Проектиране и Тестиране на Софтуер
ТУ, кат. КС, летен семестър 2012

Лекция 2

Тема:

1. Рекурсия и/или Итерация?

2. Данни

Съдържание:

- Част 1
  - Приложение на Рекурсия или Итерация в процедурно ориентираното програмиране
- Част 2
  - Данни и тяхното вътрешно представяне

Част 1

Рекурсия и Итерация в процедурното програмиране

Съдържание:

- Основни понятия
- Синтаксис за изразяване на итерация
- Синтаксис за изразяване на рекурсия
- Сравнение на итеративни и рекурсивни алгоритми
- Примери – първични текстове на C/C++, Java, C#, Perl, Python

Вместо въведение

“To iterate is human,
to recurse is divine.”

L. Peter Deutsch

Кой е L Peter Deutsch ?

- **L Peter Deutsch** (born **Laurence Peter Deutsch**, Aug 7, 1946,) is the creator of Ghostscript, a free software PostScript and PDF interpreter.
- Deutsch's other work includes the definitive Smalltalk implementation that, among other innovations, inspired Java just-in-time technology 15 years later. He also wrote the PDP-1 Lisp 1.5 implementation, Basic PDP-1 LISP, when he was 12–15 old.
- Deutsch received Ph.D. in CS from the Uni California, Berkeley in 1973. In 1994 he was inducted as a Fellow of the ACM.
- Deutsch changed his legal first name from "Laurence" to "L" on Sep 12, 2007. His published work and public references before that time generally use **L. Peter Deutsch** (with a dot after the **L**).
- In January 2009, after auditing undergraduate Music courses at Stanford University, he entered the postgraduate Music program at California State University, East Bay, and was awarded a M.A. in March 2011. As of mid 2011, he has had 6 compositions performed on public concerts, and now generally identifies himself as a composer rather than a software developer.

Основни понятия

- *Итерация* е начин, подход, средство за изразяване на повторение или цикличност в обработките на даден процес или структура данни.
- *Рекурсия* е начин, подход, средство за описание на
  - функция в термините на самата функция или
  - процес в термините на самия процес или
  - структура данни чрез елементи на самата структура данни.

Итерация

Iteration изразява повторение на процес в една компютърна програма. Използва се като **общ термин**, синоним на повторение и като **специфичен термин**, изразяващ вид повторение с промяна на състоянието (mutable state).

В първия смисъл, рекурсията е пример за итерация с прилагане на рекурсивна нотация, което не е случаят итерация.

Във втория смисъл, итерация описва програмистки стил (маниер) от процедурните (imperative) ПЕ.

Итерация

Пример за итерация на процедурен псевдокод:

```
var i, a := 0               // initialize a before iteration
 for i from 1 to 3 {        // loop three times
        a := a + i          // increment a by current value of I
               }
 print a                    // the number 6 is printed
```

В програмата променливата *i* получава стойности 1, 2 и 3. Тези променящи се стойности (*mutable state)* — са характеристика на итерацията.

Итерация

Итерация може да се апроксимира ползвайки рекурсия във функционалните ПЕ. Следният пример е на Лисп диалект Scheme. Забележете, че дефиницията на функция iter "how to iterate", съдържа обръщение към себе си:

```
(define (sum n)
   (define (iter n i)
     (if (= n 1) i (iter (- n 1)(+ n i) ) )
   )
   (iter n 1)
)
```

Итерация

- **Итерация** се изразява чрез разнообразни форми на оператори за цикъл (counter and logically controlled).

Синтаксисът на ПЕ предлага много форми на итеративни оператори като ***for, while, do … while, foreach*** и дори оператори ***if*** и ***goto***.

Итерация (примери)

```
for (int I=1; I<=100; I++) tab[I]= I*I*I;


int I=1;
while(I<101) { tab[I] = I*I*I; I++; }


int I=1;      do { tab[I] = I*I*I; I++; }
                  while (I<=100);
```

Рекурсия

Удобно е рекурсивно определение да се представи като дефиниция на обекти в термините на “преди това дефинирани" обекти от същия клас (категория).

Ето пример от математиката за рекурсивно определение на множеството на естествените числа:

- 0(1) е естествено число.
- Всяко естествено число има наследник (приемник), който също е естествено число.

Нулата (0) естествено число?

- В математиката, **естествено число** означава елемент на м-вото {1, 2, 3, ...} (т.нар. Положителни цели числа) или елемент на множеството {0, 1, 2, 3, ...} (т.нар. Не отрицателни цели числа). Първото множество се ползва в теория на числата number theory, докато второто се ползва в теория на множествата set theory и computer science.
- Естествените числа имат две главни приложения: за изброяване counting ("there are 3 apples on the table"), и за подреждане ordering ("this is the 3rd largest city in the country").

History of natural numbers and the status of zero

- The natural numbers presumably had their origins in the words used to count things, beginning with the number one.
- The first major advance in abstraction was the use of numerals to represent numbers. This allowed systems to be developed for recording large numbers. For example, the Babylonians developed a place-value system based essentially on numerals for 1 and 10. The ancient Egyptians had a system of numerals with distinct hieroglyphs for 1, 10, and all the powers of 10 up to one million. A stone carving from Karnak, dating from around 1500 BC and now at the Louvre in Paris, depicts 276 as 2 hundreds, 7 tens, and 6 ones;

History of natural numbers and the status of zero

- A much later advance in abstraction was the development of the idea of zero as a number with its own numeral. A zero digit had been used in place-value notation as early as 700 BC by the Babylonians. The Olmec and Maya civilization used zero as a separate number as early as 1st century BC, apparently developed independently.
- The concept as used in modern times originated with the Indian mathematician Brahmagupta in 628. Nevertheless, zero was used as a number by all medieval computists (calculators of Easter), but in general no Roman numeral was used to write it. Instead, the Latin word for "nothing," *nullus*, was employed.

Рекурсия

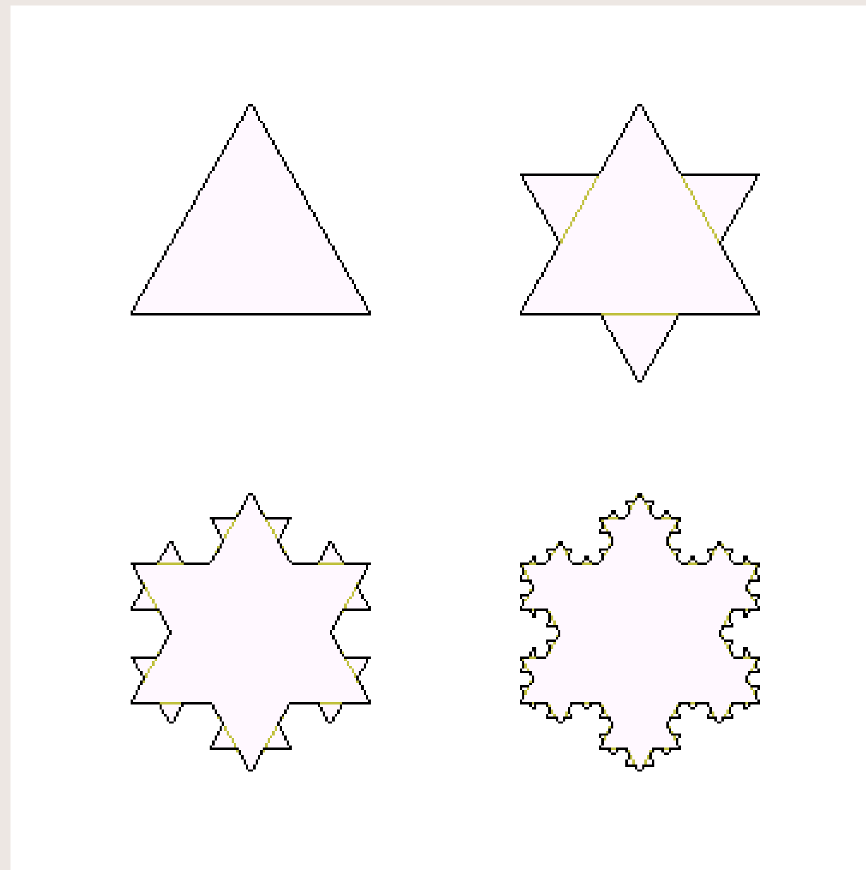
За визуализиране на рекурсията, тук са представени изображенията на три рекурсивно-дефинирани същности (2 фигури и 1 множество):

- крива на Кох (Koch curve);
- триъгълник на Шерпински (Sierpinski gasket);
- множество на Кантор (Cantor set).

Крива на Кох (Koch curve)

Снежинка на Кох обединява безброй области с форма на равностранен триъгълник. Периметърът на кривата расте неограничено (клони към безкрайност) с увеличение на броя триъгълници, но площта на фигурата остава крайна.

Крива на Кох (Koch curve)



Крива на Кох (Koch curve)

Кривата на Кох е като **Koch Snowflake** (още **Koch Star**), само че започва с линия (line segment) вместо с равностранен триъгълник (equilateral triangle).

