DUOMENŲ STRUKTŪROS IR ALGORITMAI

Rekursija

Rekursija

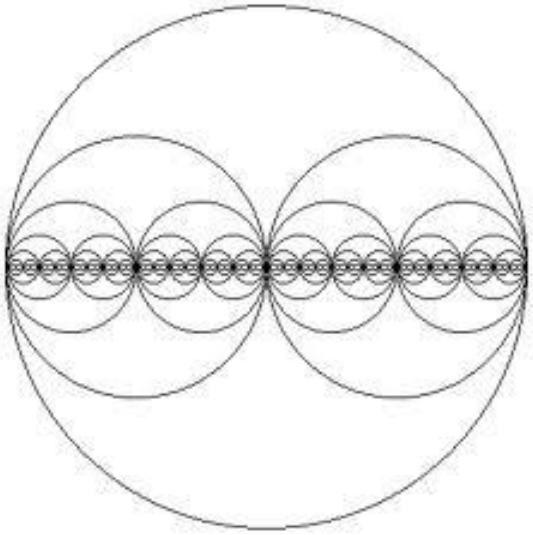
Rekursija – tai programų ar algoritmų sudarymo metodas, kai programa **kreipiasi pati į save**, esant mažesnėms (didesnėms) argumentų reikšmėms.

Pavyzdys

Sakykime, žmogus stovi tarp dviejų veidrodžių, o jo atspindys kartojasi begalybę kartų.

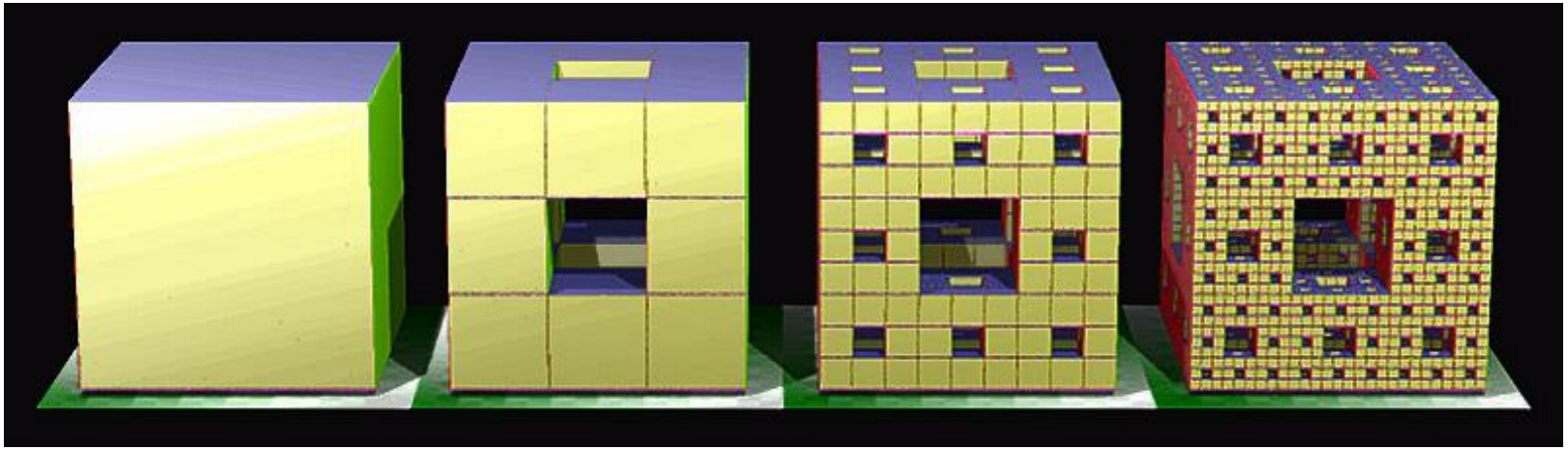
Rekursijai realizuoti naudojama funkcija, kurioje bent vieną kartą yra iškviečiama ji pati.

Pavyzdžiai



Mengerio kempinė

Pažintį su rekursija pradėkime nuo geometrinės figūros, vadinamos Mengerio kempine konstravimo algoritmo.



