

### Лекція 3. Рекурсія та рекурсивні алгоритми

**Рекурсія** - це одна з фундаментальних концепцій у математиці й програмуванні, Рекурсія - це одна з форм мислення, цей потужний засіб, що дозволяє будувати елегантні й виразні алгоритми.

Об'єкт називається рекурсивним, якщо він містить сам себе або визначений за допомогою самого себе.

Якщо процедура  $p$  містить явне звертання до самої себе, то вона називається явно рекурсивною. Якщо процедура  $p$  містить звертання до деякої процедури  $q$ , що у свою чергу містить пряме або непряме звертання до  $p$ , то  $p$  - називається побічно рекурсивною.

Але рекурсивна програма не може викликати себе нескінченно, інакше вона ніколи не зупиниться, у такий спосіб у програмі (функції) повинна бути присутнім ще один важливий елемент - так звана термінальна умова, тобто умова за якої програма припиняє рекурсивний процес.

#### Рекурентності

Рекурентність - це рекурсивне визначення функції. Вони широко поширені в математику. Можливо найбільш знайома Вам з такого роду функцій - це факторіал. Факторіал - це добуток натуральних чисел від одиниці до якого-небудь даного натурального числа. Він визначається формулою:

$$N! = N((N-1)!), \quad \text{для } N \geq 1 \text{ і } 0! = 1.$$

Це прямо відповідає нижченаведеній рекурсивній програмі:

```
function factorial( N : integer ) : integer;  
begin  
  if N=0 then  
    factorial := 1  
  else  
    factorial := N * factorial( N-1);  
end;
```

Ця програма демонструє основні властивості рекурсивних програм: програма викликає сама себе (з меншим значенням аргументу), і в неї є термінальна умова при якому вона прямо обчислює результат.

Необхідно також пам'ятати про те, що це - програма, а не формула: наприклад ні формула, ні програма не працюють із негативними  $N$ , але згубні наслідки спроби зробити обчислення для негативного числа більше помітні для програми, чим для формули. Виклик `factorial(-1)` приведе до нескінченного рекурсивного циклу. Тому перед викликом даної програми потрібно робити перевірку умови незаперечності.

Друге, добре відоме рекурентне співвідношення - співвідношення визначального числа Фібоначі. Числа Фібоначі - це елементи числової послідовності 1, 1, 2, 3, 5, 8 ..., у яких кожний наступний елемент дорівнює сумі попередніх.

$$F_N = F_{N-1} + F_{N-2}, \text{ де } N \geq 2 \text{ і } F_0 = F_1 = 1.$$

І знову, рекурентність відповідає простій рекурсивній програмі:

```

function fibonacci( N : integer ) : integer;
begin
  if N<=1 then
    fibonacci := 1
  else
    fibonacci := fibonacci( N-1 ) + fibonacci( N-2 );
  end;
end;

```

Як ми побачимо, багато цікавих алгоритмів можна легко реалізувати за допомогою рекурсивних програм, і багато розроблювачів алгоритмів бажають виражати алгоритми рекурсивно. Але часто трапляється також і так, що настільки ж цікавий алгоритм ховається в деталях нерекурсивної реалізації. Давайте розглянемо такий алгоритм:

```

const
max = 25;
var
i : integer;
F : array [0..max] of integer;

procedure fibonacci;
begin
  F[0] := 1;
  F[1] := 1;
  for i := 2 to max do
    F[i] := F[i-1]+F[i-2];
  end;
end;

```

Ця програма обчислює перші max чисел Фібоначі, використовуючи масив розміру max. Цей метод називає ітераційним.

Який же метод краще? Точних правил для вибору між рекурсивною й нерекурсивною версіями алгоритму рішення завдання не існує. Стислість і виразність більшості рекурсивних процедур спрощує їхнє читання й супровід. З іншого боку, виконання рекурсивних процедур вимагає великих витрат і пам'яті, і часу процесора ніж їхні ітераційні аналоги.

### **Розподіл навпіл.**

Велика частина алгоритмів використовує два рекурсивних виклики, кожний з яких працює приблизно з половиною вхідних даних. У дизайні алгоритмів таке явище називають "розподіл навпіл"; його часто використовують для досягнення істотної економії.

Як приклад, давайте розглянемо завдання нанесення розподілів на лінійку: на ній повинна бути мітка в точці  $1/2$ ", мітка трохи коротша через кожні  $1/4$ ", ще більш коротка через  $1/8$ " і так далі.