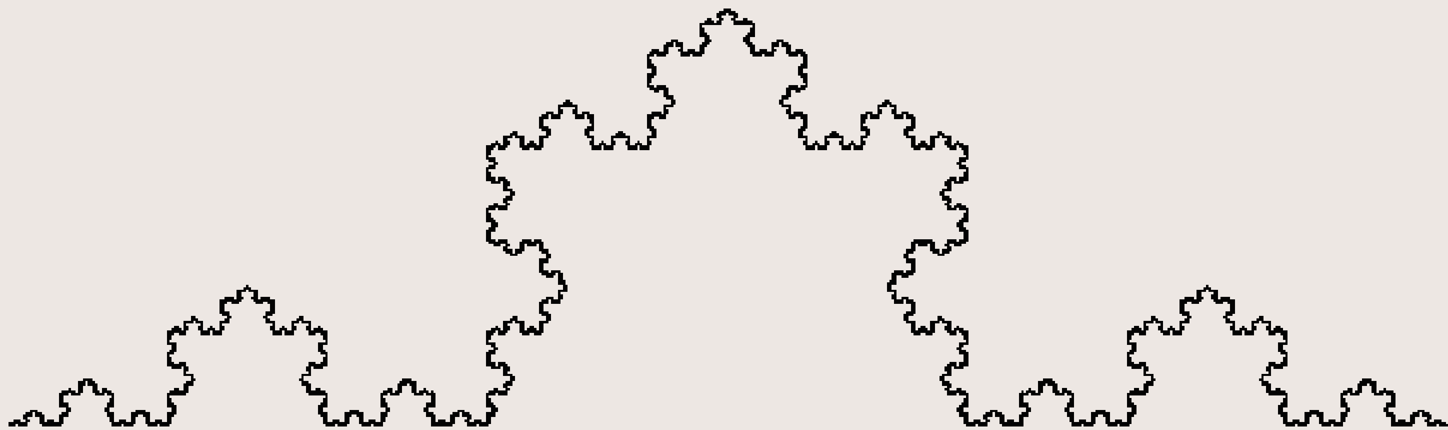
Рекурсивният алгоритъм се описва така:

1. Линията се дели на три сегмента с равна дължина.
2. Строи се равностранен триъгълник с основа средния сегмент.
3. Изтрива се сегментът, основа на триъгълника от т. 2.

Кривата на Кох е с безкрайна дължина.

Кривата на Кох е с площ 8/5 площта на стартовия триъгълник. Безкраен периметър огражда крайна площ.

Крива на Кох (Koch curve)



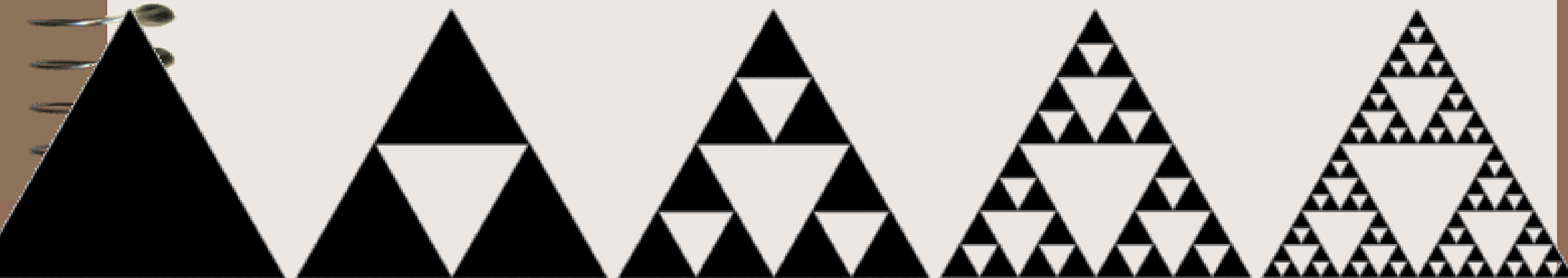
Триъгълник на Шерпински

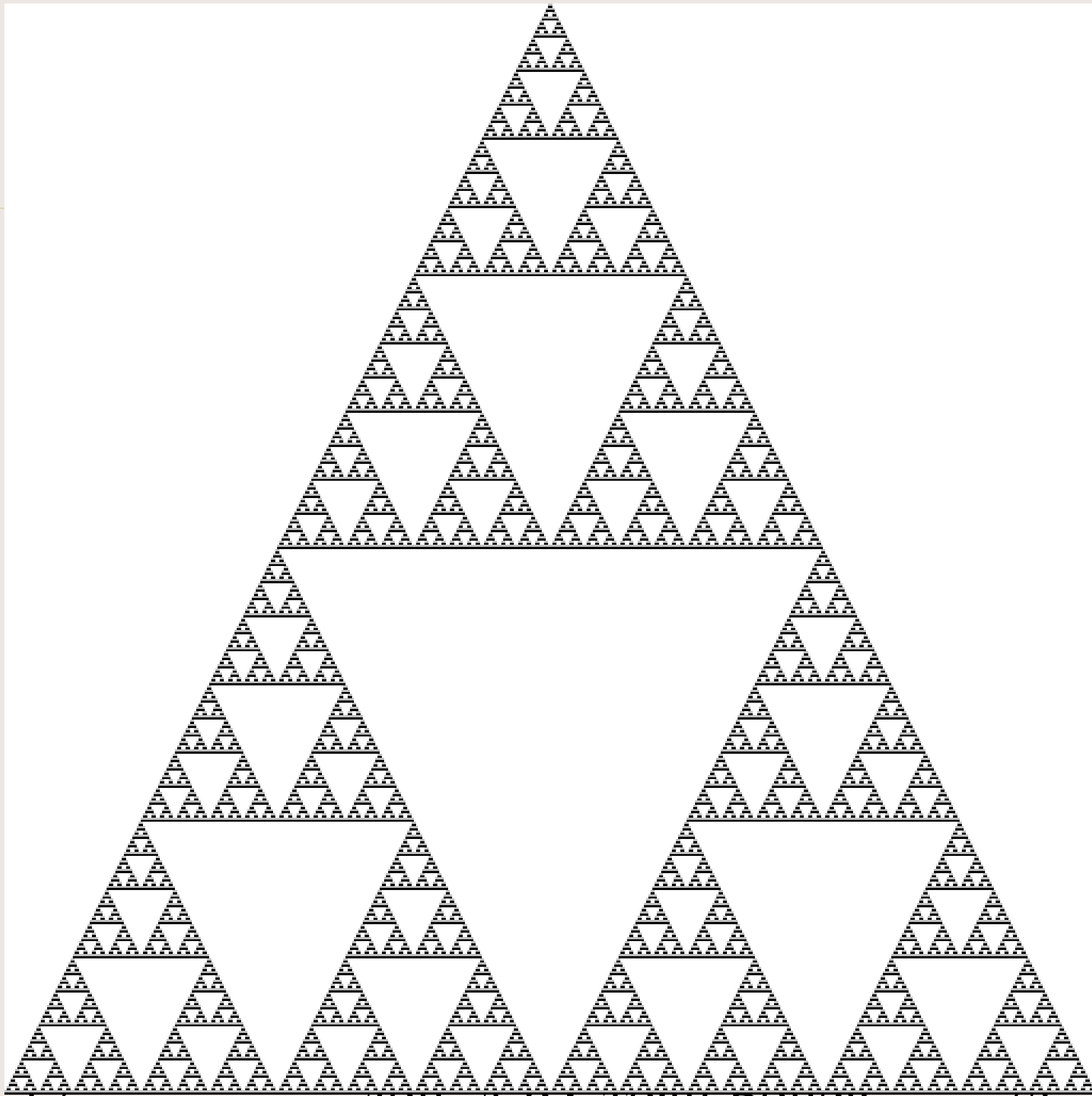
Рекурсивната природа на тази фигура следва от следния алгоритъм:

Начало: триъгълник в равнина. Каноничен Sierpinski triangle е равностранен триъгълник с основа успоредна на хоризонталната ос.

1. Триъгълникът се дели на 1/2, получават се три нови триъгълници и те се разполагат така, че всеки триъгълник допира другите два във върховете (виж следния слайд).
2. Стъпка 1 се повтаря с всеки един от създадените по-малки триъгълници.

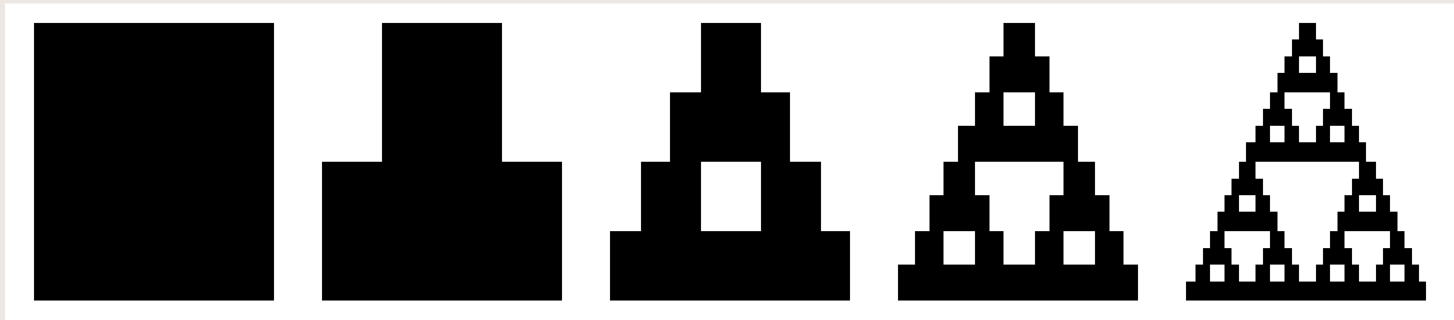
Триъгълник на Шерпински





Триъгълник на Шерпински

Забележка: Описаният безкраен рекурсивен алгоритъм не зависи от формата на началната фигура. Следният слайд представя Sierpinski gasket с начална фигура квадрат.