Mengerio kempinės algoritmas

1. Paimamas kubas.

2. Kubas suskaidomas į 27 vienodo dydžio kubelius.

3. Pašalinamas kubo viduryje esantis kubelis, taip pat dar 6 kiekvienos sienos viduryje esantys kubeliai.

4. Toliau su kiekvienu likusiu kubeliu veiksmai kartojami nuo 2 žingsnio.

Pateiktas konstravimo algoritmas yra rekursyvus, nes ketvirtame žingsnyje nurodoma tą patį algoritmą taikyti kitiems duomenims.

Rekursija

- Aukšto lygio programavimo kalbos suteikia galimybę aprašyti rekursyvias funkcijas, t. y. funkcijas, kurios iškviečia pačios save. Kiekvieną kartą kreipiantis į funkciją, įsimenamas **grįžimo adresas**, **padaromos parametrų kopijos** ir **sukuriami nauji lokalūs funkcijos kintamieji**.

- Taip rekursijos vykdymo metu organizuojama steko duomenų struktūra. Taigi kiekvieną rekursyvų algoritmą galima realizuoti nenaudojant rekursijos, o suprogramuojant ir panaudojant savo dėklo duomenų struktūrą.

Rekursija

Algoritmas vadinamas rekursiniu, jei jis kviečia save patį atliekti dalį skaičiavimų

Rekursinis algoritmas privalo turėti dvi dalis:

- **Bazinę dalį**, kuri sprendžiama nenaudojant rekursijos;
- **Rekursinę dalį,** kurioje atliekamas tos pačios funkcijos kvietimas**.**

Kiekvienoje rekursinėje procedūroje turi būti numatyti visi **ribiniai atvejai, kuriuos pasiekus rekursija nutraukiama**.

Pavyzdys - Faktorialas

Klasikinis rekursijos pavyzdys – faktorialo skaičiavimas:

$$n! = (n - 1)! * n \quad \text{for } n > 1; \quad 1! = 0! = 1.$$

```
long Fact(int n)
{
    if (n < 1)
        return 1;                  // Bazinė dalis
    return n * Fact(n-1);      // Rekursinė dalis vykdomas, kol n > 1
}
```

Faktorialas

fact(4)

n4 = 4 * n3

= 4 * 3 * n2

= 4 * 3 * 2 * n1

= 4 * 3 * 2 * 1 * n0

= 4 * 3 * 2 * 1 * 1

= 4 * 3 * 2 * 1

= 4 * 3 * 2

= 4 * 6

= 24

Skaičiavimo laikas:

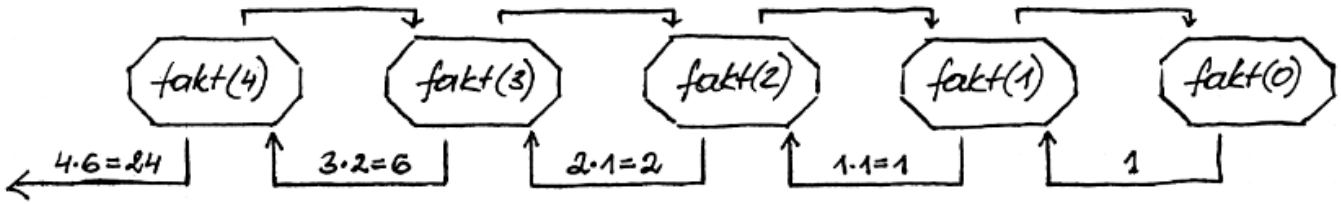
T(n) = T(n - 1) + 1 = (T(n - 2) + 1) + 1

(T(n - 2) + 1) + 1 = T(n - 2) + 2

T(n) = T(n - (n - 1)) + (n - 1)

= T(1) + n - 1

= n - 1



Iteracinis algoritmas

```
int fact_2(n)
{   int factorial = 1;
    if (n <= 1)
        return 1;
    for (int i = 1 ; i <= n ; i++)
        factorial = factorial * i;
    return factorial;
}
```

Rekursija

Rekursijos gylis – tai ilgiausia procedūrų grandinė, kuri suskaičiuoja funkcijos reikšmę.