Рисунок 1 – Лінійка

Ми також припускаємо, що в наше розпорядження надана процедура mark(x, h) для нанесення мітки висотою h одиниць на лінійку в позицію x. Центральна мітка повинна мати висоту n одиниць, мітки в центрах лівої й правої половинок - n-1 одиниць, і так далі. Наступна рекурсивна програма - прямий шлях досягнення нашої мети:

```

procedure rule( l, r, h : integer );
var m : integer;
begin

```

```

if h>0 then
begin
  m := (l+r) div 2;
  mark( m, h );
  rule( l, m, h-1);
  rule( m, r, h-1);
end;
end;

```

Ідея цього методу полягає в наступному: для того, щоб промаркувати лінійку, ми спершу наносимо довгу мітку по її середині. Це ділить її на дві рівні половини. Тепер ми наносимо (більш короткі) мітки в середині кожної із цих половинок використовуючи ту ж саму процедуру.

Необхідно звертати особливу увагу на термінальну умову рекурсивної програми - у протилежному випадку вона ніколи не зупиниться! У вищенаведеній програмі ми термінуємо, коли ми повинні нанести мітку висоти 0. Розглянемо приклад, у результаті виклику `rule(0, 8, 3)`. Ми ставимо мітку по середині й викликаємо `rule` для лівої половини, потім робимо теж саме для лівої половини, і так далі поки висота мітки не стане дорівнює 0. В остаточному підсумку ми вертаємося з `rule` і розмічаємо праву половину подібним

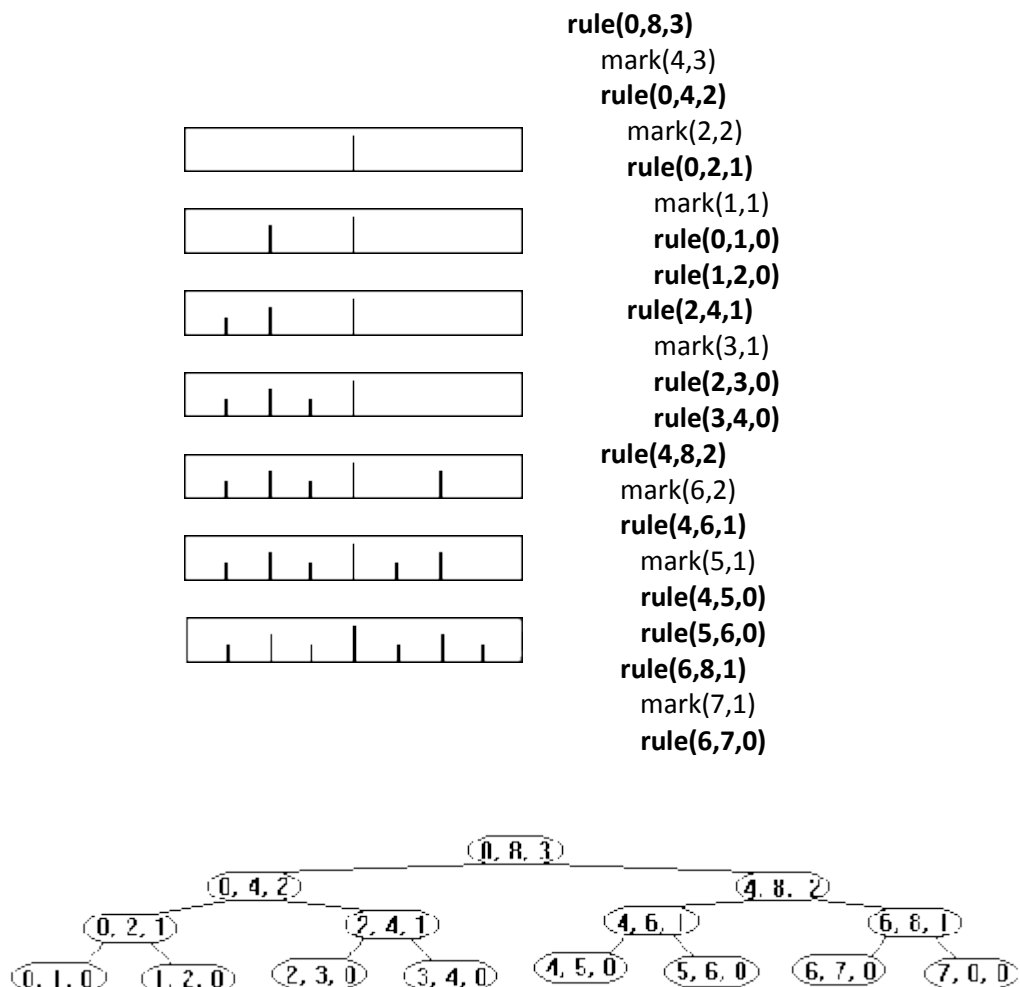


Рисунок 2 – Дерево рекурсивних викликів для малювання лінійки

Інший нерекурсивний алгоритм полягає в тому, щоб малювати спершу найкоротші мітки, потім трохи менш короткі й так далі, як показано в наступній, досить компактній, нерекурсивній програмі:

```

procedure draw(l, r, h : integer);
var
i, j : integer;
begin
  j:=1;
  for i:=1 to h do
  begin
    for x:=0 to (l+r) div j do
mark(l+j+x*(j+j),i);
    j:=j+j;
  end;
end;

