih napišemo ko metodo pišemo, imenujemo **formalni** parametri. Ko pa neko metodo pokličemo, formalne parametre nadomestimo z **dejanskimi** parametri.

Iskanje večje vrednosti dveh števil

Napišimo metodo, ki dobi za parametra poljubni celi števili, vrne pa večje izmed teh dveh števil.
Ker bo metoda imela za parametra dve celi števili (tip *int*), je večje izmed teh dveh prav gotovo tudi tipa *int*. Metoda bo torej vrnila celo število, zaradi česar bomo za tip metode uporabili tip *int*.

```
public static int max(int a, int b) //metoda ima dva parametra, a in b
{
    if (a > b)  //če a večji od b metoda vrne stevilo a
      return a;
    return b;   //sicer pa metoda vrne stevilo b
}
```

Parametra *a* in *b*, ki smo ju napisali v glavi metode, sta **formalna parametra**. Ko bomo metodo poklicali, pa bomo zapisali **dejanska parametra**. Metodo lahko uporabimo tako, da v glavnem programu najprej preberemo (določimo, zgeneriramo) dve celi števili, nato pa ti dve števili uporabimo za parametra metode `max`.

```
Console.Write("Prvo število: ");
int prvo = int.Parse(Console.ReadLine());
Console.Write("Drugo število: ");
int drugo = int.Parse(Console.ReadLine());

int vecje = max(prvo, drugo); //parametra prvo in drugo sta dejanska
                              //parametra
Console.WriteLine("Večje od vnesenih dveh števil je : " + vecje);
```

Funkcijo `max` bi seveda lahko poklicali tudi takole:

```
Console.WriteLine("Večje od vnesenih dveh števil je : " + max(prvo,drugo));
```

Pri klicu metode `max` je parameter *a* dobil vrednost spremenljivke *prvo*, parameter *b* pa vrednost spremenljivke *drugo*. Formalna parametra *a* in *b*, smo torej pri klicu metode nadomestili z dejanskima parametroma *prvo* in *drugo*. Namesto spremenljivk bi seveda pri klicu metode lahko uporabili tudi poljubni dve števili, ali pa številska izraza, npr.:

```
Console.WriteLine("Večje izmed števil 6 in 11 je : " + max(6,11);

Console.WriteLine("Večje od vnesenih izrazov je : " + max(5+2*3,41-3*2));
```

Metodo `max` pa lahko pa pokličemo tudi takole:

```
Random naklj = new Random();
int prvo=naklj.Next(100);
int drugo=naklj.Next(100);
int tretje=naklj.Next(100);
Console.WriteLine("Največje izmed treh naključnih števil je: "+
                  max(prvo, max(drugo, tretje)));
```

Zgledi

Trikotnik

Napišimo metodo, dobi za parameter polubno celo število n in ki na zaslon nariše trikotnik, sestavljen iz samih zvezdic. Velikost trikotnika določa nenegativno celo število n, ki je podano kot argument te metode.

```
Primer:  n=0  *     n=1  *     n=2  *     n=3  *
                    ***        ***        ***
                              *****      *****
                                        *******
```

```
//Metoda
static void trikotnik(int n)
{
    int i, j;
    for( i=0; i<n+1; i++ )
    {
       for( j=n-i; j>0; j-- )      //Najprej ustrezno število presledkov
           Console.Write(" ");
       for( j=0; j<(2*i+1); j++ ) //Rišemo zvezdice
           Console.Write("*");
       Console.WriteLine();        //Skok v novo vrstico
    }
}
//Glavi program
static void Main(string[] args)
{
    Console.Write("Velikost trikotnika: ");//Velikost trikotnika preberemo
    int velikost = Convert.ToInt32(Console.ReadLine());
    trikotnik(velikost); //klic metode
}
```

Število znakov v stavku

Napišimo metodo, ki dobi dva parametra: poljuben STAVEK in poljuben ZNAK. Metoda naj ugotovi in vrne kolikokrat se v stavku pojavi izbrani znak.

```
//Metoda
static int kolikokrat(string stavek, char znak)
{
    int skupaj = 0; //Začetno število znakov je 0
    for( int i=0; i<stavek.Length; i++ )
    {
       if (stavek[i] == znak) //tekoči znak primerjamo z našim znakom
           skupaj++;
    }
    return skupaj; //Metoda vrne skupno število najdenih znakov
}
//Glavni program
static void Main(string[] args)
{
    Console.Write("Vnesi poljuben stavek: ");
    string stavek = Console.ReadLine();
    Console.Write("Vnesi znak, ki te zanima: ");
    char znak = Convert.ToChar(Console.Read());
    Console.WriteLine("V stavku je " + kolikokrat(stavek, znak)+" znakov
                      "+znak+".");
}
```

Število Pi

Število PI lahko izračunamo tudi kot vsoto vrste 4 - 4/3 + 4/5 - 4/7 + 4/9 - ... Napiši metodo, ki ugotovi in vrne, koliko členov tega zaporedja moramo sešteti, da se bo tako dobljena vsota ujemala s konstanto Math.PI do npr. vključno devete decimalke. Metoda naj ima za parameter število ujemajočih se decimalk. Rešitev za npr. 9 členov: 1096634169

```
//Metoda
static long stClenov(int clenov)
{
    double pi=4;   //definiramo in inicializiramo začetno vrednost za pi
    double clen;   //tekoči člen zaporedja
    long i=1;      //definicija in inicializija števca členov
    while (Math.Round(pi,clenov)!=Math.Round(Math.PI,clenov))
    {
       clen=4.00/(i*2+1);  //izracun tekocega clena
       if(i%2!=0)
           pi-=clen; //lihe člene odštevamo, krajši zapis za pi=pi-clen
       else
           pi+=clen; //sode člene prištevamo, krajši zapis za pi=pi+clen
       i++;
    }
    return i;
}

//Glavni program
static void Main(string[] args)
{
```

```
    Console.WriteLine("Računam koliko členov zaporedja 4 - 4/3 + 4/5 -
           4/7 + 4/9 - ... je potrebno\nsešteti, da bo tako dobljena vsota
           enaka konstanti Math.PI!");
    Console.WriteLine("\nTrenutek ......\n");

    //ujemanje na 9 decimalk
    Console.WriteLine("Število členov: " + stClenov(9));
}
```

Povprečje ocen

Napišimo program, ki bo sprejel nekaj 10 ocen, jih "zložil" v tabelo in nam vrnil povprečno, najnižjo in najvišjo oceno.

```
public static void Main(string[] args)
{
   int[] ocene = new int[10];
   for(int stevec = 0; stevec < ocene.Length; stevec++)
   {
        Console.Write("Vnesi " + (stevec + 1) + ". oceno:");
        ocene[stevec] = int.Parse(Console.ReadLine());
   } // for
   double povprecje = povprecjeOcen(ocene);
   Console.WriteLine("Tvoje povprečje je " + povprecje + ".");
   Console.WriteLine("Zaokroženo povp.je " + (int)(povprecje + 0.5) + ".");
   Console.WriteLine("Najnižja ocena je " + minOcena( ocene));
   Console.WriteLine("Najvišja ocena je " + maxOcena( ocene));
} // main

public static int minOcena(int[] ocene)
{
   int najnizja = ocene[0];

   for(int i = 1; i < ocene.Length; i++)
   {
       najnizja = Math.Min(najnizja, ocene[i]);
   } // for
   return najnizja;
} // minOcena

public static int maxOcena(int[] ocene)
{
   int najvisja = ocene[0];

   for(int i = 1; i < ocene.Length; i++)
   {
       najvisja = Math.Max(najvisja, ocene[i]);
   } // for
   return najvisja;
} // maxOcena

public static int vsotaOcen(int[] ocene)
{
   int vsota = 0;

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

```
        vsota = vsota + ocene[i];
    } // for
    return vsota;
} // vsotaOcen

public static double povprecjeOcen(int[] ocene)
{
    return (double)vsotaOcen(ocene) / ocene.Length;
} // povprecjeOcen
```

Program je sestavljen iz petih metod. Začnimo pri metodi *minOcena*, prvo oceno v naši tabeli označimo za najnižjo, postopoma se sprehodimo čez preostal del tabele in na vsakem koraku v spremenljivko *najnizja* shranimo trenutno najnižjo vrednost . Za primerjavo uporabimo metodo *Min()*iz razreda *Math*, ki nam vrne manjše število od obeh, ker je metoda tipa int nam vrne vrednost in to je mimimalna ocena. Sledi ji metoda *MaxOcena*, ki je podobna prejšnji metodi le da ta vrne maksimalno oceno. Naslednja metoda je *VsotaOcen*, ki vrne vsoto vseh ocen. S pomočjo zanke *for* se sprehodimo čez tabelo in seštevamo vrednosti. Metoda *PovprecjeOcen* pa vrne povprečje, ki ga dobimo z preprosto enačbo vsoto ocen deljeno z dolžino tabele ocene. Še zadnja je metoda *main,* v kateri preberemo podatke, jih zložimo v tabelo in s pomočjo klicev metod, ki smo jih ustvarili znotraj tega programa, izpišemo statistiko naših ocen.

Dopolnjevanje niza

Napišimo metodo *DopolniNiz*, ki sprejme niz *s* in naravno število *n* ter vrne niz dolžine *n*. V primeru, da je dolžina niza s večja od n, spustimo zadnjih nekaj znakov. Drugače pa na konec niza *s* dodamo še ustrezno število znakov '*+*'. Preverimo delovanje metode.

```
public static string dopolniNiz(string niz, int n)
{ // Metoda DopolniNiz
    // Vrni nov niz dolžine n, ki ga dobimo tako, da bodisi skrajšamo
    // niz niz, ali pa ga dopolnimo z znaki '+'.
    int dolzina = niz.Length; // Določimo dolžino niza
    string novNiz = ""; // Nov niz

    if (dolzina > n)
    { // Dolžina niza je večja od n
        novNiz = niz.Substring(0, n);
    }
    else
    { // Dolžina niza je manjša ali enaka n
        novNiz += niz;
        for (int i = dolzina; i < n; i++)
        {
            novNiz += '+'; // Dodamo manjkajoče znake'+'
        }
    }

    return novNiz; // Vrnemo nov niz
}
public static void Main(string[] args)
{ // Glavna metoda
    Console.Write("Vnesi niz: ");
    string niz = Console.ReadLine();
    Console.Write("Vnesi n: ");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine("Nov niz: " + dopolniNiz(niz, n)); // Klic metode
}
```

Razlaga. Program začnemo z metodo `dopolniNiz`, ki ji v deklaraciji podamo parametra niz in n. V metodi določimo dolžino niza niz, ki jo shranimo v spremenljivko dolzina. Določimo nov niz novNiz, ki je na začetku prazen. Preverimo pogoj v pogojnem stavku. Če je izpolnjen (resničen), kličemo metodo Substring(), s katero izluščimo podniz dolžine n. Rezultat te metode shranimo v spremenljivki novNiz. Če pogoj ni izpolnjen (neresničen), potem nizu novNiz dodamo niz niz in ustrezno število znakov '+'. Te nizu dodamo s pomočjo zanke for. Na koncu metode s ključno besedo return vrnemo niz novNiz.
Delovanje metode preverimo v glavni metodi `Main`. V tej metodi preberemo podatka iz konzole, ju shranimo v spremenljivki niz in n ter s pomočjo klicane metode DopolniNiz izpišemo spremenjeni niz.

Papajščina

Sestavimo metodo public static string Papajscina(string s), ki dani niz s pretvori v "papajščino". To pomeni, da za vsak samoglasnikom, ki se pojavi v nizu, postavi črko p in samoglasnik ponovi.

Primer:

Ko se izvedejo ukazi
string s = "Danes je konec šole.";
string papaj = Papajscina(s);
je niz papaj enak "Dapanepes jepe koponepec šopolepe.".

```
//metoda
public static string papajscina(string s)
{
    string sPapaj = "";
    for (int i=0;i<s.Length;i++)
    {
        //metoda bo delovala tudi če so v staku velike črke
        char znak=char.ToUpper(s[i]);
        if (znak=='A' || znak=='E' || znak=='I' || znak=='O' || znak=='U')
            sPapaj = sPapaj + s[i] + 'p'+s[i]; //dodamo znak 'p' in še
                                               //samoglasnik se ponovi
        else sPapaj = sPapaj + s[i];
     }
     return sPapaj; //vračanje papajščine
}
//glavni program
public static void Main(string[] args)
{
   string s = "Danes je konec šole.";
   string papaj = papajscina(s);
   Console.WriteLine(papaj);
}
```

Naloge za utrjevanje znanja iz metod

- Izmed spodnjih imen metod izberi tisto, ki se začne izvajati ob zagonu programa z imenom Test.
  - `static void Test(string[] vhod)`
  - `static void Main()`
  - `static void Test()`
  - `static void Main(string[] args)`

- Kakšen je izpis naslednje metode, če jo pokličemo s stavkom *vraca(10)*? Najprej reši "peš" in šele potem preizkusi z računalnikom.