Множество на Кантор

Множеството на Кантор (**Cantor set)** обхваща реални числа (real numbers) в интервала [0, 1].

Множеството се определя чрез отстраняване на средната третина от линейни сегменти. Започва се с елиминиране на средната третина на интервала [0, 1], като остават под интервалите [0, 1/3] и [2/3, 1]. Следва отстраняване на “средните третини" от оставените за обработка интервали. Този процес е безкраен (continued ad infinitum). Множеството на Кантор е безкрайно и съдържа всички точки в интервала [0, 1], които не са отстранени по време на описаното безкрайно делене на интервала [0, 1] на три.

Първите 6 стъпки на този процес са показани на слайда.

Множество на Кантор

Рекурсия в реалния живот

- Представете си зала с огледала на две от срещуположните стени

Рекурсия в естествения език

Тези примери служат повече като закачки (jokes), които развиват интуицията и разбирането за рекурсия.

Ние знаем.

Той знае, че ние знаем.

Ние знаем, че Той знае, че Ние знаем.

Той знае, че Ние знаем, че Той знае, че Ние знаем.

и т.н. ....

Рекурсия в математиката

- **Functional Recursion**
- Предполага функция, която вика себе си многократно до достигане на гранично условие (end state, base case, simple case). Всяко извикване увеличава дълбочината на обръщенията, които спират при достигане граничното условие. Две фактора са важни:

1. Всяко следващо извикване на функцията е с по-малък набор от аргументи (smaller subset of the values with which it began).
2. Функцията има гранично условие, което прекъсва (терминира) процеса (веригата) от рекурсивни обръщения.

Рекурсия в програмирането

- Рекурсия в програмирането дефинира ПП функция в термините на самата функция (извикване на самата себе си).
- Рекурсивната функция следва да има заложено гранично условие (един или няколко основни случаи base cases), както и правила за привеждане (свеждане) на общите случаи до граничното условие.
- Този тип рекурсия е аналитична рекурсия.
- Примерът с естествените числа илюстрира генеративна (синтетична) рекурсия.

Рекурсия

- **Рекурсия** се изразява чрез оператор за обръщение към ПП. Рекурсивна функция е функция, която извиква себе си.

За разлика от итерацията, Синтаксисът на ПЕ предлага само един начин за изразяване на рекурсия: механизъм за предаване на у-ние на ПП (function, procedure) – оператор call. Дефиницията присвоява име на ПП, с което се позволява многократно извикване както отвън (други ПП), така и отвътре (самата ПП).

Рекурсивни алгоритми

- **Гранично условие (Simple case)**: Това е пряко, недвусмислено решение.
- Рекурсивните алгоритми стандартно съдържат оператор *if* в сл. форма:

if (<this is simple case>) <solve it>;

else <redefine problem using recursion>;

Each recursive algorithm needs a simple case. It is also called a boundary condition that is used to stop, to interrupt, to finalize or to break the recursive calls sequence.

V

- Локални променливи в рекурсивни функции:
  - Еднакви имена
  - Различни клетки от паметта
  - Отделни ”слоеве”

Рекурсия (примери)

Условен израз по McCarthy:

Означение:

$$[\ b_1 > e_1,\ b_2 > e_2,\ \dots,\ b_n > e_n,\ e_{n+1}\ ]$$

където

$b_1, b_2, \dots, b_n$	Булев израз(и)
$e_1, e_2, \dots, e_n, e_{n+1}$	Израз(и)

Factorial

- Типичен пример на рекурсивна функция е функцията, която връща factorial на своя integer аргумент:

- Factorial(n) = n * Factorial(n-1), n<>0
- Factorial(0) = 1

Factorial(n) = [(n=0) > 1, n*Factorial(n-1)]

Greatest common divisor

- Най-голям общ делител на две цели положителни стойности

- GCD(m, n) = GCD(n, m % n), n<>0
- GCD(m, 0) = m

GCD(m, n) = [(n=0) > m, GCD(n, m%n)]

Редица на Фибоначи 1, 1, 2, 3, 5,...

- Fib(n) = Fib(n-1) + Fib(n-2), n>=2
- Fib(0) = 1
- Fib(1) = 1

Fib(n) = [(n=0)>1, (n=1)>1, Fib(n-1)+Fib(n-2)]

Редица на Фибоначи 1, 1, 2, 3, 5,...

- Fibonacci (1170–1250) открива безкрайната редица, която е тясно свързана със златното сечение (1.618033989).
- Още от времето на Renaissance, художници и архитекти създават своите творби в пропорция на златно сечение, т.нар. Златен правоъгълник **golden rectangle**, страните на който са в релация златно сечение, с вярата, че създават предпоставки за естетическо и хармонично възприемане на художествените творби.
- http://www.answers.com/Golden%20Ratio

Видове рекурсия

- **Рекурсивно определение - Recursive definition (primitive recursion)**: Извикване на функцията се среща в тялото на функцията.
  - Пример: факториел, НОД, Фибоначи, ...

- **Рекурсивно позоваване - Recursive reference (referenced recursion)**: Извикване на функцията се среща като аргумент в списъка на формалните параметри.
  - Пример: двоен интеграл $I_2 = I_1(a,b,I_1(c,d,G))$

Функция на Ackerman

- Това е функция, която илюстрира и двата вида – рекурсивно определение и рекурсивно позоваване.
- $$A(m,n) = \begin{cases} n+1, & m=0 \\ A(m-1, 1), & n=0 \\ A(m-1, A(m, n-1)) & \end{cases}$$

Задача: Дефинирайте горната функция с условен израз по McCarthy

Видове рекурсия

- Пряка(директна) и непряка(косвена) рекурсия
- Дълбочина на рекурсия – брой на рекурсивни обръщения за конкретен аргумент (крайна стойност).
- Безкрайна по дълбочина рекурсия се нарича регресия.

Видове рекурсия

Рекурсивни структури от данни:

- двоично дърво;
- списък
- граматика на аритметични изрази

Сравнение по памет и време

Обектен код, размер в байтове

	Факториел	НОД	Фибоначи
Итерация	407	434	567
Рекурсия	387	406	426

Бързодействие, мсек

	Факториел	НОД	Фибоначи
Обраб. Ст.	10 000	20 000	30
Итерация	770	220	<10
Рекурсия	2200	380	17470

Итерация с/у Рекурсия

- Итерация заема повече памет.
- Рекурсия работи по-бавно.
- Не всички рекурсивни алгоритми са прозрачни (bad readability).

  Трудности при локализация на грешки, тестиране и настройка.

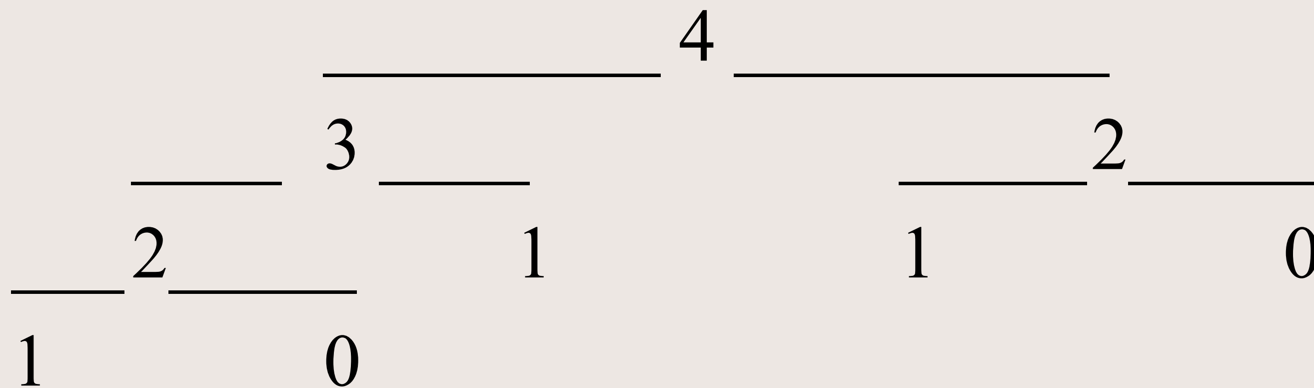
Ниското бързодействие при Фибоначи - защо?