Rekursyvi funkcija su ją iškvietusia funkcija (savo pačios „kopija“) gali bendrauti tik **parametrais bei globaliais kintamaisiais. Visų funkcijos kopijų kintamieji yra lokalūs.**

Pavyzdys

```
#include <iostream>
using namespace std;

int sum (int num)
{
    if (num==0)
            return 0;
  return sum(num-1)+(num);
}
```

```
int main()
{
      int num = 10;
      cout << sum(num)) << endl;
      getchar();

return 0;
}
```

Pavyzdys

```
#include <iostream>
using namespace std;

void Triangle (int x)
{
   if (x <= 0)
            return;
   Triangle(x - 1);
   for (int i = 1; i <= x; i++)
        cout << "*";
        cout << endl;
}
```

```
int main() {
   Triangle(7);
   return 0;
}
```

Trace of the program

Triangle(7)
Triangle(6)
Triangle(5)
Triangle(4)
Triangle(3)
Triangle(2)
Triangle(1)
Triangle(0) <-- base case
Triangle(1) <-- prints 1 star & new line
Triangle(2) <-- prints 2 stars & new line
Triangle(3) <-- prints 3 stars & new line
Triangle(4) <-- prints 4 stars & new line
Triangle(5) <-- prints 5 stars & new line
Triangle(6) <-- prints 6 stars & new line
Triangle(7) <-- prints 7 stars & new line

Fibonačio skaičiai

Kitas rekursyvios funkcijos pavyzdys – Fibonačio skaičiai.

Fibonačio skaičiai – tai skaičių seka, kuri prasideda dviem 1, o likę skaičiai sekoje surandami kaip prieš tai buvusių dviejų skaičių suma.

1 1 2 3 5 8 13 21 34 55 89 ...

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Rekursinis algoritmas

```
#include <stdio.h>

int fib(int);
int N = 10;

void main()
{  int Fnumber;
   for (int i = 0; i < N; i++)
   {   Fnumber = fib(i);
       printf("%d\n", Fnumber);
   }
}
```

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

statement		time $n \geq 2$
3	$O(1)$	$O(1)$
4	$O(1)$	--
6	--	$T(n\text{-}1)+T(n\text{-}2)+O(1)$
TOTAL	$O(1)$	$T(n\text{-}1)+T(n\text{-}2)+O(1)$

Iteracinis algoritmas

```
int fib_2(n)
{
    if (n == 0 || n == 1)
        return 1;
    int fibprev = 1;
    int fib = 1;
    for (int i = 2 ; i < n ; i++)
    {
        int  temp = fib;
        fib += fibprev;
        fibprev = temp;
    }
    return fib;
}
```

Pastaba

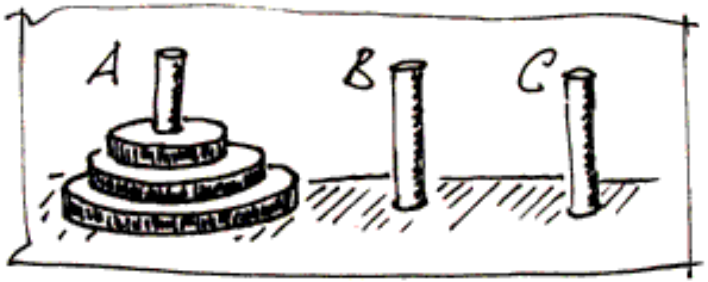
Nors ši funkcija atrodo tokia pat paprasta, kaip ir faktorialo, jos sudėtingumas yra eksponentinis. Taip yra todėl, kad kiekviena funkcija iškviečia net dvi kitas, antrines funkcijas, o joms perduodami argumentai sumažinami tik pastoviu dydžiu.

Pastebėkime, kad visi minėti uždaviniai pasižymi viena bendra savybe: spręsdami uždavinį, turime išspręsti analogiškus, bet mažesnius uždavinius.

Pavyzdžiui, jei norime suskaičiuoti n!, turime išspręsti mažesnį uždavinį – suskaičiuoti (n – 1)!

Hanojaus bokštai

Išspręsime klasikinį *Hanojaus bokštų uždavinį*, kurį 1883 metais suformulavo prancūzų matematikas Eduardas Lukas.



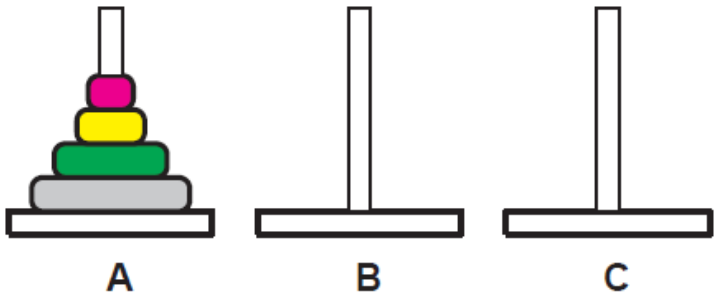
Hanojaus bokštai

Duoti trys stiebai ir *n* skirtingo dydžio diskai. Iš pradžių visi šie diskai sumauti ant pirmojo stiebo: apačioje pats didžiausias diskas, ant jo – mažesnis ir t. t. Viršuje užmautas pats mažiausias iš diskų.