```
public static void vraca(int n)
{
    int i = 1;
    while (i <= n)
    {
       if (i % 2 != 0) Console.Write(i);
       else Console.Write('#');
       i++;
    }
    Console.WriteLine('#');
}
```

- Dana je metoda

```
public static int Vsota(int n)
{
    int r = 1;
    int i = 1;
    while (i < n)
    {
        i = i + 1;
        r = i * r;
    }
    return r;
}
```

  - Kako je ime metodi?
  - Kakšen je tip rezultata, ki ga vrača metoda?
  - Koliko argumentov sprejme metoda?
  - Kakšni so tipi in imena argumentov?
  - Kaj metoda počne? Ali lahko predlagaš boljše ime za metodo?
  - Napiši program, ki demonstrira delovanje metode

- Napišite metodo, ki bo izpisala vsa števila med 0 in 1000, katerih vsota števk je enaka številu, ki nastopa kot parameter te metode.

- Denimo, da ste stari 16 let in imate zelo sitne starše, ki želijo brati vaša sporočila. Zato se s prijateljem dogovorita, da si bosta sporočila pošiljala zakodirana. Napisati morate metodo, ki bo sporočilo zakodirala. Zakodirano sporočilo naj bo sestavljeno najprej iz znakov, ki so bili v originalnem sporočilu na sodih mestih in nato iz znakov, ki so bili v originalnem sporočilu na lihih mestih.
  Primer:
  Originalno sporočilo Greva na pijaco?, se zakodira v sporočilo Gean iaorv apjc?.

- Za lažje sporazumevanje s prijateljem sestavite še metodo, ki dekodira sporočilo, kodiramo po metodi prejšnje naloge.

- Kaj se izpiše na zaslon po zagonu metode *preveri(tab)*. ? Najprej reši "peš" in šele potem preizkusi z računalnikom.

```
static void Main(string[] args)
{
    int[] tab = { 1, 2, 4, 3, 0, 100, 1 };
    preveri(tab);

    Console.ReadKey();
}
```

```
public static void preveri(int[] tab)
{
    int i = 0;
    while (i < tab.Length)
    {
        if (tab[i] == 0)
        {
            Console.WriteLine('*');
            break;
        }
        Console.Write(tab[i] + ",");
        i++;
    }
}
```

- Sestavite metodo Razlika, ki sprejme tabelo celih števil in vrne razliko med največjim in najmanjšim elementom v tabeli.
  Primer:
  Ko se izvedejo ukazi
  int[] tab = new []{5,1,2,4,6,8};
  int raz = Razlika(tab);
  ima raz vrednost 7.

- Sestavite metodo, ki kot parameter dobi tabelo in permutacijo (permutacija 1 3 2 4 pove, da prvi element ostane tam, kjer je, novi drugi element je stari tretji, tretji je stari drugi, četrti pa ostane tam kjer je). Vrnemo novo tabelo, ki ima elemente premešane tako, kot zahteva permutacija.

- Napišite metodo, ki kot parameter dobi dve tabeli celih števil. Vrne naj novo tabelo, v kateri so tiste vrednosti, ki se pojavljajo v prvi tabeli in ne v drugi! Vemo, da so vse vrednosti v obeh tabelah med seboj različne (torej se v isti tabeli število ne ponovi).
  Primera:
  Če so v prvi tabeli podatki 4, 5, 6 in v drugi 5, 6, 4, naj metoda vrne prazno tabelo.
  Če so v prvi tabeli podatki 4, 5, 6 in v drugi 1, 2, 4, naj metoda vrne tabelo 5 in 6.

- Sestavite metodo public static string Papajscina(string s), ki dani niz s pretvori v "papajščino". To pomeni, da za vsak samoglasnikom, ki se pojavi v nizu, postavi črko p in samoglasnik ponovi.
  Primer:
  Ko se izvedejo ukazi
  string s = "Danes je konec sole.";
  string papaj = Papajscina(s);
  je niz papaj enak "Dapanepes jepe koponepec sopolepe.".

- Napiši metodo sekunde, ki sprejme tri cela števila : ure, minute in sekunde. Metoda naj izračuna, koliko je to sekund in vrne rezultat.

- Napiši metodo, ki izpiše vsa števil med 0 in 1000, katerih vsota števk je enaka številu, ki nastopa kot parameter te metode.

- Napiši metodo, ki dobi za parameter poljubno decimalno število in ki izračuna in vrne vsoto vseh cifer v tem številu!

- Sestavi metodo kopije(String s, int k), ki sprejme niz s in pozitivno celo število k ter vrne niz, ki je sestavljen iz k kopij niza s.

- Napiši metodi zakodiraj in odkodiraj, ki sprejmeta niz in vrneta zakodiran oz. odkodiran niz. Metoda zakodiraj naj vsako črko v tekstu zamenja s črko, ki v (angleški) abecedi leži 5 črk za originalno. Menjava je seveda ciklična, tako da črko a zamenjata s črko e, črko z pa s črko d. Upoštevaj, da so črke lahko male ali pa velike. Seveda naj metoda ločil in ostalih znakov ne kodira!

- Napiši metodo, ki sestavi posebno enostavno križanko in jo izpiše. To pomeni, da uporabnik vnese prvo besedo s poljubnim številom črk in jo napiše navpično navzol. Potem ga program za vsako črko te besede vpraša za besedo, ki se začne na to črko in je dolga 4 črke in jo zapiše vodoravno od tiste črke.

- Napiši metodo donos, ki ima tri parametre: cas, znesek in obresti. Cas: koliko let se denar obrestuje. Znesek: koliko denarja položimo na bančni račun. Obresti: koliko obresti (v %)se pripiše glavnici vsako leto. Uporabi obrestno obrestni račun in metodo preizkusi.

- Sestavi metodo s pomočjo katere boš izračunal produkt vseh sodih števil med dvema danima pozitivnima celima številoma.

- Napiši metodo, ki sprejme pozitivno celo število n in na zaslon izpiše naslednji vzorec: najprej se n-krat izpiše .(pika) in nato n-krat znak #. Prikazan je primer za n = 5:

  .....#####

- Sestavi metodo, ki bo izračunala vsoto notranjih kotov večkotnika. Namig: vsota notranjih kotov n - kotnika se izračuna po formuli: (n−2)·180.

Prenos parametrov po referenci

V vseh dosedanjih primerih so bili parametri, ki smo jih posredovali metodam, posredovani na privzeti način. Takemu načinu pravimo **prenos parametrov po vrednosti**. To pomeni, da je bila vrednost vsake spremenljivke posredovana ustreznemu parametru v metodi, ki je tako v resnici delala s kopijo originalne spremenljivke. Zaradi tega sprememba vrednosti parametra v metodi ni vplivala na velikost spremenljivke, ki smo jo navedli pri klicu metode.

Parametre pa lahko pri klicu metode kličemo tudi po **referenci**. V tem primeru dobi metoda le referenco na ustrezno spremenljivko, kar dejansko pomeni, da vsaka sprememba tega parametra v metodi, pomeni spremembo vrednosti spremenljivke, ki smo jo uporabili pri klicu te metode. Klic po referenci dosežemo s pomočjo rezervirane besede **ref**. Vendar pozor: besedico **ref** moramo napisati tako pred tipom parametra v glavi metode, kot tudi pred imenom spremenljivke pri klicu metode.

```
//Metoda
static void metoda(ref string s) //parameter s je posredovan po referenci
{
    s = "Spremenjen!";
}

//Glavni program
static void Main()
{
    string stavek = "Originalen stavek";
    metoda(ref stavek); //parameter stavek je klican po referenci
   // stavek je sedaj spremenjen
}
```

Metoda za zamenjavo vrednosti dveh spremenljivk

Napišimo metodo, ki naj dobi za parametra dve spremenljivki *st1* in *st2*, metoda pa naj zamenja vrednosti teh dveh spremenljivk.

```
//Metoda
static void zamenjaj(ref int st1, ref int st2)//parametra podana po
referenci
{
    int zacasna = st1;
    st1 = st2;
    st2 = zacasna;
}

//Glavni program
static void Main(string[] args)
{
    int stevilo1 = 200;
    int stevilo2 = 500;
    Console.WriteLine("Število 1: "+stevilo1+", število 2: " + stevilo2);
    zamenjaj(ref stevilo1, ref stevilo2); //klic parametrov po referenci
    Console.WriteLine("Število 1: "+stevilo1+", število 2: " + stevilo2);
}
```

Jezik C# pozna še en način klica parametrov po referenci, to je klic s pomočjo rezervirane besede **out**. Besedica *out* je sicer podobna besedici *ref*, razlika pa je v tem, da klic s pomočjo besedice *ref* zahteva, da je spremenljivka, ki nastopa kot parameter metode, pred klicem že inicializirana. Tudi pri klicu s pomočjo besedice *ref* velja, da jo moramo napisati tako pred tipom parametra v glavi metode, kot tudi pred imenom spremenljivke pri klicu metode.

Primera:

```
//Metoda
static void metoda(out int i)
{
    i = 44;
}
static void Main()
{
    int stevilo;  //spremenljivka še ni inicializirana
    metoda(out stevilo);
     //vrednost spremenljivke je 44
}
```

```
//Klic po referenci - out
static void methoda1(out int i, out string s1, out string s2)
{
    i = 44;
    s1 = "Danes je lep dan!";
    s2 = null;
}

//Glavni program
static void Main()
{
```

```
    int stevilo;
    string str1, str2;
    metoda1(out stevilo, out str1, out str2);
    //spremenljivke stevilo, str1 in str2 so dobile vrednosti v metodi
}
```

Zgled

Obrni niz

Napišimo metodo, ki obrne niz znakov (npr »abeceda« pretvori v »adeceba«).

```
//Metoda: parameter stavek je klican po referenci
static void obrni(ref string stavek)
{
    string pomozni = "";
    for (int i = stavek.Length - 1; i >= 0; i--)
        pomozni = pomozni + stavek[i];
    stavek = pomozni;
}

//Glavni program
static void Main(string[] args)
{
    Console.Write("Stavek: ");
    string stavek = Console.ReadLine();
    Console.WriteLine("\n\nOriginalni stavek: \n\n" + stavek);
    obrni(ref stavek); //Parameter klican po referenci
    Console.WriteLine("\n\nObrnjeni stavek: \n\n"+stavek+"\n\n");
}
```

Kreiranje metode s pomočjo čarovnika

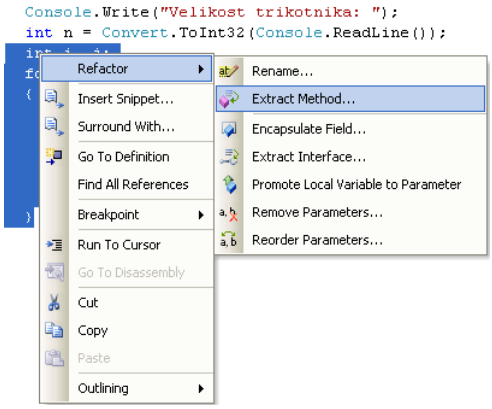
Bolj za informacijo, kot pa za neko resno uporabo si poglejmo še kako nam C# omogoča pisanje metod tudi s pomočjo čarovnika. Kot primer vzemimo naslednji program:

```
static void Main(string[] args)
{
    Console.Write("Velikost trikotnika: ");
    int n = Convert.ToInt32(Console.ReadLine());

    int i, j;
    for (i = 0; i < (n + 1); i++)
    {
        for (j = n - i; j > 0; j--)
            Console.Write(" ");

        for (j = 0; j < (2 * i + 1); j++)
            Console.Write("*");
        Console.WriteLine();
```

Tole kodo v glavnem programu bi radi nadomestili s funkcijo

```
    }
}
```

Kodo, ki bi jo radi s pomočjo čarovnika zapisali v novo funkcijo, najprej označimo. Nato kliknemo desni miškin gumb in v oknu, ki se odpre izberemo opcijo Rafactor in nato Extract Method.. V oknu, ki se odpre, nato še zapišemo ime funkcije (npr. trikotnik) in ime potrdimo s klikom na gumb OK. Čarovnik nam nato sam zgenerira ustrezno funkcijo.



V našem primeru bo rezultat takle:

```
//Glavi program
static void Main(string[] args)
{
     Console.Write("Velikost trikotnika: ");
     int n = Convert.ToInt32(Console.ReadLine());
     trikotnik(n);  //KLIC nove metode
}

private static void trikotnik(int n)
{
   int i, j;
   for (i = 0; i < (n + 1); i++)
   {
       for (j = n - i; j > 0; j--)
         Console.Write(" ");
       for (j = 0; j < (2 * i + 1); j++)
         Console.Write("*");
       Console.WriteLine();
   }
}
```

NOVA metoda – zgeneriral jo je čarovnik

Rekurzija

Naloge v praksi ponavadi zajemajo več kot en izračun, več pogojev, več zank. Zato jih smiselno razgradimo na podnaloge, rešimo vsako posebej in jih na koncu združimo skupaj. Podnaloge imenujemo podprogrami, rešimo jih z lastnimi metodami. Zaženemo pa jih v glavnem programu.
Zaenkrat imamo osnovno vedenje o metodah. Vse naše metode so statične, torej opremljene z besedico static. Vemo, kako takšne metode napišemo, kako jih kličemo …

Tokrat si bomo najprej ogledali močan prijem v programiranju – rekurzivne metode.

Rekurzivni klic metode je močno programersko orodje. Velikokrat se s pomočjo rekurzije algoritmi lažje in pregledneje izrazijo. O rekurzivnem klicu oz. rekurzivnih metodah govorimo takrat, ko **metoda kliče samo sebe**.

Kaj je to rekurzija

Vrsto let nazaj je bila zelo priljubljena pesmica, ki je zlasti prišla prav, če je učiteljica naročila, da se moraš naučiti poljubno pesmico, ki mora imeti vsaj 50 vrstic. Pesmica je šla takole

O možu in psu

Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel mu je kos mesa, zato ga je ubil.
Postavil mu je spomenik in nanj napisal:
Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel mu je kos mesa, zato ga je ubil.
Postavil mu je spomenik in nanj napisal:
Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel mu je kos mesa, zato ga je ubil.
Postavil mu je spomenik in nanj napisal:
Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel mu je kos mesa, zato ga je ubil.
Postavil mu je spomenik in nanj napisal:
Živel je mož, imel je psa, lepo ga je učil.
Nekoč ukradel ...

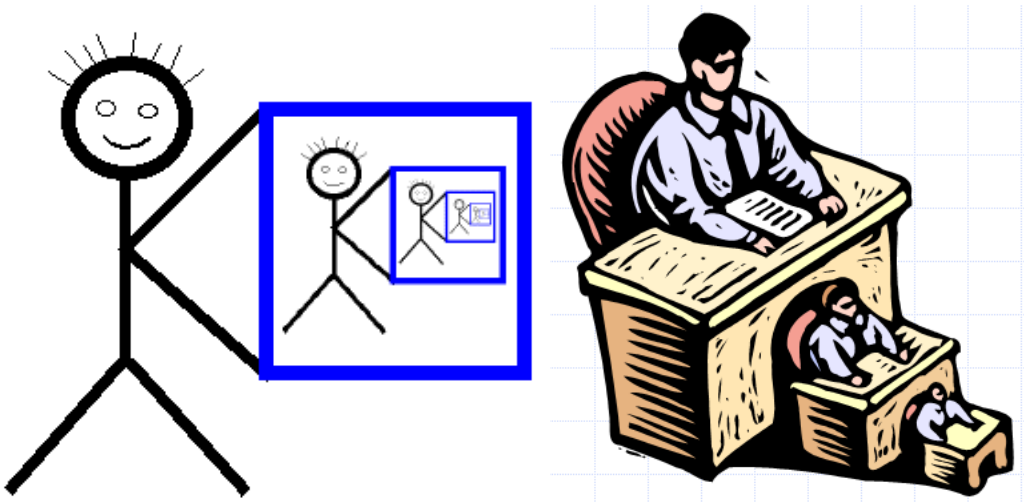
Tudi za dolge spise s predpisanim številom besed obstaja recept

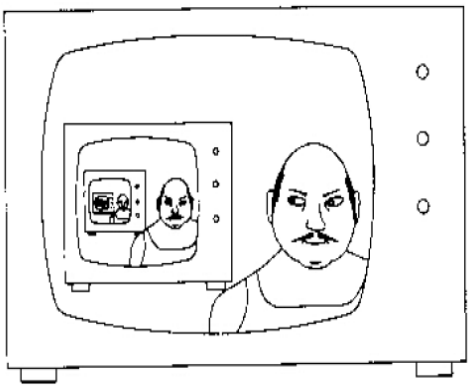
Kapitanova zgodba

Bila je temna, nevihtna noč ... Ladjo so valovi premetavali naprej in nazaj, veter je zavijal med jadri in dež se je zlival na palubo. Posadka je bila zbrana ob petrolejki. Vsi so se zavijali v odeje in trepetali, ko je kapitan pričel pripovedovati zgodbo:
"Bila je temna, nevihtna noč ... Ladjo so valovi premetavali …"

Kaj je skupnega tema dvema "literarnima deloma". Oba sta pravzaprav določena sama s sabo. Če bi ju želeli opisati na kratko bi reki:
Pesem o možu in psu govori o možu, ki je psu postavil spomenik in nanj napisal Pesem o možu. Kapitanova zgodba govori o kapitanu, ki svojim možem pripoveduje Kapitanovo zgodbo.
Tudi na likovnem področju hitro najdemo primere tovrstnih zgradb. Na sliki je le nekaj naključno izbranih primerkov iz različnih virov.

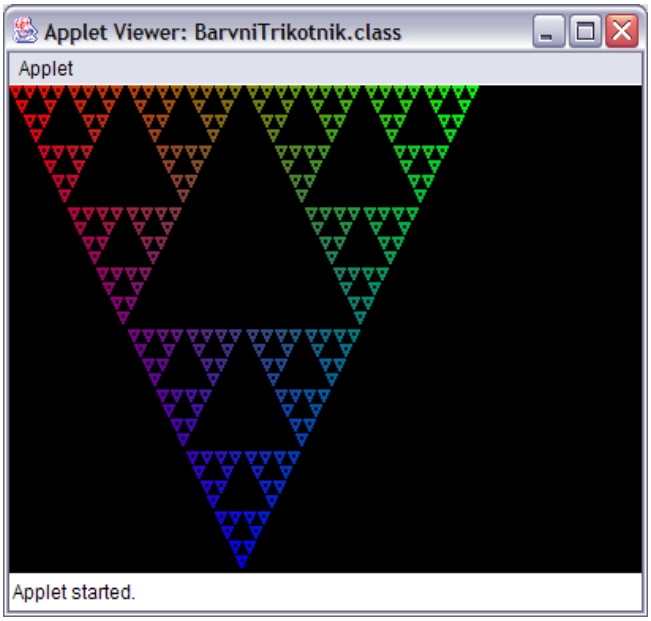
Slika v sliki



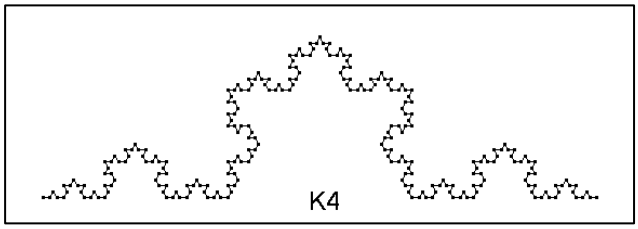


Problemi

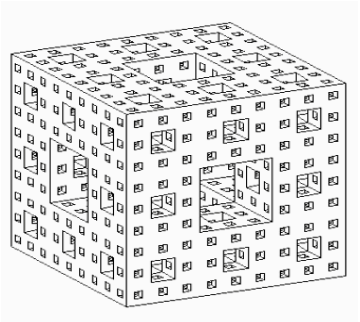
Oglejmo si nekaj različnih problemov. Prvi zahteva, da narišemo trikotnik Sierpinskega, katerega barvna slika je



Spet naslednji zahteva, da izračunamo dolžino tako imenovane Kochove črte stopnje 4 (ali več). Te Kochove črte pogosto uporabljajo pri modeliranju robov objektov v naravi (računalniške igrice, simulacije …)



Spet tretji problem (najdemo ga npr. v gradbeništvu, arhitekturi, pri konstrukciji različnih vpojnih materialov …) zahteva izračun volumna t.i. Mengerjeve spužve - kocke, preluknjane po določenem postopku n-krat



Naštejmo še nekaj problemov:

- Poišči največje in najmanjše število v tabeli števil
- Uredi podatke po velikosti.
- Izračunaj produkt naravnih števil od 1 do n.
- Izračunaj y^n s čim manj množenji.

Če je naša naloga sestaviti navodila (postopek), s katerim bi problem rešili, imajo vsi ti različni problemi, navkljub različnosti skupni prijem, ki mu rečemo **rekurzija**. Kaj pa ta beseda sploh pomeni? V Slovarju slovenskega knjižnega jezika SSKJ ni besede rekurzija, pojavlja pa se beseda rekurz -a m (u) knjiž. vrnitev (na kako stvar, dejstvo): rekurz na že omenjana dognanja ni potreben / v njegovih romanih so pogosti rekurzi v preteklost (tudi v filmih je tega precej) in rekurirati -am dov. in nedov. (i) knjiž. vrniti se (na kako stvar, dejstvo): pogosto rekurirati na nekatera dejstva, spoznanja. V Velikem slovarju tujk je dana definicija rekurzije (iz latinščine recurrere iti nazaj, vrniti se):

- definiranje funkcije ali postopka s samim seboj (informatika)
- izvajanje veličine ali funkcije, ki jo je treba šele definirati na že znano veličino (matematika). Označuje tudi zaporedje, katerega n-ti člen je določen z enim ali več predhodnimi členi.

Pri rekurziji gre torej za to, da je določena stvar (tudi postopek) določena tako, da v opisu nastopa spet ta ista stvar.

Tudi v vsakdanjem življenju srečamo rekurzijo. Definicija prednika neke osebe je lahko:

- prednik osebe je eden od roditeljev osebe (osnovni primer)
- prednik pa je tudi roditelj kateregakoli prednika (rekurzivni primer)

Če pogledamo na slovensko wikipedijo, vidimo, da rekurzijo razložijo kar s sliko, ki v bistvu prikazuje samo sebe:

Rekurzivna slika, na kateri je rekurzivna slika, na kateri je rekurzivna slika, na kateri ...

Ko govorimo o rekurziji pri programiranju, mislimo na to, da pri sestavljanju algoritma (metode) za reševanje nekega problema v rešitvi uporabimo klic (enega ali več) te metode. Rešitev problema je torej podana s samim problemom, le nad manjšim ali pa drugačnim obsegom podatkov.
V opisu postopka rešitve torej uporabimo kar ta postopek. Če želimo priti do rešitve, seveda ne moremo nadaljevati v nedogled kot npr. pri pesmici, saj se potem naš postopek ne bi nikoli končal. Zato je pri rekurzivnih algoritmih zelo pomembno določiti ustavitveni pogoj. Ta pove, daj v postopku ne uporabimo istega postopka (ne pokličemo te metode). Običajno je to takrat, ko so podatki (parametri metode) taki, da je problem "majhen" (enostaven).

Bistvo rekurzije je, da problem razdelimo na več podproblemov enake narave oz. da rešitev problema podamo s samim istim problemom, le na manjši količini podatkov. Vsaka rekurzija je sestavljena iz rekurzivnega dela, kjer funkcija kliče samo sebe in ustavitvenega pogoja. Slednji določi, kdaj pri reševanju rekurzivnega problema ne bomo uporabili rekurzije oz. kdaj je problem tako majhen, da se delitev na manjše podprobleme ne izplača več.

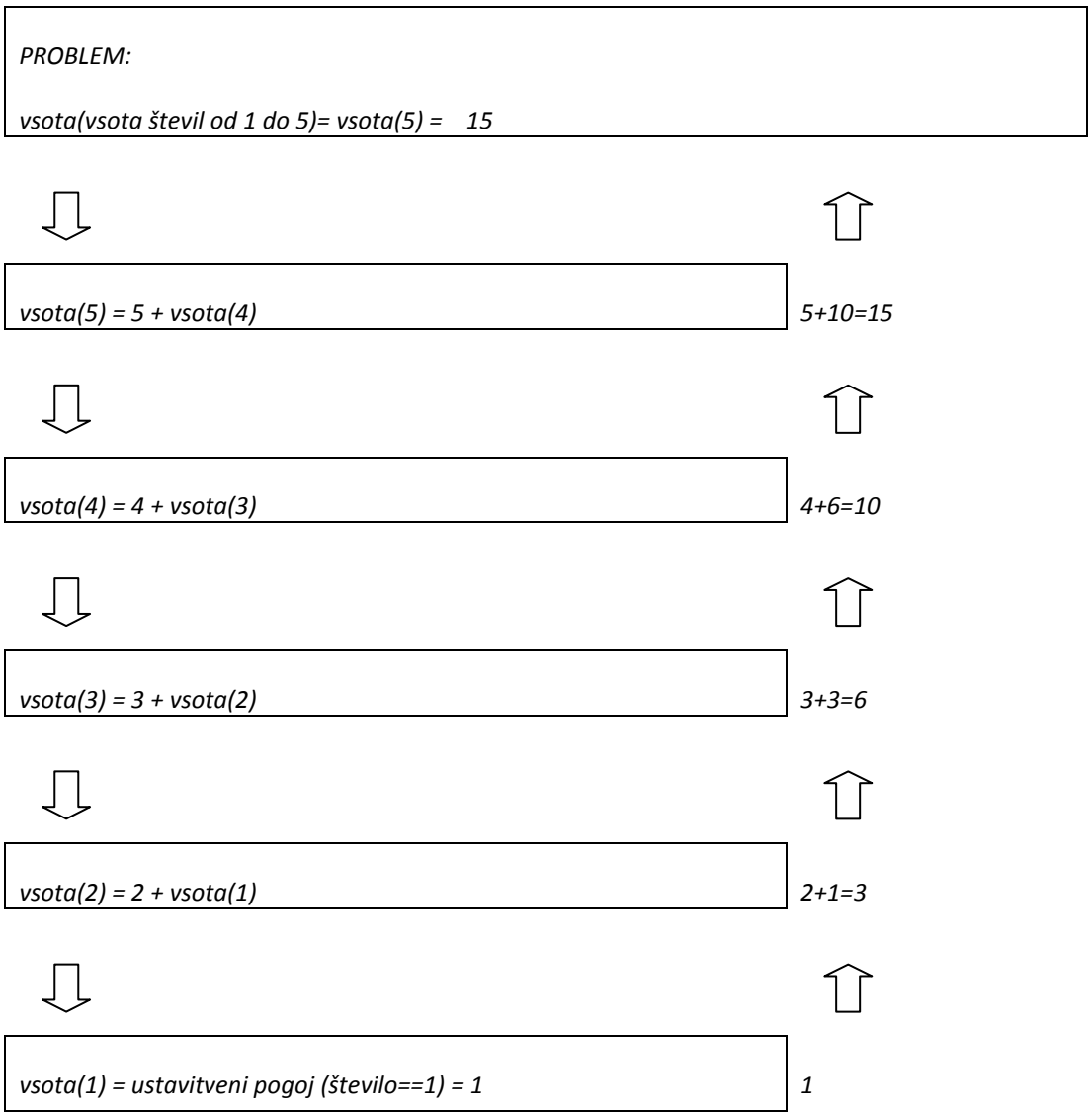
Rekurzivna metoda