Fib(n) = Fib(n-1) + Fib(n-2), n>=2

За пресмятане на Fib(n), са нужни 2 обръщения.

За Fib(n-1), са нужни 2 обръщения.

За Fib(n-2), са нужни 2 обръщения.



$Fib(n) = Fib(n-1) + Fib(n-2)$

nmr(n) – колко пъти се вика функцията за пресмятане на Fib(n)

n	0	1	2	3	4	5	6
nmr(n)	1	1	3	5	9	15	25

$nmr(n) = nmr(n-1) + nmr(n-2) + 1$

$nmr(0) = 1$

$nmr(1) = 1$

Подобрено бързодействие при Фибоначи

Функция без подобрения:

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

Подобрено бързодействие при Фибоначи

Таблица съхранява вече пресметнати стойности

```
struct val { double value; bool filled; };
extern val tab[];
int fibr(int n) {
   if (n==0 || n==1) return n;
   if (tab[n].filled == false) {
         tab[n].value = fibr(n-1) + fibr(n-2);
         tab[n].filled  =  true; }
  return tab[n].value;
       }
```

Статистика за Фибоначи

	Обектен код размер, байтове	Бързодействие мсек
Итерация	567	<10
Рекурсия	426	17470
Подобрена рекурсия	575	<10

Нечетливи рекурсивни алгоритми

Анализирайте следния алгоритъм:

```
void printd(int n)
  {
   if (n<0) { putchar('-'); n=-n; }
   if (n/10)  printd(n/10);
   putchar(n%10 + '0');
  }
```

Първични текстове на итеративни и рекурсивни алгоритми

- Факториел
- НОД – най-голям общ делител
- Редица на Фибоначи
- Квадратен корен
- Сума на естествени числа
- Сума на елементите на едномерен масив
- F(n) и функция на Ackerman
- Функция main() извиква себе си
- Моделиране на аритм операции +, -, *, /, ^

Factorial in C, C++, Java, C#

```
#include <iostream>
using namespace std;
int facti(int n);
int factr(int n);
void main()    {
   // Test Factoriel recursive and iterative versions
   for(int i=0;i<=10;i++)
   cout <<"\n" << i << "  ->  " << facti(i) << "  ->  " << 
   factr(i);
}

int facti(int n)      {      if (n==0 || n==1) return 1;
   int i, res=1;
   for (i=2; i<=n; i++) res *= i;
   return res;
}

int factr(int n)      {      if (n==0 || n==1) return 1;
   return n * factr(n-1);
}
```

Factorial in Java

```
public class ProgIterationRecursion
  { public static void main(String args[]) {
     for(int i=0;i<=10;i++)
System.out.println(" "+i+" "+facti(i)+" "+factr(i));
        } // end of main

  static int facti(int n)     {
     if (n==0 || n==1) return 1;
     int i, res=1;
     for (i=2; i<=n; i++) res *= i;
     return res;
     }
  static int factr(int n) {
     if (n==0 || n==1) return 1;
     return n * factr(n-1);
     }
} // end of class
```

Factorial in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace IterationRecursion
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i <= 10; i++)
            Console.WriteLine(" "+i+"    "+facti(i)+"    "+factr(i));
 } // end of Main
        static int facti(int n)
        {
    if (n==0 || n==1) return 1;
    int i, res=1;
    for (i=2; i<=n; i++) res *= i;
    return res;
    }
    static int factr(int n)
        {
    if (n==0 || n==1) return 1;
    return n * factr(n-1);
   }
    } // end of main class Program
} // end of namespace
```

Factorial in Perl

```
#!/usr/bin/perl

sub  Facti {   $Arg1 = shift(@_);
        if($Arg1 == 0) {return(1);}
        $Result = 1;
        for ($i=1;$i<=$Arg1;$i++) { $Result = $Result * $i; }
        return($Result);
        }

sub  Factr {   my($Arg1) = shift(@_);
        if($Arg1 == 0) {return(1);}
        else {return($Arg1 * &Factr($Arg1-1)); }
        }

# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
for($k=0; $k<=10; $k++)
{print "\n".$k."  >>  ".&Facti($k)."  -  ".&Factr($k);}
```

Factorial in Python

```
def facti(n):
   i, res = 1, 1
   while i <= n:
       res, i = res*i, i+1
   return res

def factr(n):
   if n==0:   return 1
   else:            return n * factr(n-1)

print "Python greeting Hello, world!"
listnum = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
print " "
print "Factorial recursive and iterative versions"
for k in listnum: print k, factr(k), facti(k)
```

GCD in C, C++, Java, C#

```
int gcdi(int m, int n)
{
  if(n==0) return m;
  int r;
  while ( (r=m%n)!=0 ) {   m=n;         n=r;
  }
  return n;
}

int gcdr(int m, int n)
{
  if(n==0) return m;
  return gcdr(n, m%n);
}
```

GCD in C, C++, Java, C#

```
// version C
for(i=0; i<=10; i++)
  printf("\n10,%d -> %d -> %d", i, gcdi(10,i), gcdr(10,i));

// version C++
for(int i=0;i<=10;i++)
   cout <<"\n10," <<i<< " -> "<<gcdi(10,i) << " -> " << gcdr(10,i);

// version Java
 for(int i=0;i<=10;i++)
   System.out.println("10,"+i+" -> "+gcdi(10,i)+" -> "+gcdr(10,i));

// version C#
for(int i=0;i<=10;i++)
  Console.WriteLine("10,"+i+" -> "+gcdi(10, i)+" -> "+gcdr(10,i));
```

GCD in Perl

```
#!/usr/bin/perl
sub  GCDi {        $Arg1 = shift(@_); $Arg2 = shift(@_);
            if($Arg2 == 0) {return($Arg1);}
            while( ($Rem=($Arg1%$Arg2)) != 0)
                        {
                         $Arg1 = $Arg2;
                         $Arg2 = $Rem;
                        }
            return($Arg2);
            }
sub  GCDr {        my($Arg1) = shift(@_);  my($Arg2) = shift(@_);
             if($Arg2 == 0) {return($Arg1);}
            else {return(&GCDr($Arg2, $Arg1%$Arg2)); }
            }
```

GCD in Perl

```
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nGCD";
for($l=0; $l<=10; $l++)
{print "\n10,".$l."  >>  ".&GCDi(10, $l)."  -  ".&GCDr(10, $l);}
```

GCD in Python

```
def gcdi(m, n):
    if n==0: return m
    rem = m % n
    while rem != 0:
        m, n, rem = n, rem, m % n
    return n

def gcdr(m, n):
    if n==0:      return m
    else:         return gcdr(n, m%n)
```

GCD in Python

```
print "Python greeting Hello, world!"
listnum = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
print " "
print "GCD recursive and iterative versions"
for k in listnum: print 10, k, gcdr(10, k), gcdi(10, k)
```

Sum Σi in C, C++, Java, C#

```
int sumi(int n)                    // Sum of natural numbers
{
    if (n==0 || n==1) return n;
    int i, sum = 0;
    for (i=1; i<=n; i++) sum+= i;
    return sum;
}
int sumr(int n)
{
    if (n==0 || n==1) return n;
    return n + sumr(n-1);
}
```

Sum Σi in C, C++, Java, C#

```
// version C
for(i=0; i<=10; i++)
    printf("\n%d  ->  %d  ->  %d", i, sumi(i), sumr(i));

// version C++
for(int i=0;i<=10;i++)
     cout <<"\n" << i << "  ->  " << sumi(i) << "  ->  " << sumr(i);

// version Java
 for(int i=0;i<=10;i++)
            System.out.println(" "+i+"  ->  "+sumi(i)+"  ->  "+sumr(i));

// version C#
for(int i=0;i<=10;i++)
     Console.WriteLine(" " + i + "  ->  " + sumi(i) + "  ->  " + sumr(i));
```

Sum Σi in Perl

```
#!/usr/bin/perl
sub  sumi {       $Arg1 = shift(@_);
            if($Arg1==0 || $Arg1==1) {return($Arg1);}
            $Sum = 0;
            for ($i=1; $i <= $Arg1 ; $i++) {$Sum += $i;}
            return($Sum);
            }

sub  sumr {       my($Arg1) = shift(@_);
            if($Arg1==0 || $Arg1==1) {return($Arg1);}
            return($Arg1 + &sumr($Arg1-1));
            }
```

Sum Σi in Perl

```
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nSum of natural numbers";
for($l=0; $l<=10; $l++)
{print "\n".$l."  >>  ".&sumi($l)."  -  ".&sumr($l);}
```

Sum Σi in Python

```
# Sum of natural numbers
def sumi(n):
    if n==0:   return n
    i, sum = 1, 0
    while i <= n:
        sum, i = sum + i, i+1
    return sum

def sumr(n):
    if n==0:   return n
    else:              return n + sumr(n-1)
```

Sum Σi in Python

```
print "Python greeting Hello, world!"
listnum = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
print " "
print "Sum natural numbers recursive and iterative versions"
for k in listnum: print k, sumr(k), sumi(k)
```

Sum Σa[I] in C++