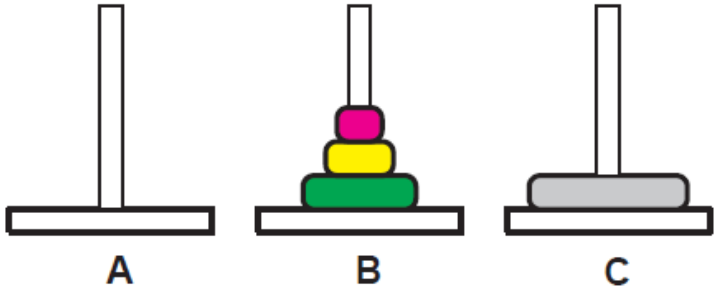
Užduotis. reikia perkelti visus diskus nuo pirmojo stiebo ant paskutinio laikantis šių taisyklių:

- Vienu ėjimu galima kelti tik vieną diską.
- Diską galima užmauti tik ant tuščio stiebo arba uždėti ant didesnio už jį disko.
- Atliekamų perkėlimų skaičius turi būti minimalus

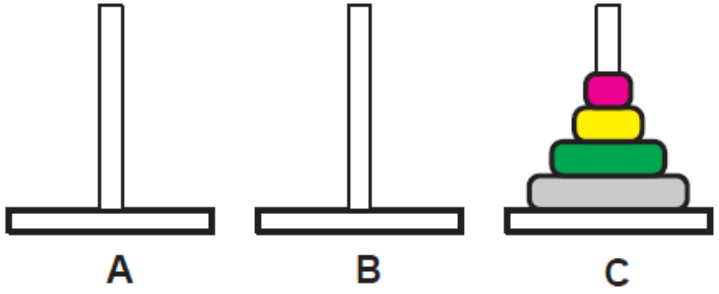
Hanojaus bokštai



a)



b)



Minimalus ėjimų skaičius $2^n - 1$, čia *n* diskų skaičius.

Hanojaus bokštai

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

hanoi(4)

= 2*hanoi(3) + 1

= 2*(2*hanoi(2) + 1) + 1

= 2*(2*(2*hanoi(1) + 1) + 1) + 1

= 2*(2*(2*1 + 1) + 1) + 1

= 2*(2*(3) + 1) + 1

= 2*(7) + 1

= 15

Rekursinis algoritmas

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

Rekursija sąrašo struktūroje

```
struct node
{ int n;
 node *next;
};
typedef struct node *LIST;

…..

void printList(LIST lst)
{ if ( ! isEmpty(lst) )                 // bazinė dalis
    { printf ("%d ", lst->n );        // spausdina elementus
      printList ( lst->next );        // rekursinė dalis
     }
}
```

Rekursija binariniame medyje

```
struct node
{   int n;
    struct node *left;
    struct node *right;
};
typedef struct node *TREE;

// Inorder printout of the binary tree :
void printTree(TREE t)
{ if (!isEmpty(t)) {
        printTree(t->left);
        printf("%d ", t->n);
        printTree(t->right);
   }
}
```

Rekursija

Rekursijos privalumai

- Patogiai kontroliuojama procedūros eiga
- Paprastas kodas

Rekursijos trūkumai

- Stack overflow
- Nevisada efektyvus algoritmas

Rekomendacija

Nenaudoti rekursijos, jei nežinomas elementų skaičius.

Užduotys

No.1

Taikant rekursinį algoritmą patikrinti ar skaičius pirminis.

No.2

Išvesti masyvo elementus tiesiogine ir atvirkštine seka naudojant rekursiją.

No.3

Parašyti rekursinę funkciją, kuri išvestų skaičiaus 2015 skaitmenis atvirštine tvarka t.y. 5 1 0 2.

Pavyzdys

```
int isPrime (int p, int i=2)
{
        if (i == p) return 1;      // geriau  if (i*i > p) return 1;
        if (p%i == 0)
            return 0;
    return isPrime (p, i+1);
}
```