```
rekurzivna_metoda (problem)
{
      if (problem majhen)
            return resitev;
      else
        razdeli problem na enega ali več manjših podproblemov enake vrste
        za vse podprobleme
      rekurzivna_metoda (manjši problem);
}
```

Zgledi

Vsota števil

Denimo, da želimo izračunati vsoto celih števil do n, kjer je n poljubno celo število. Recimo, da je n = 5. Potem moramo izračunati vsoto od 1 do 5. Sicer problem že znamo rešiti s pomočjo zanke, a tokrat se problema lotimo z rekurzijo. Vsoto števil do 5 namreč lahko izračunamo tudi tako, da k 5 prištejemo vsoto števil od 1 do 4. Slednje je spet problem iste vrste kot prvotni problem.



Problem *vsota(5)* smo razdelili na dva podproblema. Prvi sploh ni problem in vemo, da je njegova rešitev vrednost *5.* Drugi je *vsota (4).* Ker ne vemo koliko je vsota števil od 1 do 4, ponovno uporabimo rekurzijo in problem razdelimo na dva podproblema, *4* in *vsota(3)*, … Postopek izvajamo, dokler ne naletimo na *ustavitveni pogoj*, ki poskrbi, da se ustavimo. Sedaj sledi postopno vračanje. *Ustavitveni pogoj* nam vrne vrednost *1*, tej vrednosti prištejemo *2,* … Ko se vrnemo nazaj k prvotnemu problemu, dobimo rešitev našega problema, ki je *15*.

Če naše razmišljanje zapišemo v obliki algoritma, vidimo, da smo rekli sledeče

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

Ali če uporabimo zapis v obliki metode: `vsota(n) = n + vsota(n-1);`

Obema zapisoma manjka še ustavitveni pogoj. Dodajmo še tega

$$\sum_{i=1}^{1} i = 1$$

Oziroma `vsota(1) = 1;`

Zapišimo ustrezno metodo

```
public static int Vsota(int stevilo)
{
  if(stevilo == 1)
  {
    return 1;
  } // if
  else
  {
    int rezultat = stevilo + vsota(stevilo - 1); // rekurzivni klic
    return rezultat;
  } // else
} // vsota
```

Vidimo, da v metodi sami spet kličemo to metodo. Zaradi tega klica je metoda rekurzivna, saj je za opis postopka spet uporabljena metoda sama.

Zapišimo vse skupaj še v obliki programa. Tokrat našo metodo dopolnimo s klici izpisa, ki povedo, kako kličemo metodo in kakšen rezultat vrača.

```
public static void Main(string[] args)
{
    Console.Write("Število do katerega računaš vsoto: ");
    int stevilo = int.Parse(System.Console.ReadLine());

    vsota(stevilo);
    Console.ReadKey();
} // main

public static int vsota(int stevilo)
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == 1) // ustavitveni pogoj
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                                 "rezultat 1");
        return 1;
    } // if
    else
    {
        int rezultat = stevilo + vsota(stevilo - 1); //rekurziven klic
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                                 "rezultat " + rezultat);

        return rezultat;
    } // else
```

```
} // vsota
```

Program prevedemo in poženemo:

```
Stevilo do katerega racunas vsoto: 5
Racunam vsota(5)
Racunam vsota(4)
Racunam vsota(3)
Racunam vsota(2)
Racunam vsota(1)
vsota(1) vrne rezultat 1
vsota(2) vrne rezultat 3
vsota(3) vrne rezultat 6
vsota(4) vrne rezultat 10
vsota(5) vrne rezultat 15
```

Opis programa.
Oglejmo si metodo *Vsota*. Na začetku se nahaja ustavitveni pogoj. Če ta ni izpolnjen, sledi rekurzivni klic. Seveda je izpis znotraj programa namenjen le programerju za lažje razumevanje delovanja programa in v sam program ne spada (metodo bi zares napisali tako, kot smo navedli prej), nam pa potrdi, da je zgornja razlaga metode *Vsota* prava.

V primeru, če kot parameter pri klicu uporabimo negativno število, se program zacikla. Premislimo zakaj! No, če kot podatek vnesemo *-5* , vidimo da začne z izpisovanjem

```
Racunam vsota(-5)
Racunam vsota(-6)
Racunam vsota(-7)
Racunam vsota(-8)
Racunam vsota(-9)
Racunam vsota(-10)
Racunam vsota(-11)
Racunam vsota(-12)
...
```

Popravimo program tako, da bo delal tudi za negativna števila. Dogovoriti se moramo, kaj pomeni vsota števil do negativnega števila n. Recimo, da je n -4. Želimo, da vsota(-4) pomeni -4 + -3 + -2 + -1. Vemo sicer, da bi zadevo lahko enostavno rešili takole

$$Vsota(-n) = -Vsota(n)$$

A da se bomo bolj utrdili v spoznavanju rekurzije, bomo sestavili dve rekurzivni metodi. Prva, imenovana *PozitivnaVsota*, bo obstoječa metoda *Vsota* z drugim imenom, druga, *NegativnaVsota*, pa metoda, ki z rekurzijo izračuna vsoto negativnih števil, torej *vsota(-4) = -4 + vsota(-3)*.

```
public static void Main(string[] args)
{
    System.Console.Write("Število do katerega računaš vsoto: ");
    int stevilo = int.Parse(System.Console.ReadLine());
    if (stevilo >= 0)
    {
        PozitivnaVsota(stevilo);
    } // if
    else
    {
        NegativnaVsota(stevilo);
    } // else
} // main

public static int PozitivnaVsota(int stevilo)
```

```
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == 0)
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                        "rezultat 1");
        return 0;
    } // if
    else
    {
        int rezultat = stevilo + PozitivnaVsota(stevilo - 1);
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                        "rezultat " + rezultat);

        return rezultat;
    } // else
} // pozitivnaVsota

public static int NegativnaVsota(int stevilo)
{
    System.Console.WriteLine("Računam vsota(" + stevilo + ")");
    if (stevilo == -1)
    {
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                            "rezultat -1");
        return -1;
    } // if
    else
    {
        int rezultat = stevilo + NegativnaVsota(stevilo + 1);
        System.Console.WriteLine("vsota(" + stevilo + ") vrne " +
                            "rezultat " + rezultat);

        return rezultat;
    } // else
} // negativnaVsota
} // VsotaStevil3
```

Opis programa.
Program je sestavljen iz treh metod, metode *Main* in dveh rekurzivnih metod *PozitivnaVsota* in *NegativnaVsota*. V metodi *Main* preverimo, če je dano število pozitivno in če je, kličemo metodo *PozitivnaVsota*. Če pa je število negativno, kličemo metodo *negativnaVsota*. Ker smo si metodo *PozitivnaVsota* že podrobneje pogledali v prejšnjem zgledu, se bomo osredotočili na metodo *NegativnaVsota*. Kot vsaka rekurzivna metoda se tudi ta začne z ustavitvenim pogojem (ko pridemo do vrednosti *-1,* končamo z rekurzijo) in rekurzivnim delom. Recimo, da je prebrano število *-7*. Metoda *Main* kliče metodo *NegativnaVsota(-7)*. Ker ustavitveni pogoj ni izpolnjen, nadaljujemo z rekurzijo:

–7 + NegativnaVsota(-6);
–7 + (–6) + NegativnaVsota (-5)
–7 – 6 + (–5) + NegativnaVsota (-4)
–7 – 6 – 5 + (– 4) + NegativnaVsota (-3)
–7 – 6 – 5 – 4 + (– 3) + NegativnaVsota (-2)
–7 – 6 – 5 – 4 – 3 + (– 2) + ustavitveni pogoj
–7 – 6 – 5 – 4 – 3 – 2 – 1
–7 – 6 – 5 – 4 – 3 – 3
–7 – 6 – 5 – 4 – 6
–7 – 6 – 5 – 10

–7 – 6 – 15
–7 – 21
–28.

Faktoriela / fakulteta / n!

V matematiki, še posebej v kombinatoriki, pogosto naletimo na pojem faktoriele ali kot tudi rečemo fakultete nekega naravnega števila. Označimo jo s !. Faktoriela števila 7 je torej 7!. Faktoriela je zelo hitro naraščajoča zadeva.

- 3! = 6
- 5! = 120
- 42! = 1405006117752879898543142606244511569936384000000000

V matematiki jo običajno definiramo z rekurzivno definicijo:

n! = n * (n-1)!

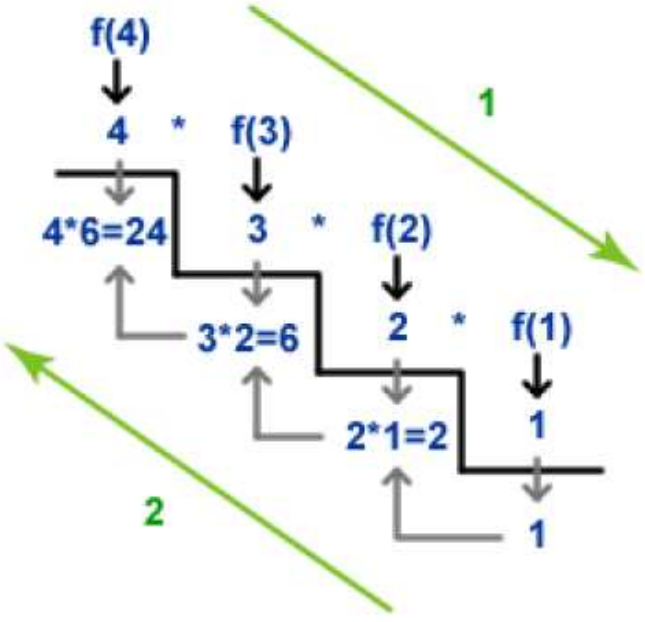
N! bomo torej izračunali, če bomo poznali (n-1)!

- 3! = 3 * 2!
  - 2! = 2 * 1! =
    - 1! = 1 * 0! =
      - 0! = 0 * (-1)! = ???

Omenjena rekurzivna definicija torej ni popolna. Manjka ji ustavitveni pogoj! Ta je določen z

0! = 1

Ko nas torej zanima fakulteta števila in je to število 0, ne uporabimo zgoraj omenjene rekurzivne definicije, ampak takoj odgovorimo, da je rezultat 1. Ker torej poznamo 0!, lahko izračunamo 1! (ki je 1). Če nas zanima fakulteta le pozitivnih (naravnih) celih števil, včasih kot ustavitveni pogoj določimo kar podatek 1 in torej za 1! kar neposredno odgovorimo, da je to 1. Če torej n! zapišemo kot f(4), se f(4) izračuna tako, kot kaže naslednja slika.



Postopek računanja 4! torej zahteva izračun 3!. Ta spet zahteva, da poznamo 2!, ki pa spet zahteva poznavanje 1!. A koliko je 1! vemo – to je 1. Zato lahko izračunamo 2! (2 * 1 = 2). Ker pa poznamo 2!, lahko izračunamo 3! (3 * 2 = 6). In ker poznamo 3!, ni ovir, da ne bi izračunali še 4!. To je 4 * 6, torej 24.

Faktoriela – postopek
Če stvar napišemo bolj računalniško, v obliki "kvazi metode". Kot ustavitveni pogoj uporabimo kar podatek 0. 1! se v tem primeru izračuna z enim klicem v globino, kot tudi rečemo, torej kot 1 * 0!

Fak(n):
  Če je n = 0, je rezultat 1
  sicer pa
    rezultat = n * fak(n – 1)

Zapišimo to še v obliki metode v jeziku C#.

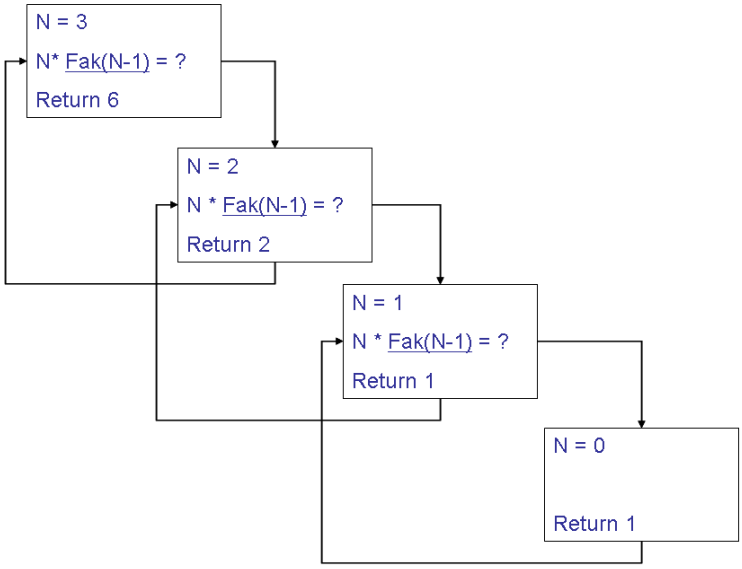
```
public static int fak(int stevilo)
{
    if (stevilo == 0)
    {
        return 1;
    }
    else
    {
        return stevilo * fak(stevilo - 1);
    }
}
```

Še glavni program in klic rekurzivne metode *fak*