```
// Sum of 1-dimensioned array elements
int ar1sumi(int b[], int n)
{
    int i, sum=0;
    for (i=0; i<=n; i++) sum += b[i];
    return sum;
}
int ar1sumr(int b[], int n)
{
    if (n==0) return b[0];
    return b[n] + ar1sumr(b, n-1);
}
```

Sum Σa[I] in C++

```
#include <iostream>
using namespace std;
// Sum of 1-dmensioned array elements
int ar1sumi(int b[], int n);
int ar1sumr(int b[], int n);
int main()
{
    cout <<"CPP Greeting Hello, World";
// Test Sum of 1-dimensioned array elements
    cout <<"\n\nSum of 1-dimenisioned array elements\n";
    int mas1[] = { 5, 10, 20, 30, 40, 50, 60};
    for (int i=0; i<=6; i++) cout << mas1[i] << "  ";
    cout << "  >  " <<  ar1sumi(mas1, 6) << "  >  " << ar1sumr(mas1, 6);
}
```

Sum Σa[I] in Perl

```
sub  ar1sumi{  @ArgList = @_;  $sum = 0;
        for ($i=0;$i<=$#ArgList;$i++)
            {$sum+=$ArgList[$i];}
        return($sum);
        }


sub  ar1sumr{  my(@ArgList) = @_;
   my($Index)=$#ArgList;
        if ($Index==0) {return ($ArgList[$Index]);}
        return ($ArgList[0]+ar1sumr(@ArgList[1..$Index]));
        }
```

Sum Σa[I] in Perl

```
print "\nSum of 1-dimensioned array elements  ";
@mas = ( 5, 10, 20, 30, 40, 50, 60 );
for ($i=0; $i<=$#mas; $i++) { print $mas[$i]."  ";}
print "\nIterative sum of an array/list = ".ar1sumi(@mas);
print "\nRecursive sum of an array/list = ".ar1sumr(@mas);
getc();
```

Fibonacci series in C, C++, Java, C#

```
int fibi(int n)
{
   int i, va=1, vb=1, vc=1;
   if (n==0 || n==1) return 1;
   for (i=2; i<=n; i++)
   { vc = va + vb; va = vb; vb = vc; }
   return vc;
}

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

Fibonacci series in C, C++, Java, C#

```
// version C
for(i=0; i<=10; i++)
      printf("\n%d  ->  %d  ->  %d", i,  fibi(i),  fibr(i));

// version C++
for(int i=0;i<=10;i++)
     cout <<"\n" << i << "  ->  " << fibi(i) << "  ->  " << fibr(i);

// version Java
 for(int i=0;i<=10;i++)
     System.out.println(" "+i+"  ->  "+fibi(i)+"  ->  "+fibr(i));

// version C#
for(int i=0;i<=10;i++)
     Console.WriteLine(" " + i + "  ->  " + fibi(i) + "  ->  " + fibr(i));
```

Fibonacci series in Perl

```
#!/usr/bin/perl
sub  fibi {           $Arg1 = shift(@_);
           if($Arg1==0 || $Arg1==1) {return($Arg1);}
           $va=0; $vb=1;
           for ($i=2; $i <= $Arg1 ; $i++)
                      {$vc = $va+$vb; $va = $vb; $vb = $vc;}
           return($vc);
           }
sub  fibr {           my($Arg1) = shift(@_);
           if($Arg1==0 || $Arg1==1) {return($Arg1);}
           return( &fibr($Arg1-1) + &fibr($Arg1-2) );
           }
```

Fibonacci series in Perl

```
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nFibonacci";
for($l=0; $l<=10; $l++)
{print "\n".$l."  >>  ".&fibi($l)."  -  ".&fibr($l);}
getc();
```

Fibonacci series in Python

```
# Fibonacci series module
def fibi(n):        # Fibonacci series up to n
    va, vb, i = 0, 1, 2
    while i <= n:
        va, vb, i = vb, va+vb, i+1
    return vb

def fibr(n):
    if n==0:        return n
    elif n==1:      return n
    else:           return fibr(n-1) + fibr(n-2)
```

Fibonacci series in Python

```
print "Python greeting Hello, world!"

listnum = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
print "Fibonacci recursive and iterative versions"
for k in listnum: print k, fibr(k), fibi(k)
#read (ch)
```

Square root in C, C++, Java, C#

```
// C,C++: abs( )          Java: Math.abs( )          C#: Math.Abs( )
double sqrti(double x, double eps)
{
    double yn=1, yn1=(x/yn+yn)/2;
    while (abs(yn-yn1)>eps) {yn=yn1; yn1=(x/yn+yn)/2;}
    return yn;
}
double sqrtr(double x, double start, double eps)
{
    if (abs(start*start-x) < eps) return start;
    return sqrtr(x, (start*start +x)/(2*start), eps);
}
```

Square root in C, C++, Java, C#

```
// version C
    printf("\n\nSquare root simulations");
    printf("\n 25  ->  %lf  ->  %lf", sqrti( 25,0.001), sqrtr( 25,1,0.001));
    printf("\n144  ->  %lf  ->  %lf", sqrti(144,0.001), sqrtr(144,1,0.001));
    printf("\n289  ->  %lf  ->  %lf", sqrti(289,0.001), sqrtr(289,1,0.001));
// version C++
    cout<<"\n"<< 25<<" -> "<<sqrti(25,0.001)<<" -> "<< sqrtr(25,1,0.001);
    cout<<"\n"<<144<<" -> "<<sqrti(144,0.001)<<" -> "<< sqrtr(144,1,0.001);
    cout <<"\n"<<289<<" -> "<<sqrti(289,0.001)<<" -> "<< sqrtr(289,1,0.001);
// version Java
  System.out.println("25  -> "+sqrti(25,0.001) +" -> "+sqrtr(25,1,0.001));
  System.out.println("144 -> "+sqrti(144,0.001)+" -> "+sqrtr(144,1,0.001));
  System.out.println("289 -> "+sqrti(289,0.001)+" -> "+sqrtr(289,1,0.001));
// version C#
  Console.WriteLine("25  -> "+sqrti(25,0.001) +" -> "+sqrtr(25,1,0.001));
  Console.WriteLine("144 -> "+sqrti(144,0.001)+" -> "+sqrtr(144,1,0.001));
  Console.WriteLine("289 -> "+sqrti(289, 0.001)+" -> "+sqrtr(289, 1, 0.001));
```

Square root in Perl

```
#!/usr/bin/perl
sub  sqrti {            $Arg1 = shift(@_); $Arg2 = shift(@_);
            $yn=1.;     $yn1=($Arg1/$yn + $yn)/2.;
            while( abs($yn-$yn1) > $Arg2)
                        {
                         $yn =$yn1;
                         $yn1=($Arg1/$yn + $yn)/2.
                        }
            return($yn);
            }
sub  sqrtr {            my($Arg1) = shift(@_);       my($Arg2) = shift(@_);
            my($Arg3) = shift(@_);
            if(abs($Arg2*$Arg2-$Arg1)<$Arg3) {return($Arg2);}
            return(&sqrtr($Arg1,($Arg2*$Arg2+$Arg1)/(2.*$Arg2),$Arg3));
            }
```

Square root in Perl

```
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nSquare root simulations";
print "\n 25  >>  ".&sqrti( 25,0.001)."  -  ".&sqrtr( 25,1.,0.001);
print "\n144  >>  ".&sqrti(144,0.001)."  -  ".&sqrtr(144,1.,0.001);
print "\n289  >>  ".&sqrti(289,0.001)."  -  ".&sqrtr(289,1.,0.001);
getc();
```

Square root in Python

```
# square root simulations
def sqrti(x, eps):
    yn  = 1
    yn1 = (x/yn + yn)/2
    while abs(yn-yn1)>eps:
        yn  = yn1
        yn1 = (x/yn+yn)/2
    return yn
def sqrtr(x, yn, eps):
    if abs(yn*yn-x)<eps:    return yn
    else:                   return sqrtr(x, (yn*yn + x)/(2*yn), eps)
```

Square root in Python

```
print " "
print "SQRT recursive and iterative versions"
listx = [ 25, 49, 81, 100, 289 ]
for k in listx: print k, sqrtr(k, 1, 0.001), sqrti(k, 0.001)
```

Последователно търсене – C++

```
// Iterative Linear Search
int LinSearchIterative(int target, int table[], int size) {
  int  k = 0;
  while ( k < size )   { if (target == table[k])  return k;
                          k++;  }
  return  -1;
}

// Recursive linear search
int LinSearchRecursive(int target, int table[], int low, int high)
{
  if (high < low) return -1;
  if (target == table[high]) return high;
  return LinSearchRecursive(target, table, low, high-1);
}
```

Двоично търсене – C++

```
// Iterative Binary search
int BinSearchIterative(int target, int table[], int size)
{
  int  low,  high,  mid;  low  =  0;  high  =  size - 1;
  while  ( low <= high )
  {
     mid = (low + high) / 2;
     if (target == table[mid])  return  mid;
     if (target  < table[mid])  high  =  mid - 1;
     if (target  > table[mid])  low = mid + 1;
  }
  return  -1;
}
```

Двоично търсене – C++

```
// Recursive Binary search
int BinSearchRecursive(int target, int table[], int low, int high)
{
  if (low > high) return -1;
  int middle;       middle = (low + high) / 2;
  if (target == table[middle]) return middle;
  if (target <  table[middle]) // low half of the table
    {
          // high = middle-1;
         return BinSearchRecursive(target, table, low, middle-1);
    }
  if (target >  table[middle]) // high half of the table
    {
          // low = middle+1;
           return BinSearchRecursive(target, table, middle+1, high);
    }
}
```

Операция (*) - C/C++

```
int muli(int m, int n)
{
   if (n==0) return 0;
   if (n==1) return m;
   int i, res=0;
   for (i=1; i<=n; i++) { res = res + m;}
   return res;
}

int mulr(int m, int n)
{
   if (n==0) return 0;
   if (n==1) return m;
   return m + mulr(m, n-1);
}
```

Операция (*) - C++

```
#include <iostream>
using namespace std;
int muli(int m, int n);
int mulr(int m, int n);
void main()
{
    // Test multiplication as recursive and iterative routine
    cout <<"\nMultiplication as Recursion and Iteration";
    for (int k=0; k<=12; k++)  {        cout <<"\n";
                                         for (int l=0; l<=12; l++)
    cout <<"\n"<<k<<","<<l<<"  ->  "<<muli(k,l)<<"  ->  "<<mulr(k,l);}
}
```

Операция (*) - Perl

```
#!/usr/bin/perl
sub  muli {          $Arg1 = shift(@_); $Arg2 = shift(@_);
            if($Arg2==0)      { return 0;}
            if($Arg2==1)      { return $Arg1;}
            $res = 0;
            for ($i=1; $i<=$Arg2; $i++) { $res = $res + $Arg1; }
            return $res;
            }
sub  mulr {          $Arg1 = shift(@_); $Arg2 = shift(@_);
            if($Arg2==0)      { return 0;}
            if($Arg2==1)      { return $Arg1;}
            return $Arg1 + &mulr($Arg1, $Arg2-1);
            }
```

Операция (*) - Perl

```
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
# Test for MULTIPLICATION based on Recursion and Iteration
print "\nMultiplication implemented using Recursion and Iteration";
for($k=0; $k<=12; $k++)
{
    print "\n";
    for($l=0; $l<=12; $l++)
    {
     print "\n".$k."  ".$l."  ".&muli($l, $k)."  ".&mulr($l, $k);
    }
}
getc();
```

Операция (*) - Python

```
# Recursive and iterative simulation of MULTIPLICATION
def muli(m, n):
    if n==0:      return 0
    if n==1:      return m
    res, i = 0, 1
    while i<=n: res, i = res+m, i+1
    return res

def mulr(m, n):
    if n==0:      return 0
    if n==1:      return m
    return m + mulr(m,n-1)
```

Операция (*) - Python

```
print "Python greeting Hello, world!"

listnum = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
print " "
print "Multiplication impl. using Recursion and Iteration"
for k in listnum:
    print " "
    for l in listx:
            print k, l, mulr(k, l), muli(k, l)
```

- Проблеми, решени само във версия Рекурсия
- Решение версия итерация е трудно и непрозрачно

Ackermann F. in C, C++, Java, C#

```
// Ackerman function
int acker(int m, int n)
{
    if(m==0) return n+1;
    if(n==0) return acker(m-1,1);
    return acker(m-1, acker(m,n-1));
}
```

Ackermann F. in C, C++, Java, C#

```
// version C
    printf("\nAckerman Arguments 1,1  >  %d", acker(1,1));
    printf("\nAckerman Arguments 2,1  >  %d", acker(2,1));
    printf("\nAckerman Arguments 0,51 >  %d", acker(0,51));
// version C++
    cout <<"\nAckerman Arguments 1 1  >  " << acker(1,1);
    cout <<"\nAckerman Arguments 2 1  >  " << acker(2,1);
    cout <<"\nAckerman Arguments 0 51 >  " << acker(0,51);
// version Java
  System.out.println("Ackerman Arguments 1,1  >  " + acker(1,1));
  System.out.println("Ackerman Arguments 2,1  >  " + acker(2,1));
  System.out.println("Ackerman Arguments 0,51 >  " + acker(0,51));
// version C#
  Console.WriteLine("Ackerman Arguments 1,1  >  " + acker(1,1));
  Console.WriteLine("Ackerman Arguments 2,1  >  " + acker(2,1));
  Console.WriteLine("Ackerman Arguments 0,51 >  " + acker(0, 51));
```

Ackermann Ф. на Perl

```
#!/usr/bin/perl
sub  acker {         my($Arg1) = shift(@_); my($Arg2) = shift(@_);
              if($Arg1==0) {return($Arg2 + 1);}
              if($Arg2==0) {return(&acker($Arg1-1, 1));}
              return(&acker($Arg1-1, &acker($Arg1, $Arg2-1)));
              }
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nAckerman function acker(m,n)";
print "\nAckerman Arguments 0,0  >  ".&acker(0,0);
print "\nAckerman Arguments 1,1  >  ".&acker(1,1);
print "\nAckerman Arguments 2,1  >  ".&acker(2,1);
print "\nAckerman Arguments 0,51 >  ".&acker(0,51);
getc();
```

Ackermann Ф. на Python

```
# Ackerman function
def acker(m, n):
    if m==0:     return n+1
    if n==0:     return acker(m-1, 1)
    else:        return acker(m-1, acker(m, n-1))
print "Python greeting Hello, world!"
print " "
print "Ackerman function recursive version"
print acker(0, 0), acker(1, 1), acker(2, 1), acker(0,51)
```

Ханойски кули - C/C++

```
// Hanoi Towers game
void trantower(int ndsks, int fromndl, int tondl, int wrkndl)
{
    if (ndsks > 0)
    {
        trantower(ndsks-1, fromndl, wrkndl, tondl);
        printf("\n%2d  >>  %2d", fromndl, tondl);
        trantower(ndsks-1, wrkndl, tondl, fromndl);
    }
}
```

Ханойски кули - C

```
#include <stdio.h>
void trantower(int ndsks, int fromndl, int tondl, int wrkndl);
void main()
{
    printf("C Greeting Hello, World");
    // Test Hanoi Towers game recursive version
    printf("\n\nHanoi towers game - 1 disk");
    trantower(1, 1, 3, 2);
    printf("\n\nHanoi towers game - 2 disks");
    trantower(2, 1, 3, 2);
    printf("\n\nHanoi towers game - 3 disks");
    trantower(3, 1, 3, 2);
}
```

Ханойски кули - C++

```
#include <iostream>
using namespace std;
void trantower(int ndsks, int fromndl, int tondl, int wrkndl);
int main()
{
    // Test Hanoi Towers game recursive version
    cout <<"\n\nHanoi towers game - 1 disk";
    trantower(1, 1, 3, 2);
    cout <<"\n\nHanoi towers game - 2 disks";
    trantower(2, 1, 3, 2);
    cout <<"\n\nHanoi towers game - 3 disks";
    trantower(3, 1, 3, 2);
}
```

Ханойски кули - Perl

```
#!/usr/bin/perl
sub  trantower{     my($Arg1) = shift(@_);       my($Arg2) = shift(@_);
              my($Arg3) = shift(@_);  my($Arg4) = shift(@_);
              if($Arg1 >= 1) {
                                  trantower($Arg1-1,$Arg2,$Arg4,$Arg3);
                                  print "\n".$Arg2."  >>  ".$Arg3;
                                  trantower($Arg1-1,$Arg4,$Arg3,$Arg2);
                                                }
              }
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\n\nHanoi towers game - 1 disk "; &trantower(1, 1, 3, 2);
print "\n\nHanoi towers game - 2 disks"; &trantower(2, 1, 3, 2);
print "\n\nHanoi towers game - 3 disks"; &trantower(3, 1, 3, 2);
getc();
```

Ханойски кули - Python

```
# Hanoi towers game
def trantower(ndsks, fromndl, tondl, wrkndl):
    if ndsks>=1:
            trantower(ndsks-1, fromndl, wrkndl, tondl)
            print fromndl, "  >>  ", tondl
            trantower(ndsks-1, wrkndl, tondl, fromndl)
print "Python greeting Hello, world!"
print " "
print "Hanoi towers game - recursive function implemented"
trantower(1, 1, 3, 2)
print" "
trantower(2, 1, 3, 2)
print" "
trantower(3, 1, 3, 2)
```

itoa() моделирана на C

```
#include <stdio.h>
void printd(int n);
void main()
{
   // Test C/C++ itoa function recursive version
   printf("\n\nC/C++ itoa run time library function");
   printf("\n");  printd(123);
   printf("\n");  printd(-86);
}

// C/C++ itoa run time function simulated
void printd(int n)
{
   if (n<0) { putchar('-'); n = -n;}
   if(n/10 != 0) printd(n/10);
   putchar(n%10+'0');
}
```

itoa() моделирана на C++

```
#include <iostream>
using namespace std;
void printd(int n);
int main()
{
   // Test C/C++ itoa function recursive version
   cout <<"\n\nC/C++ itoa run time library function";
   cout << "\n";  printd(123);
   cout << "\n";  printd(-86);
}

void printd(int n)
{
   if (n<0) { cout << "-"; n = -n;}
   if(n/10 != 0) printd(n/10);
   cout << n%10;
}
```

itoa() моделирана на Perl

```
#!/usr/bin/perl
sub  printd {        my($Arg1) = shift(@_);
            if($Arg1<0){ print"-"; $Arg1 = -$Arg1; }
            if( ($Arg1/10) != 0) { &printd($Arg1/10); }
            if ($Arg1%10 != 0) {print chr($Arg1%10 + 48);}
            }
# start main program untitled unstructured
print "\nPerl greeting: Hello, world!\n";
print "\nC/C++ itoa run time library function";
print  "\n";  &printd(123);
print  "\n";  &printd(-86);
print  "\n";  &printd(5);
getc();
```

itoa() моделирана на Python

```
# C/C++ run-time function itoa recursively simulated
def printd(n):
    if n<0:                     print"-",
    if n<0:                     n = -n
    if n/10!=0:     printd(n/10)
    print n%10,
print "Python greeting Hello, world!"
print " "
print "The itoa run-time function recursively implemented as printd"
printd(123)
print " "
printd(-567890)
print " "
printd(86)
```