```
public static void Main(string[] args)
{
    Console.Write("Vnesti število, katerega fakulteta te zanima:");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine(n + "! = " + fak(n));
}
```

V primeru, da je vneseno število *n* enako 3 dobimo na ekranu rezultat:

Poglejmo, kaj se dogaja ob klicu Fak(3)



Tisto, kar nam pri razumevanju rekurzije največkrat dela težave je, kje se dogaja vse to "knjigovodstvo" – zakaj je enkrat n 3, pa se potem spremeni v 2 in potem v 1 … Zavedati se je potrebno, da pri vsakem klicu metode sploh ni pomembno, da gre za rekurzivni klic. Ob vsakem klicu začnemo s "svežimi" vrednostmi (svežimi škatlicami kot so na zgornji sliki) in spremenljivke se med seboj "ne motijo". Ko klic metode opravi svoje delo (zve, koliko je rezultat), se vrnemo nazaj na mesto, kjer smo metodo klicali (v "škatlico" na sliki), kjer sedaj seveda veljajo tiste spremenljivke, ki so v tisti škatlici. Za vse te kopije spremenljivk, škatlice, mesta, kamor se mora metoda, ko konča delo vrniti, skrbi prevajalnik in izvajalno okolje.
Vračanje nazaj iz klicev v globino torej poteka avtomatsko, brez našega "vmešavanja":

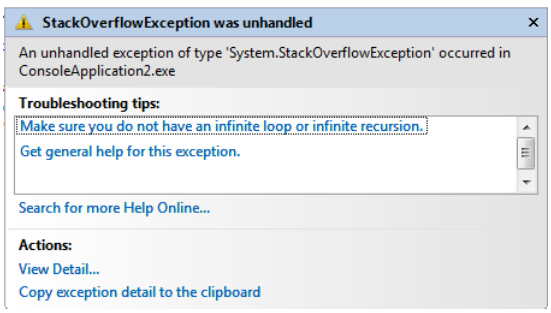
Opozorilo:

> Posebno pozornost namenimo argumentom v rekurzivnem klicu! Če namreč ob klicu argumenta ne spremenimo ustrezno, zlahka dobimo neskončno rekurzijo. Klici v globino se nikoli ne nehajo. Rekurzivni klic mora imeti izstopni pogoj oziroma ustavitveni pogoj.

Če npr. zgornjo metodo napišemo kot

```
public static int fak(int stevilo)
{
   return stevilo * fak(stevilo - 1);
}
```

nas kaj hitro pozdravi opozorilno okno o napaki:



Sistem javlja nekakšen **StackOverflowException**. Gre enostavno za to, da je izvajalnemu okolju zmanjkalo prostora za vse tisto knjigovodstvo, ki ga mora ob klicih metod opravljati. Ker se klic nobene metode ni zaključil (saj je vsaka čakala na rezultat metode, ki jo je ona poklicala) , je izvajalno okolje moralo skbeti za vse metode in enkrat je pač zmanjkalo prostora. Če naletimo na takšno napako, smo po vsej verjetnosti narobe napisali ustavitveni pogoj!

Pri računanju zlahka prekoračimo obseg tipa int. Če namreč poskusimo v zgornjem programu kot podatek vnesti npr. 32, dobimo v primeru izvajanja v okolju Visual Studio rezultat negativen – prekoračili smo namreč obseg celih števil.

Kako pa potem izračunati npr. 42! ? Seveda tudi to gre. Le na v C# vgrajene številske tipe se ne moremo več zanašati. Sestaviti bi bilo potrebno npr. metodo, ki bi rezultat vračala kot niz. Seveda pa bi bilo potem potrebno napisati še metodo, ki bi znala dolgo število (npr. z 80 števkami) predstavljenega kot niz pomnožiti s celim številom. Potem bi napisali nekaj takega:

```
1:      public static string Fak(int stevilo)   {
2:         if (stevilo == 0) {
3:            return "1";
4:         } else {
5:            return PomnoziSteviloInNiz(stevilo, Fak(stevilo - 1));
6:         }
7:      }
```

Metodo PomnoziSteviloInNiz pa napišite za vajo kar sami!

Če nekoliko premislimo in si ogledamo postopek računanja, vidimo, da faktorielo števila lahko izračunamo tudi drugače. Faktoriela je namreč produkt vseh števil od 1 do tistega števila. 7! je torej
7! = 1 × 2 × 3 × 4 × 5 × 6 × 7 = 5040
Zato tu rekurzivni pristop lahko zlahka nadomestimo z iterativnim oz. z zanko

```
public static int fakulteta(int st)
{
     int rezultat = 1;
     for (int j = st; j > 1; j--)
     {
         rezultat = rezultat * j;
     }
     return rezultat;
}
```

Vedno pa ni mogoče tako zlahka rekurzivnega postopka spremeniti v iterativnega. Pogosto je tudi zapis v rekurzivni obliki krajši in bolj pregleden.

Fibonaccijevo zaporedje

Dano je Fibonaccijevo zaporedje : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ... Vsak člen tega zaporedja (razen prvih dveh) je vsota predhodnih dveh členov. Označimo splošni, n-ti člen tega zaporedja z F(n). Pravilo o tem, kako dobimo n -ti člen , lahko izrazimo rekurzivno takole :

$F(n) = F(n-1) + F(n-2)$ n= 3, 4, 5 ....

Za prva dva člena zaporedja pa predpišemo še poseben pogoj :

$F(1) = 1$, $F(2) = 1$

Formulacija je rekurzivna, ker se v enačbi za splošni člen F(n) sklicujemo tudi na člena F(n-1) in F(n-2). Rekurzivno definicijsko enačbo lahko uporabimo tudi direktno za izračun določenega člena zaporedja, npr. za n=5. To storimo takole :

F(5) = F(4) + F(3)

Sedaj uporabimo pravilo za izračun F(4) in F(3) itn. ;

F(5)= F(4) + F(3) = (F(3) + F(2)) + (F(2) + F(1)) =
= ((F(2) + F(1)) + 1) + (1 + 1) =
= ((1 + 1) + 1) + (1 + 1) = 5

Napišimo sedaj rekurzivno metodo int `fib(int n)`, ki za dani n izračuna ustrezno Fibonaccijevo število . Zgornji rekurzivni princip lahko neposredno uporabimo takole :

če je n enako 1 ali 2, potem je fib(n) = 1,
sicer pa je fib(n) enako fib(n-1) + fib(n-2)

Zapis v C# :

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

//Primer klica te metode v glavnem programu
Console.WriteLine(Fib(33));  //Klic rekurzivne metode
```

Rekurzivna definicija se v gornjem podprogramu zrcali v tem, da podprogram kliče samega sebe (in to dvakrat) v stavku `return (fib(n - 1) + fib(n - 2));`

Takole pa bi napisali metodo za Fibonaccijeva števila brez rekurzije :

```
static int Fib1(int n)
{
    if (n == 1) return 1;
    if (n == 2) return 1;
    int F1 = 1, F2 = 1, P;
    for (int i = 0; i < n-2; i++)
    {
        P = F2;
        F2 = F1 + F2;
        F1 = P;
    }
    return F2;
}

//Primer klica te metode v glavnem programu
Console.WriteLine(Fib1(33)); //Klic iterativne funkcije
```

Za tako rešitev pravimo, da je formulirana iterativno, za razliko od rekurzivne.

Rekurzivna rešitev je krajša, enostavnejša in jasnejša, saj se v njej neposredno zrcali originalna matematična definicija Fibonaccijevega zaporedja in je zato tudi lažje razumljiva. Rekurzivna rešitev poleg tega tudi ne uporablja nobenih dodatnih spremenljivk. Glede varčnosti pomnilniškega prostora pa se izkaže, da je pri rekurzivni rešitvi le navidezna. Zahteve po rekurzivnih klicih se pri rekurzivnem podprogramu skrivajo v rekurzivnih klicih. Vsak rekurzivni klic namreč zahteva nekaj prostora, zapomniti si je potrebno tudi mesto, kam se je potrebno vrniti ob povratku iz podprograma. Za vsak rekurzivni klic pa si je potrebno zapomniti tudi vmesne rezultate. V resnici si vsak rekurzivni klic zgradi svoj povratni naslov in svojo verzijo vmesnih rezultatov

oz. lokalnih spremenljivk. Ti podatki se nalagajo v pomnilniški sklad (stack). Sklad je enostavno shranjen v pomnilniku in zanj poskrbi sam prevajalnik, tako da programerju o njem ni potrebno razmišljati. Sklad tako raste in pada po potrebi. Izkaže se, da tak sklad navadno zahteva več prostora kot iterativni podprogrami, rekurzivni podprogrami pa so zato nekoliko potratnejši od iterativnih. Podobna je situacija glede hitrosti izvajanja. Klic podprograma traja nekoliko dlje kot enostavne operacije v iterativnih zankah. V našem zgledu je rekurzivni podprogram še posebno počasen, saj zahteva več seštevanj kot iterativni.

Rekurzivne formulacije imajo torej prednost pred iterativnimi v tem, da je programiranje bolj enostavno, jasno in razumljivo. Nekoliko manj učinkovit pa je rekurzivni podprogram glede časa izvajanja in prostora v pomnilniku. Od konkretnega primera pa je odvisno, za katero pot se bomo odločili.

Izračun potence števila

Denimo, da želimo izračunati y^n, kjer je y poljubno pozitivno decimalno število, n pa neko naravno število, npr. 16.

y16 = y* y* y *y *y* y *y* y* y *y *y* y *y* y *y *y

Zlahka napišemo metodo, v njej uporabimo zanko in problem je rešen:

```
public static double potenca(double y, int n)
{
    double rezultat = 1;
    int i = 1;
    while (i <= n)
    { //n krat pomnožimo z y
      rezultat = rezultat * y;
      i = i + 1;
    }
    return rezultat;
}
```

A če to metodo uporabljamo zelo velikokrat, nas malo moti, da je potrebno kar 16 množenj. Še posebej, ker nam premislek pokaže, da do rezultata lahko pridemo le s štirimim množenji! Kako pa? Poglejmo:

- y16 = y8 y8

Če torej poznamo y8, lahko do y16 pridemo le z enim množenjem. In ko računamo y8, spet lahko uporabimo isti postopek:

- y8 = y4 y4
- y4 = y2 y2
- y2 = y y

Prihranek je torej precejšen. Še večji je pri večjih potencah. Npr. klasično je za izračun y1024 potrebnih 1024 množenj, z našim postopkom pa le 10. 100x hitreje torej.
A ta postopek gre tako lepo le, če je eksponent potenca števila 2 (torej 2, 4, 8, 16, 32, …). Kaj pa, če n npr.14. Če malo pomislimo, bo šlo z našim postopkom vedno, ko bo eksponent sodo število. Če pa je eksponent lih, naredimo le en vmesen korak:

- y14 = y7 y7
- y7 = y y6
- y6 = y3 y3
- y3 = y y2
- y2 = y y ..... 5 množenj

Ko torej računamo Pot(y, n) in je n sod, izračunamo Pot(y, n/2) in ga shranimo v pomožno spremenljivko, denimo pom. Rezultat metode pa je potem pom * pom.
Če pa je n lih, je rezultat te metode kar y * Pot(y, n – 1)

```
public static double pot(double y, int n)
 {
    if (n % 2 == 0)
    {
       double pom = pot(y, n / 2);
       return pom * pom;
    }
    return y * pot(y, n-1); // lih eksponent
}
```

Metoda pa ni v redu! Zakaj ne? Seveda, pozabili smo na ustavitveni pogoj. Razmislek pokaže, da je smiselno, da se ustavimo, ko je eksponent enak 0. Takrat vemo rezultat – 1 je!

```
public static double pot(double y, int n)
{
    if (n == 0) return 1;
    if (n % 2 == 0)
    {
       double pom = pot(y, n / 2);
       return pom * pom;
    }
    return y * pot(y, n-1); // lih eksponent
}
```

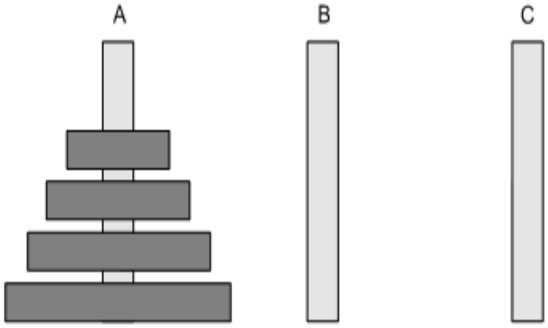
Seveda bi tudi to metodo (z manj množenji) napisali iterativno, a malo bolj bi se že namučili.

Hanoiski stolpiči

Preden se lotimo še enega primera, ki bo res pokazal moč rekurzije, premislimo.
Kaj je torej rekurzija? Kako naj bo v slovarju definirana beseda 'rekurzivno' ? Enostavno:

Piše naj: Glej 'rekurzivno'.

Zakaj gre pri problemu Hanoiskih stolpov?
Obstaja legenda, ki pravi, da so v hindujskem templju tri palice. Na začetku je bil na prvi sklad iz 64 zlatih obročev. Vsi so bili različne velikosti in zloženi po velikosti, tako da so bili manjši obroči na večjih v obliki piramide.

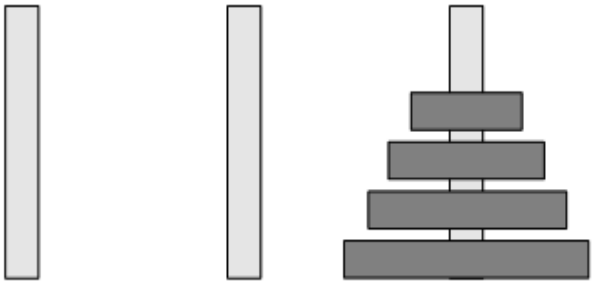


Brahmani premikajo te diske v skladu z pradavnimi pravili. Njihov cilj je vse diske prenesti na zadnjo palico, pri čemer morajo upoštevati pravila:

- vedno lahko premaknejo le po en disk naenkrat (to imenujemo poteza)

- na katerikoli palici morajo biti diski vedno razporejeni od največjega (na dnu) do najmanjšega (na vrhu).

Brahmani marljivo delajo noč in dan. Takoj, ko bi bila opravljena zadnja poteza



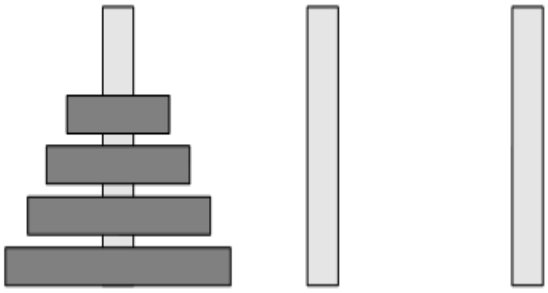
, naj bi se po starodavni prerokbi tempelj sesul v prah in svet naj bi izginil. Da ne boste preveč zaskrbljeni nad svojo usodo in usodo svojih potomcev, naj vas potolažimo: proces bo trajal zagotovo dlje, kot je do sedaj znana starost vesolja (za nekaj velikostnih razredov).
Zanima nas, kako morajo svečeniki prestavljati obroče, da bodo naredili kar se da malo potez (ds se končno znebimo te zasvinjane Zemlje in začnemo na novo). Problem Hanoiskih stolpičev je torej ta, da izpiše navodila, kako prestavljati obroče, če je na začetku na prvi palici n zlatih obročev (no, tudi če so železni ali virtualni, se problem ne spremeni). Npr. če bi našo metodo poimenovali Hanoi, naj bi klic
Hanoi(2, "A", "B", "C")
izpisal navodila:

- Preloži obroč z A na B
- Preloži obroč z A na C
- Preloži obroč z B na C

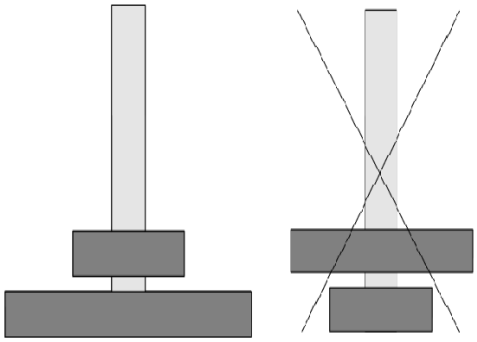
klic Hanoi(3, "P", "D", "T") pa

- Preloži obroč z P na T
- Preloži obroč z P na D
- Preloži obroč z T na D
- Preloži obroč z P na T
- Preloži obroč z D na P
- Preloži obroč z D na T
- Preloži obroč z P na T

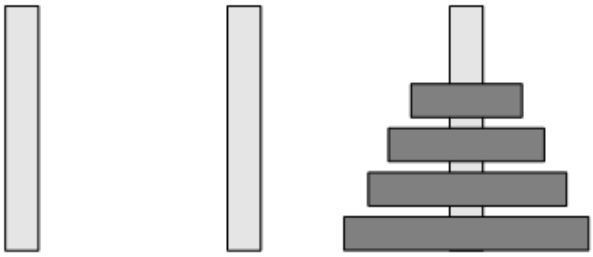
Pri problemu Hanoiskih stolpičev imamo torej začetno stanje



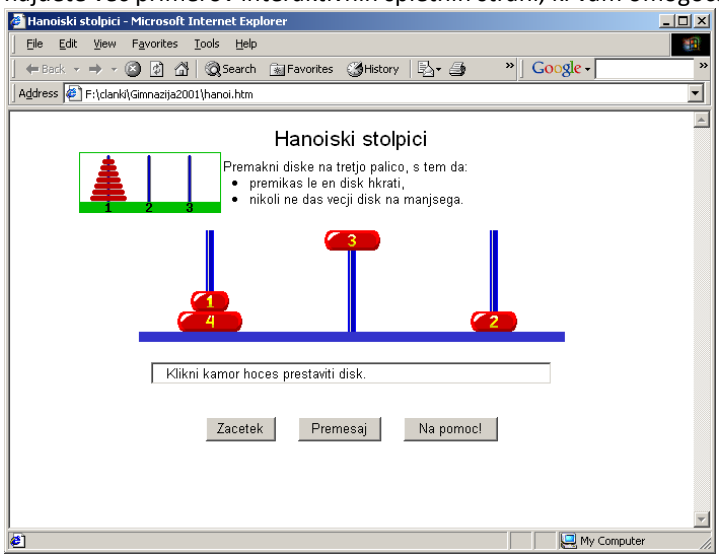
S prelaganjem po enega obroča hkrati, pri čemer pa moramo upoštevati pravilo "nikoli večji na manjšega"



moramo doseči končno stanje

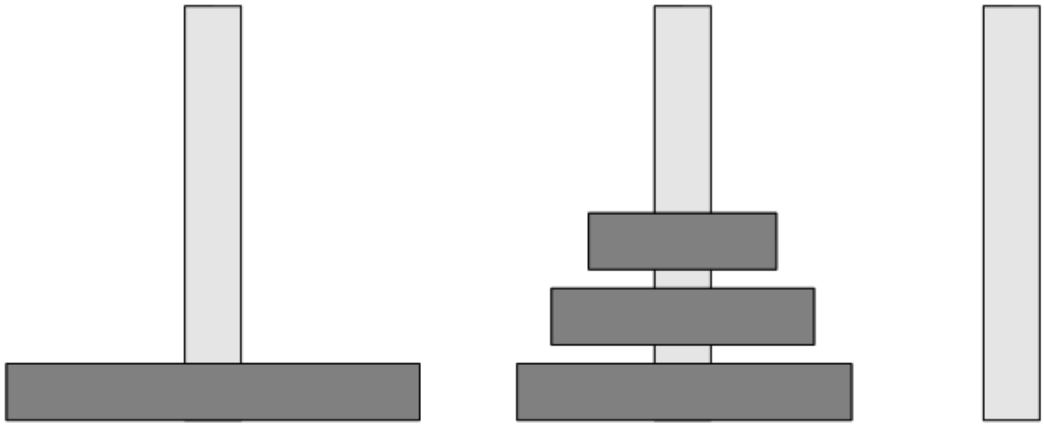


Da boste bolj spoznali problem, lahko na spletu (med drugim tudi v spletni učilnici, ki podpira ta predmet) najdete več primerov interaktivnih spletnih strani, ki vam omogočajo igranje te igre.



Hanoiski stolpiči - ideja

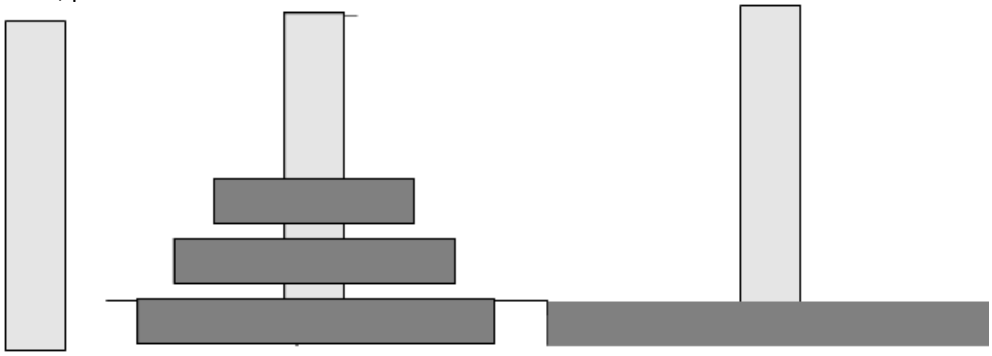
Potem, ko se nekoliko igramo z obroči, hitro spoznamo, da je eno od ključnih stanj pri premikanju tole:



Namreč, če želimo prestaviti največji obroč s stolpa A na stolp C, ne sme biti na stolpu C nobenega obroča (saj bi drugače dajali večji obroč na manjšega). Na stolpu A pa seveda tudi ne sme biti nobenega drugega obroča kot ta zadnji, saj drugače ne moremo do njega. Torej je vseh n – 1 obročev na srednjem stolpu, seveda pravilno zloženih. Vemo tudi, da prej največjega, n-tega obroča, nismo nič premikali (no, ko enkrat dosežemo opisano stanje, si ga lahko izmenično podajamo med A in C, a to nima smisla). Zato če za hip odmislimo največji obroč vidimo, da moramo za dosego zgoraj opisanega stanja pravzaprav rešiti zelo podoben problem.