Траверс съдържание на директория

Всяка директория съдържа 2 типа структури:

- файлове
- поддиректории

Проблем: да се обходят и изведат имената на всички файлове и поддиректории в състава на указана директория – корен – стартов/начален аргумент

Решение на Java. Ползва клас File и следните методи:

- **`isDirectory()`**
- **`isFile()`**
- **`listFiles()`**

Траверс на директория

Извикване:

```
File fil = new File(“F:\\SBant”);
DirFilTraverse(fil);
```

Определение:

```
static void DirFilTraverse(File fl) {
if ( fl.isDirectory() ) {
   System.out.println(" Directory:"+fl.toString());
   File[] files = fl.listFiles();
   for (int i=0; i<files.length; i++)
   DirFilTraverse(files[i]);
  }
else if ( fl.isFile() )
      System.out.println(" File:"+fl.toString());
     else
      System.out.println(" No file, no directory");
}
```

Рекурсия и Хумор

- Да видим следната “дефиниция" на рекурсия.
  - **Recursion**
    - See "Recursion".
- Това е пародия на позоваване, което се среща в речници и енциклопедии. В примера по-горе няма гранично условие, или прост случай.

Част 2

Данни и тяхното вътрешно представяне

Съдържание

- **Данни** в ПЕ
- **Данни**
  – Константи;
  – Променливи.
- **Данни**
  – Основни типове данни;
  – Абстрактни (дефинирани от потребителя) типове данни.

Видове Данни:

- Константи
  - Основни атрибути: *Type, Address, Value.*
- Променливи
  - Основни атрибути: *Type, Address, Value, Name, Lifetime, Scope*.

Типове Данни:

- Първични - Primitive data types: Boolean, Symbolic, Numeric;
- Изброими - User defined ordinal data types;
- Масиви - Array data types;
- Структури - Record (structure) data types;
- Обединения - Union data types;
- Указатели - Pointer data types;
- Абстрактни - Abstract (class) data types.

Primitive Data Types

- Типове данни, които не се дефинират чрез други типове, са първични - *primitive data types*.
- Класификация:
  - Булеви – Boolean. Само две стойности – true/false или да/не, или 1/0.
  - Символни – единични символи или низове от символи
  - Числени – цели, фиксирана запетая
  - Числени – реални, плаваща запетая
  - Числени – десетични, бизнес приложения.

User-Defined Ordinal Types

- Изброим тип (*ordinal type)* е тип, при който диапазонът възможни стойности се асоциира с множество цели положителни числа. Две разновидности: *enumeration* and *subrange*.
- *Enumeration* тип: всички възможни стойности са именовани константи (named constants) и се изброяват в дефиницията.

  enum day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
- *subrange* тип: (contiguous subsequence of an ordinal type). *12..14* е подмножество (subrange) на целите данни. Пример от Pascal и Ada.

Structured Data Types

- *Масиви* – хомогенен агрегат от данни. Отделен елемент се адресира по индекс – отместване спрямо началото – базов адрес
- *Структури, Records* – хетерогенен агрегат от данни. Отделен елемент се адресира по име
- *Обединения, Unions* – на един и същ адрес в паметта по различно време се разполагат данни от различен тип.

Pointer Data Types

- Този тип променливи приемат стойности, които са адреси от паметта или специална стойност *nil* (*NULL*).
- *NULL* не е адрес, а индикатор, че указателят сочи никъде, т.е. не може да се ползва за косвен достъп до обект.

Символни данни

- **Външно представяне**
  - Константи *‘a’ “a”*
  - Променливи *char x, y[10], *ptr = “AUBG”;*
    string a, b=“AUBG”;
- **Вътрешно представяне.** Символи и низове се кодират като
  - ASCII – 1 byte/char
  - EBCDIC – 1 byte/char
  - UNICODE – 2 bytes/char.

Цели числени данни

- **Външно представяне**
  - Цели Константи
  - Цели Променливи
- **Вътрешно представяне**
  - Фиксирана запетая Fixed Point:
    - Знакови и без_знакови данни
    - Пряк код за положителни стойности
    - Допълнителен код за отрицателни стойности
    - Диапазон на данните

Цели числени данни

- **Външно представяне**
  - Цели константи

decimal	1234
octal	0157
hexadecimal	0x1f

31 = 037 = 0x1f

32 = 040 = 0X20 = ‘ ‘

1234		
1234l	12345678ul	
1234L	12345678Ul	0xful
1234U	12345678uL	0XFUL
1234u	12345678UL	

  - Цели променливи – запазени думи int, short, long, signed, unsigned

    *int a, b[10], c[10][20], *ptr;*

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни без знак при размер на поле 1 бит, или 2 бита, или 3 бита.
  - Диапазон на цели данни със знак при размер на поле 3 бита.
  - Диапазон на цели данни без знак при размер на поле n бита.
  - Диапазон на цели данни със знак при размер на поле n бита.

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни без знак при размер на поле 1 бит.
    - 1 bit: bit configurations:
      - 0
      - 1
    - Range 0..1 in decimal.

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни без знак при размер на поле 2 бита.
    - 2 bits: bit configurations:
      - 00
      - 01
      - 10
      - 11
    - Range 0..3 in decimal.

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни без знак при размер на поле 3 бита.
    - 3 bits: bit configurations:
      - 000
      - 001
      - 010
      - 011
      - 100
      - 101
      - 110
      - 111
    - Range 0..7 in decimal.

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни със знак при размер на поле 3 бита.
    - Bit configurations:
      - Sign Informative
      - 1 0 0
      - 1 0 1
      - 1 1 0
      - 1 1 1
      - 0 0 0
      - 0 0 1
      - 0 1 0
      - 0 1 1
    - Range: -4..+3 in decimal.

Цели числени данни

- **Вътрешно представяне**
  - Диапазон на цели данни без знак при размер на поле n бита.

    $0 .. 2^n - 1$
  - Диапазон на цели данни със знак при размер на поле n бита.

    $-2^{n-1} .. 2^{n-1} - 1$

Реални числени данни

- **Външно представяне**
  - Реални десетични Константи
  - Реални десетични Променливи
- **Вътрешно представяне**
  - Плаваща запетая Floating Point
    - Формат основан на стандарт IEE754.
    - Мантиса
    - Порядък
    - Диапазон и точност на представените данни.

Реални числени данни

- **Външно представяне**
  - Реални константи

    3.14159265 3.14e+2 3. .14159265

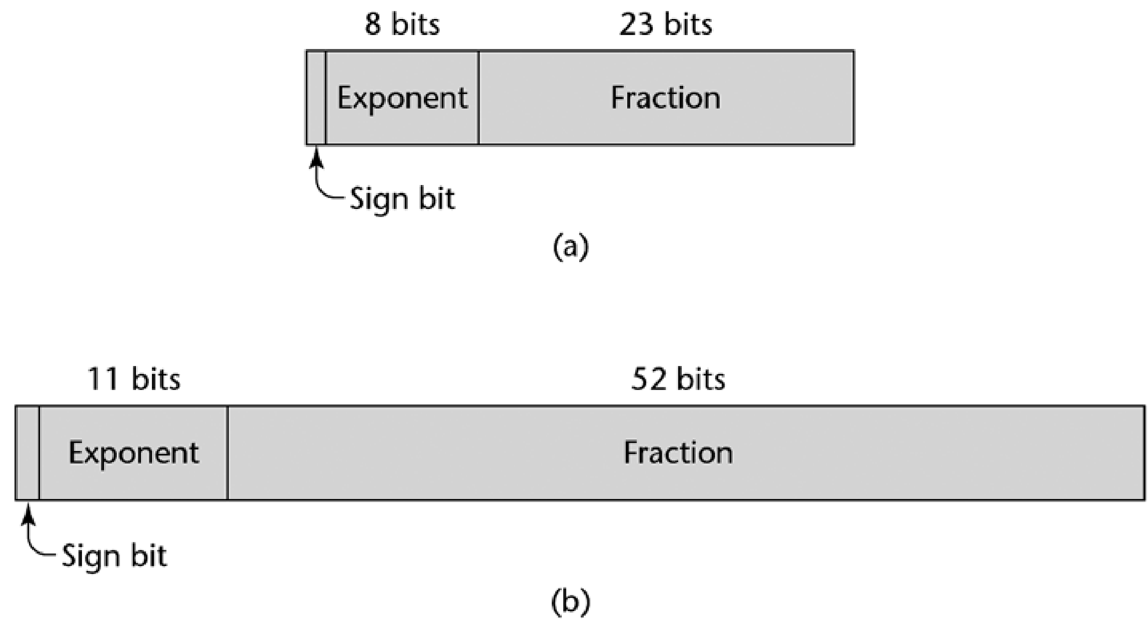
    123.4f float – single precision
    123.4F

    123.4 double – double precision

    123.4l long double – extended precision
    123.4L

  - Реални променливи – запазени думи *float, double, long double*.