premakni n-1 obročev z A na B (s pomočjo C)

Ko to naredimo, prestavimo obroč z A na C



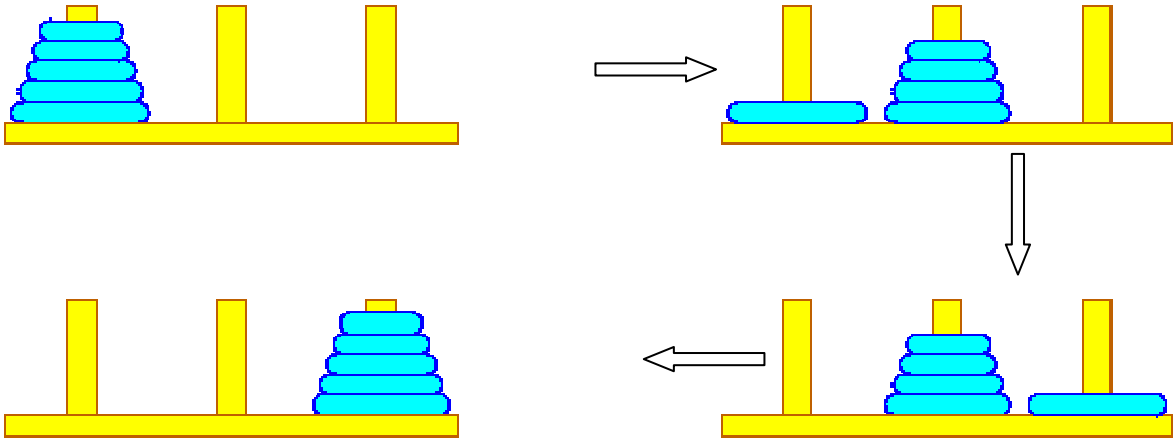
In če sedaj pogledamo sliko, vidimo, da nas največji obroč na C-ju ne bo nič motil – pridno bo čakal na svojem mestu. Zato ga lahko odmislimo. In spet smo pred podobnim problemom

premakni n –1 obročev z B na C (s pomočjo A)

In ko rešimo še ta problem, smo pri končni rešitvi. Če torej postopek reševanja zapišemo shematsko

- Preloži n obročev z A na C s pomočjo B
  - Preloži n-1 obročev z A na B s pomočjo C
  - Daj obroč z A na C
  - Preloži n-1 obročev z B na C s pomočjo A

in če zadevo še enkrat narišemo



Pri opisu postopka smo torej uporabili dva rekurzivna klica. Torej, da rešimo problem velikosti n, moramo rešiti dva enaka problema velikosti n – 1. Vsakič se sicer vloge strbrov malo zamenjajo, a to ne moti …
Seveda je potrebno napisati tudi ustrezni ustavitveni pogoj. Če rešujemo problem le za 1 obroč, potem ni kaj premišljevati. Le damo obroč z A na C in gotovi smo. Če torej upoštevamo še to, je postopek reševanja:

- Preloži n obročev z A na C s pomočjo B
  - Če je n == 1,
    - Daj obroč z A na C
  - sicer pa // n > 1
    - Preloži n-1 obročev z A na B s pomočjo C
    - Daj obroč z A na C
    - Preloži n-1 obročev z B na C s pomočjo A

In če zapišemo še malo bolj računalniško

```
/* n obročev z A na C s pomočjo B */
Hanoi(n, A, B, C)
  Če je n = 1 potem daj obroč z A na C
  sicer pa
     Hanoi(n-1, A, C, B)
     Daj obroč z A na C
     Hanoi(n-1, B, A, C)
```

Sedaj ni več težav s tem, da napišemo ustrezno metodo v jeziku C#

```
1: public static void hanoi(int n, string st1, string st2, string st3)
2: {
3:     if (n == 1)
4:     {
5:       System.Console.WriteLine("Preloži z " + st1 + " na " + st3);
6:     }
7:     else
8:     {
9:        hanoi(n-1, st1, st3, st2);
10:       System.Console.WriteLine("Preloži z " + st1 + " na " + st3);
11:       hanoi(n-1, st2, st1, st3);
12:    }
```

```
13: }
```

Poglejmo, kaj se dogaja, ko pokličemo `hanoi(3, "A", "B", "C")`

Naredimo "škatlico" za to metodo in vanjo shranimo n = 1, st1 = "A", st2 = "B" in st3 = "C".
Po preverjanju pogoja pride do klica hanoi(2, "A", "C", "B"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3) in to, da se mora vrniti po končanem omenjenem klicu na konec vrstice 9. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 2, v st1 = "A", v st2 = "C" in v st3 = "B". Po preverjanju pogoja se tudi pri izvajanju te metode pride do klica Hanoi(1, "A", "B", "C"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3) in to, da se mora vrniti po končanem omenjenem klicu (torej po končanem klicu hanoi(1, "A", "B", "C")) na konec vrstice 9. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 1, st1 = "A", st2 = "B" in st3 = "C". Ko se preveri pogoj, vidimo, da je na vrsti vrstica z izpisom. Ta izpiše niz

"Preloži z " + st1 + " na " + st3

kjer st1 in st3 zamenja z vrednostmi (katerimi – ja tistimi svojimi, iz trenutne škatlice, torej z "A" in "C". Izpiše se torej

Preloži z A na C

Sedaj je konec pogojnega stavka in s tem tudi konec izvajanja klica metode hanoi(1, "A", "B", "C"). Zato se škatlica izvajanja te metode "uniči" in vrnemo se tja, od kjer smo bili poklicani. To je na konec 9. vrstice v izvajanju klica metode hanoi(2, "A", "C", "B"). Sedaj sledi 10 vrstica. Ta izpiše niz

"Prelozi z " + st1 + " na " + st3

kjer st1 in st3 zamenja z vrednostmi (katerimi – ja tistimi svojimi, iz trenutne škatlice, torej z "A" in "B". Izpiše se torej

Prelozi z A na B

Sedaj je na vrsti 11. vrstica. Ne pozabimo, trenutne vrednosti n, st1, st2 in st3 so 2, "A", "C" in "B". Pride torej do klica hanoi(1, "C", "A", "B"). Zato si izvajalno okolje zapomni trenutne vrednosti škatlice (n, st1, st2, st3, torej 2, "A", "C" in "B") in to, da se mora vrniti po končanem omenjenem klicu (torej po končanem klicu hanoi(1, "C", "A", "B")) na konec vrstice 11. Sedaj se naredi nova škatlica za izvajanje metode. V n te nove škatlice shranimo 1, st1 = "C", st2 = "A" in st3 = "B". Ko se preveri pogoj, vidimo, da je na vrsti vrstica 3. Ta izpiše niz

Prelozi z C na B

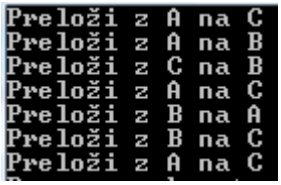
Sedaj je konec pogojnega stavka in s tem tudi konec izvajanja klica metode hanoi(1, "C", "A", "B"). Zato se škatlica izvajanja te metode "uniči" in vrnemo se tja, od kjer smo bili poklicani. To je na konec 11. vrstice v izvajanju klica metode hanoi(2, "A", "C", "B"). S tem pa smo tudi ta klic končali. Torej se vrnemo tja, od koder smo bili poklicani. To pa je konec 9 vrstice izvajanja klica Hanoi(3, "A", "B", "C"). Sedaj je na vrsti 10. vrstica, ki izpiše

Prelozi z A na C

V 11. vrstici spet pride do klica, tokrat hanoi(2, "B", "A","C") …
Če shemo klicanja napišemo shematsko

```
hanoi(3, "A", "B", "C")
        hanoi(2, "A", "C", "B")
                hanoi(1, "A", "B", "C")
                        izpis: Prelozi z A na C
                izpis: Prelozi z A na B
                hanoi(1, "C", "A", "B")
                        izpis: Prelozi z C na B
        izpis: Prelozi z A na C
        hanoi(2, "B", "A", "C")
                hanoi(1, "B", "C", "A")
                        izpis: Prelozi z B na A
                izpis: Prelozi z B na C
                hanoi(1, "A", "B", "C")
                        izpis: Prelozi z A na C
```

Izpis programa:



In koliko resnice je v legendi? Izračunamo lahko število premikov, ki jih morajo opraviti svečeniki. Premikov je točno 264 -1. Če bi vsako sekundo premaknili en obroč, bi potrebovali malo več kot 580 bilijonov let. In če upoštevamo, da je vesolje staro približno 13,7 bilijonov let, smo kar nekaj časa še lahko mirni!

MinMax

Dano imamo tabelo celih števil. Radi bi sestavili rekurzivno metodo public static int[] MinMax(int[] tabela), ki vrne tabelo dveh elementov. V odg[0] je minimalni element tabele stevila, v odg[1] pa maksimalni element tabele stevila.

Ideja:

- tabelo razpolovimo
- poiščemi min/max prvega dela (prvih pol elementov)
- poiščemi min/max drugega dela (druga polovica elementov)
- na podlagi min/max obeh delov poiščemo min/max celotne tabele

Npr.:

- Naj bo tabela {23, 4, 546, 56, 776, 8}.
- Poiščemo min/max {23, 4, 546} --- 4 in 546
- Poiščemo min/max {56, 776, 8} --- 8 in 776
- Odgovor za celo tabelo je 4 in 776

```
public static int[] minMax(int[] stevila)
{
   int[] rez = new int[2];
   if (stevila.Length == 1)
   { // le en element, vemo rezultat
      rez[0] = stevila[0];
      rez[1] = stevila[0];
      return rez;
   }
   // tabela je dolga, razpolavljamo
   int dol = stevila.Length;
   int[] prvidel = new int[dol/2];
   int[] drugidel = new int[dol - dol/2]; // new int[dol/2 + dol%2]
   int i = 0;
   // preložimo prvih pol elementov v tabelo prvidel
   while (i < dol/2)
   {
      prvidel[i] = stevila[i];
      i = i + 1;
   }
   // preložimo drugih pol elementov v tabelo drugidel
   i = 0;
   while (i < drugidel.Length)
   {
      // prelagamo drugi del tabele stevila, zato + dol/2!
      drugidel[i] = stevila[i + dol/2];
      i = i + 1;
```

```
    }
    // poiščemo min/max prvega dela
    int[] rezPrviDel = minMax(prvidel);
    // poiščemo min/max drugega dela
    int[] rezDrugiDel = minMax(drugidel);
    // določimo minimum celotne tabele
    if (rezPrviDel[0] < rezDrugiDel[0])
    {
        rez[0] = rezPrviDel[0];
    }
    else {
        rez[0] = rezDrugiDel[0];
    }
    // določi maksimum celotne tabele
    if (rezPrviDel[1] > rezDrugiDel[1])
    {
        rez[1] = rezPrviDel[1];
    }
    else
    {
        rez[1] = rezDrugiDel[1];
    }
    // vrni rezultat
    return rez;
}
```

MinMax – rešitev II

Zgornja rešitev ni najboljša, ker imamo veliko dela s prelaganjem elementov. Sestavi rešitev, kjer to ne bo potrebno!
Namig: sestavi pomožno metodo public static int[] MinMax(int[] tabela, int odKje, int DoKje), ki poišče minimalni in maksimalni element v delu tabele od odKje do doKje.

Palindrom

S pomočjo rekurzije preveri, če je niz palindrom.
Ideja:

- Prazen niz oziroma niz z enim znakom je palindrom
- Niz je palindrom, če se ujemata prvi in zadnji znak in je tudi srednji del (niz) palindrom

```
public static bool JePalindrom(string niz)
{
    // s pomočjo rekurzije ugotovi, če je niz niz palindrom
    // niz dolzine 0 in niz dolzine 1 JE palindrom
    if (niz.Length < 2) return true;
    // preveri prvi in zadnji znak
    if (niz[0] != niz[niz.Length-1]) return false;
    // vemo, da sta prvi in zadnji enaka
    string srednjiDel = niz.Substring(1, niz.Length - 1);
    // niz BO palindrom, če je srednji del tudi palindrom
    return JePalindrom(srednjiDel);
}
```

Številski sestav

Napišimo rekurzivno metodo za pretvarjanje iz desetiškega sestava v poljuben številski sestav med 2 in 9.