Intel формат за *float* и *double*



IEEE 754 Стандарт

Две компоненти: порядък, мантиса.

Всеки компонент има 4 характеристики:

	порядък	мантиса
Бройна С-ма	2	2
знак	няма-offset	най-ляв бит
Точка (.) поз.	отдясно	отляво
стойност	бит конфиг.	бит конфиг.

IEEE 754 Стандарт

Мантисата е стойност в интервала

$$1.0 \le \text{mantissa} < \text{base}$$

Мантисата е нормализирана: най-левият бит е 1.

Стойността на реалното число под формат с плаваща запетая се изчислява по формулата:

$$\text{Value} = \text{mantissa} * (\text{base})^{\text{exp-offset}}$$

Коментари - плаваща запетая

- Размерът на мантисата влияе в/у точността на представените данни.
- Размерът на порядъка влияе в/у диапазона на представените данни.
- Конверсия от 10-на в 2-на бройна система води до загуба на точност.
- Разликата м/у реални числа (безкрайно множество) и FP числа (крайно множество)

Коментари - плаваща запетая

- Размерът на мантисата влияе в/у точността на представените данни.

- 1.234 1.2
- Example:
  - Exp = 3, mantissa с размер 2 бита, 4 разл. стойнсти

  - Exp = 3, mantissa с размер 3 бита, 8 разл. стойнсти

Коментари - плаваща запетая

- Размерът на мантисата влияе в/у точността на представените данни.

- 1.234 1.2
- Пример 1:
  - Exp = 3, mantissa с размер 2 бита, 4 разл. стойнсти

    1.xx * 2^3

    1.00 * 2^3 е равно = 8

    1.01 * 2^3 е равно =

    1.10 * 2^3 е равно =

    1.11 * 2^3 е равно =

Коментари - плаваща запетая

- Размерът на мантисата влияе в/у точността на представените данни.

- 1.234 1.2
- Example 1:
- Example 1:
  - Exp = 3, mantissa с размер 2 бита, 4 разл. стойнсти

    1.00 * 2^3 е равно = 8

    1.01 * 2^3 е равно = 10

    1.10 * 2^3 е равно = 12

    1.11 * 2^3 е равно = 14

Коментари - плаваща запетая

- Размерът на мантисата влияе в/у точността на представените данни.
- Пример 2:
  - Exp = 3, mantissa с размер 3 бита, 8 разл. стойнсти

    1.xxx * 2^3

    1.000 * 2^3 е равно = 8

    1.001 * 2^3 е равно = 9

    1.010 * 2^3 е равно = 10

    1.011 * 2^3 е равно = 11

    1.100 * 2^3 е равно = 12

    1.101 * 2^3 е равно = 13

    1.110 * 2^3 е равно = 14

    1.111 * 2^3 е равно = 15

Коментари - плаваща запетая

- Размерът на порядъка влияе в/у диапазона на представените данни.
- Предпоставка:

$1.1_2 = 1.5_{10}$

$1.11_2 = 1.75_{10}$

$1.111_2 = 1.875_{10}$

$1.1111_2 = 1.9375_{10}$

. . . . . . … …

$1.11...1_2 \approx 2.0_{10}$

Коментари - плаваща запетая

- Размерът на порядъка влияе в/у диапазона на представените данни.
- Пример:
  - Мантиса any size, порядък 3 бита, range -4 ; +3
    - Max полож. число: 1.111…111 * 2^3 ≈ 2 * 2^3 = 2^4 = 16
    - Min полож. число: 1.000…000 * 2^(-4) = 1/(2^4) = 1/16
  - Мантиса, any size, порядък 4 бита, range -8 ; +7
    - Max полож. число: 1.111…111 * 2^7 ≈ 2 * 2^7 = 2^8 = 256
    - Min полож. число: 1.000…000 * 2^(-8) = 1/(2^8) = 1/256

Коментари - плаваща запетая

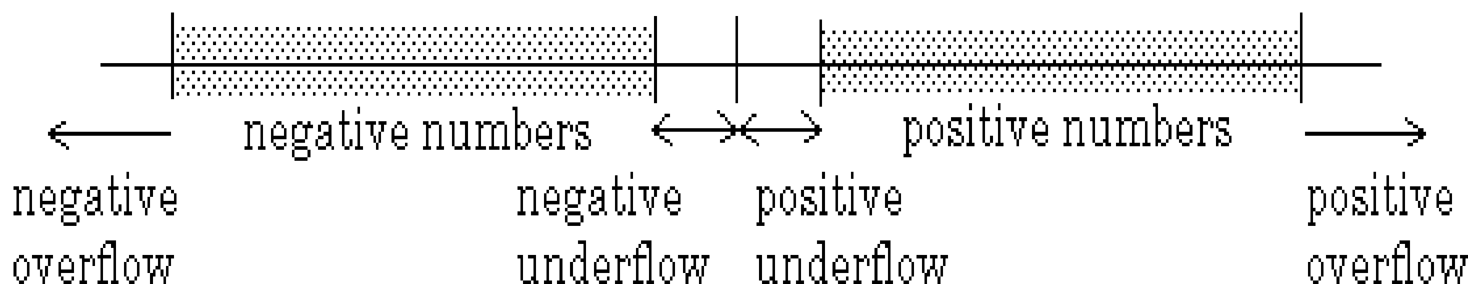
- Конверсия от 10-на в 2-на бройна система води до загуба на точност.
- 1.5_{10} =
- 1.75_{10} =
- 1.875_{10} =
- 1.6_{10} =

Коментари - плаваща запетая

- Конверсия от 10-на в 2-на бройна система води до загуба на точност.
- $1.5_{10} = 1.1_{2}$
- $1.75_{10} = 1.11_{2}$
- $1.875_{10} = 1.111_{2}$
- $1.6_{10} = 1.10011001..._{2}$

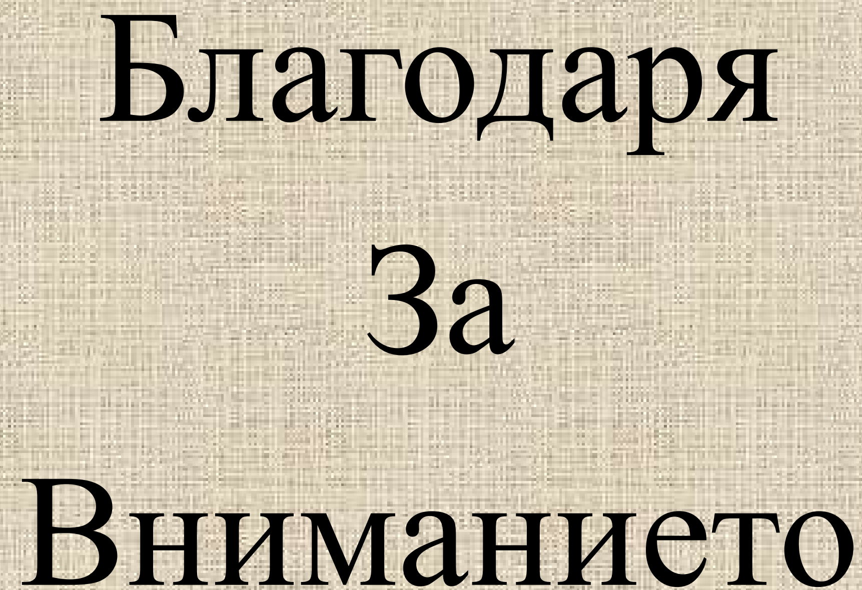
Коментари - плаваща запетая

- Разликата м/у реални числа (безкрайно множество) и FP числа (крайно множество)
  – Препълване;
  – Изчезване на порядък.



Numeric Data Type: decimal

- Десетичните типове се съхраняват като символни низове като се ползват двоични кодове за десетичните цифри. Тези кодове се наричат ***binary coded decimal*** (BCD).
- Две разновидности:
  - Зонов формат – 1 цифра/байт;
  - Пакетиран формат – 2 цифри/байт.
- Десетичните данни заемат повече памет от двоичните формати с фикс. и плав. запетая.


Благодаря
За
Вниманието