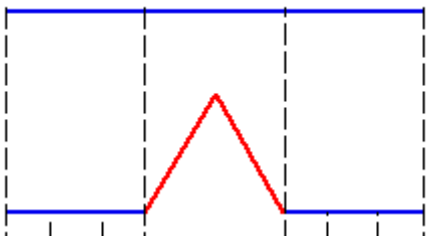
```
static void pretvori (int n,int s)
{
    if (n>0)
    {
        pretvori(n / s,s);
        Console.Write(n%s);
    }
    else Console.Write("\nPretvorjeno število : ");
}
```

Klic metode v glavnem programu:

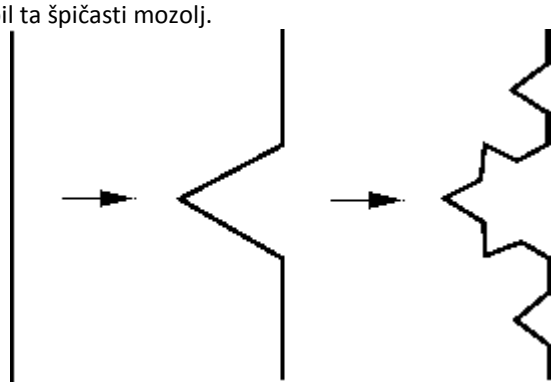
```
static void Main(string[] args)
{
    int n, s;
    Console.Write("Vnesi število : ");
    n=Convert.ToInt32(Console.ReadLine());
    Console.Write("\nŠtevilski sestav : ");
    s=Convert.ToInt32(Console.ReadLine());
    pretvori(n, s);
    Console.WriteLine();
}
```

Črta dobi mozolje

Zvečer je bila črta še čisto normalna. Lepo gladka je potekala od točke A do točke B. A zjutraj se je zbudila s čudnim občutkom. Odtavala je pred ogledalo in groza! Ni bila več lepo gladka. Nad njeno srednjo tretjino se je bohotil mozolj. Ampak kakšen – špičast, trikoten z robovi kar take dolžine, kot je bila prej dolžina srednje črte.



In pri tem se ni ustavilo. Naslednje jutro je vsak njen raven delček dobil nov mozoj – in to enak. Spet je nad srednjo tretjini ravnega dela bil ta špičasti mozolj.



In tako je šlo dan za dnem.

Končno ji je njena najboljša prijateljica, krožnica, povedala za čudovito kremo! Če se namaže z njo po vsakem delčku svoje kože, bo rast mozoljev vsaj ustavljena.

A krema je draga ... In za vsak cm potrebuje črta vsaj 6g te kreme. Koliko jo mora kupiti, če je bila na začetku dolga d in je preteklo že n dni, kar je dobivala mozolje?

```
public static double Dolzina(double razd, int dni)
{
     // izračunamo koliko je dolga črta, če se
     // stvar dogaja na delu njene kože v razdalji razd
     // in mozolje dobiva že n dni
     if (dni == 0)
     { // nobenega mozolja še ni
        // koža je gladka, torej je dolžina
        // kar enaka razdalji
        return razd;
     }
     else
     {
        // dolžina je enaka dogajanjem na 4 delih,
        // pri vsakem delu na razdalji 1/3 te
        // a za en dan pridobivanja mozoljev manj
        return 4 * Dolzina(razd / 3, dni - 1);
     }
}
```

Naloge za utrjevanje znanja iz rekurzij

- Dana je naslednja rekurzivna metoda. Ugotovi, kaj dela. Šele potem!! jo prenesi v testni program in zaženi ter preveri, da res dela tisto, kar si si predstavljal.

```
public static string KajDelam(int stevec)
{
   if(stevec <= 0)
     {
      return "";
     }
     else
     {
       return "" + stevec + ", " + KajDelam(stevec - 1);
     }
}
```

Metodo nato prepiši tako, da bo vrnila niz s števili v obratnem vrstnem redu kot prvotna.

- Nekega dne je Ančka vsa obupana prosila brata Jureta, naj ji napiše program za domačo nalogo. Ta bi moral rekurzivno izračunati vsoto prvih n naravnih števil. A ker se je Juretu mudilo na vsakodnevni žur, je moral zelo hiteti in je zato v programu naredil nekaj napak. Jih najdeš?

```
public static int Vsota(int n)
{
     if(n > 0)
      {
        return 1;
     }
      else
```

```
      {
        return vsota(n-1);
      }
}
```

- Jure je napisal kodo, ki s pomočjo rekurzije izračuna vsoto 1+1/2+1/3+...+1/n. Žal mu je med sprehodom po Ljubljani list s kodo padel v sneg in se je nekaj kode izbrisalo. Dopolni jo!

```
public static double VsotaRec(int n) // n>=1
{
    if (____________)  // ustavitveni pogoj
    {
       return _______________;
    }
    return _______ + VsotaRec(_______); //rekurzivni klic
}
```

- Oglej si naslednji rekurzivni program:

```
public static int puzzle(int baza, int limit)
{
//base in limit sta nenegativni števili
    if (baza > limit)
  {
    return -1;
  }
  else
  {
      if (baza == limit)
    {
      return 1;
    }
    else
    {
        return baza * puzzle(baza + 1, limit);
    }
  }
}
```

Program si najprej oglej, nato pa BREZ uporabe računalnika odgovori na spodnja vprašanja:

- kateri del metode Puzzle je ustavitveni pogoj?
- kje se izvede rekurzivni klic?
- kaj izpišejo sledeči stavki:
  - Console.WriteLine(Puzzle(14,10));
  - Console.WriteLine(Puzzle(4,7));
  - Console.WriteLine(Puzzle(0,0));

- Napiši rekurzivni podprogram, ki dobi za parameter poljubno celo število (npr. 1234567) vrne pa celo število v katerem so cifre zapisane v obratnem vrstnem redu (v našem primeru 7654321).

- Sestavi rekurzivno funkcijo, ki izračuna največji skupni delitelj dveh pozitivnih števil, če veš, da velja:

gcd(n, n) = n
gcd(n, k) = gcd(n - k, k) za n > k
gcd(n, k) = gcd(k, n) za n < k

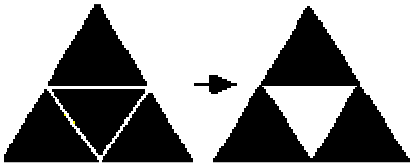
- Sestavi program, ki s pomočjo rekurzije izpiše vse permutacije števil od 1 do n. Permutacije naj bodo izpisane v leksikografskem vrstnem redu.

  1 2 3
  1 3 2
  2 1 3
  2 3 1
  3 1 2
  3 2 1

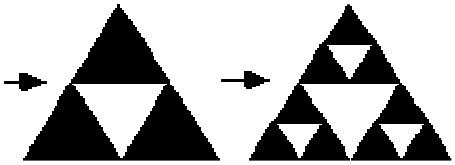
- Napiši rekurzivno funkcijo za izpis **poštevanke** poljubnega števila!

- S pomočjo rekurzije napiši program, ki izračuna vsoto vrste $1^0 + 2^1 + 3^2 + 4^3 + \ldots + x^{(x-1)}$.

- Napiši rekurzivno metodo, ki izračuna ploščino trikotnika Sierpinskega dane stopnje. Osnovni trikotnik je kar črni enakokraki trikotnik. Trikotnik stopnje 1 dobimo tako, da iz osnovnega trikotnika izrežemo trikotnik, katerega oglišča so razpolovišča stranic osnovnega trikotnika. Na tak način dobimo 3 manjše črne trikotnike spodaj levo, spodaj desno in zgoraj:



Trikotnik 2. stopnje dobimo iz trikotnika stopnje 1 tako, da izrežemo trikotnike iz treh črnih trikotnikov na enak način, kot smo ga izrezali iz osnovnega trikotnika:



Trikotnik tretje stopnje spet dobimo tako, da iz vsakega črnega trikotnika v trikotniku 2. stopnje ponovno izrežemo trikotnik itd.
Namig: namesto da »izrežeš« beli trikotnik, raje računaj ploščine črnih trikotnikov – razmisli, kako na tak način prideš iz ene stopnje do druge!
Alternirajoče vsote
Sestavi metodi

- public static int Vsota (int n) , ki izračuna vsoto naravnih števil od ena do n.
- public static int AltVsota (int n) , ki izračuna alternirajočo vsoto naravnih števil med ena in n. Tako klic AltVsota (4) izračuna vsoto 1 - 2 + 3 - 4 = -2.

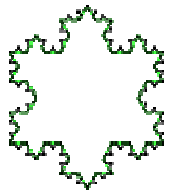
Obe metodi napiši nerekurzivno in rekurzivno!

- Jure je razvil napreden postopek, ki na podlagi makroekonomskih podatkov predvidi gibanje cen delnic podjetij. Algoritem za postopek imenuje JurePostopek. Algoritem sprejme kot parameter tabelo celih števil, deluje pa po sledečih pravilih:
  - če tabela prazna, naj vrne 0
  - če je dolžina tabele enaka 1, vrne ta element povečan za ena
  - sicer vrne kot rezultat (a+c)*JurePostopek(tabela brez prvega elementa), kjer je a prvi, c pa zadnji element tabele

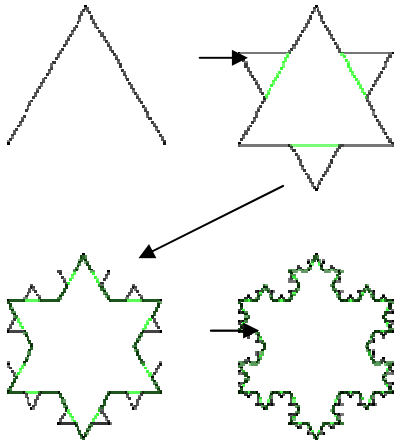
Primer izvedbe postopka na tabeli [1, 2, 3]:
JurePostopek([1, 2, 3]) = (1 + 3) * JurePostopek([2, 3]) = 4 * 20 = 80
JurePostopek([2, 3]) = (2 + 3) * JurePostopek([3]) = 5 * 4 = 20
JurePostopek([3]) = (3 + 1) = 4

Sestavi metodo, ki deluje po principu Juretovega postopka.

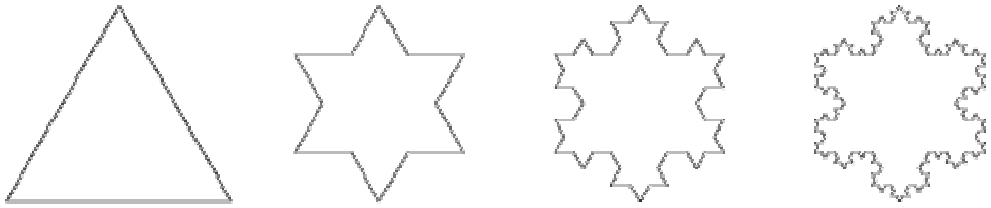
- Na sliki je Kochova snežinka stopnje 4



To lepo snežinko lahko dobimo z naslednjim postopkom.

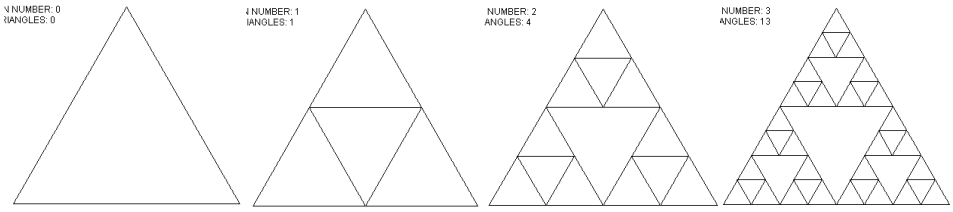


Stopnja 0 le kar enakostranični trikotnik. Nad vsako stranico podobno kot pri Kochovi črti potem srednji del nadomestimo z enakostraničnim trikotnikom.



Sestavi metodo, ki izračuna ploščino Kochove snežinke stopnje d in začetno stranico d.

- Trikotnik Sierpinskega stopnje 0 je običajni, enakostranični trikotnik s stranicami dolžine d. Trikotnik Sierpinskega stopnje n in s stranico d pa nastane, če zložite skupaj tri trikotnike Sierpinskega s pol krajšimi stranicami in stopnje n – 1. Na sliki so zaporedoma trikotniki Sierpinskega vsi z enako dolgimi stranicami, a stopnje 0, 1, 2 in 3.



Denimo, da bi radi iz žice sestavili model trikotnika Sierpinskega. Sestavi metodo, ki vrne true, če je možno iz žice dolžine d sestaviti trikotnik Sierpinskega stopnje st in s stranico s. Tako npr. je iz žice dolžine 20 sestaviti trikotnik Sierpinskega stopnje 1 s stranico 4, saj je skupna dolžine vseh črt v trikotniku

Sierpinskega stopnje 1 in s stranico 2 enaka 18. Iz žice dolžine 10 pa ni mogoče sestaviti trikotnik Sierpinskega stopnje 0 s stranico 4, saj bi za to potrebovali vsaj žico dolžine 12.