```

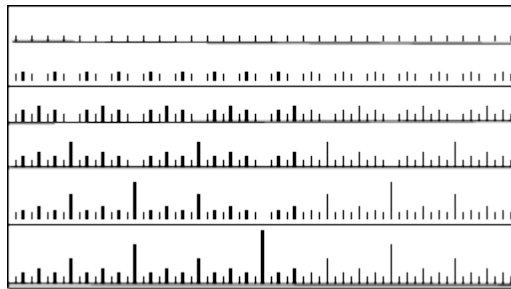


Рисунок 3 – Нерекурсивне малювання лінійки

Зараз створимо двомірний візерунок, що демонструє як проста рекурсія може дати рішення тому, що здається складним.

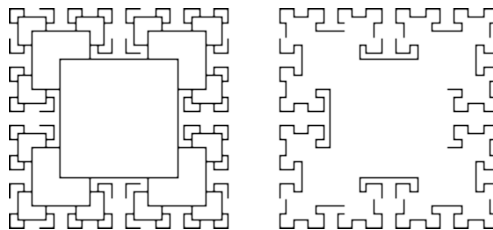


Рисунок 3 – Фрактальна зірка (ліворуч) і її обриси (праворуч)

```

uses
graph,crt;
var
r,gd,gm: Integer;

procedure star(x,y,r:integer);
begin
  if r>2 then begin
    star(x+r,y+r,r div 2);
    star(x+r, y-r,r div 2);
    star( x-r,y-r,r div 2);
    star( x-r,y+r,r div 2);
    bar( x-r,y-r,x+r,y+r);
  end;
end;

begin
clrscr;

```

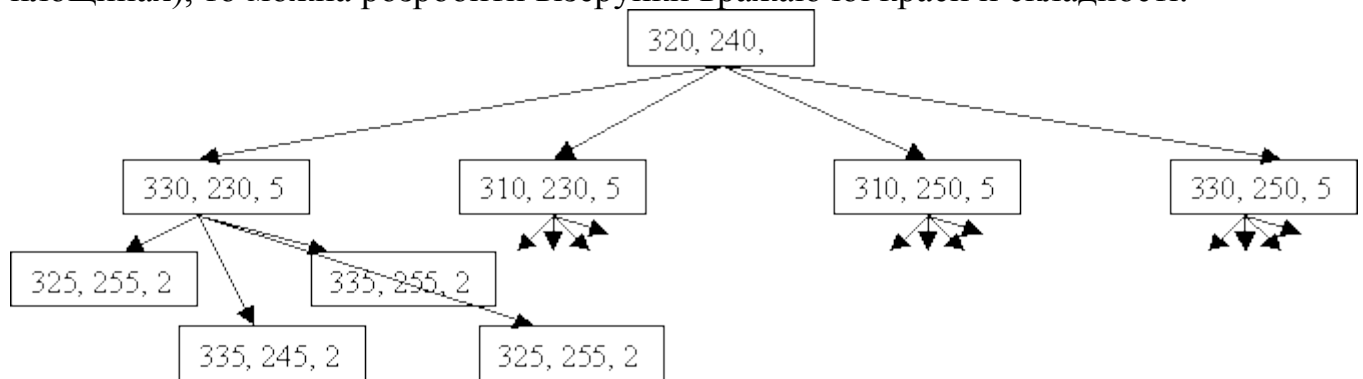
```

Gd := Detect;
write('Enter the lenght: ');
readln(r);
InitGraph(Gd, Gm, 'з:\bp\bgi');
if GraphResult <> grOk then
Halt(1);
setbkcolor(0);
SetFillStyle(1, 9);
star(getmaxx div 2, getmaxy div 2, r);
repeat
until keypressed;
end.

```

Малюючий примітив тут - просто процедура яка малює квадрат розміру  $2r$  із центром в  $(x, y)$ .

Рекурсивно визначені геометричні візерунки подібні цьому називають фракталами. Якщо використовується більш складний примітив для малювання й більш складні рекурсивні виклики (особливо на речовинній і комплексній площинах), то можна розробити візерунки вражаючої краси й складності.



## Рекурсія

**Рекурсія** — процес повторення чого-небудь самоподібним способом. Наприклад, вкладені віддзеркалення, вироблені двома точно паралельними один одному дзеркалами, є однією з форм нескінченної рекурсії. Даний термін має більш спеціальні значення в різних областях знань — від лінгвістики до логіки.

Найбільш загальне застосування рекурсія знаходить у математиці й інформатиці. Тут вона є методом визначення функцій, при якому обумовлена функція застосована в тілі свого ж власного визначення. При цьому нескінченний набір випадків (значень функції) описується за допомогою кінцевого виразу, що для деяких випадків може посилатися на інші випадки, якщо при цьому не виникає циклів або нескінченного ланцюга посилань. Фактично це спосіб визначення множини об'єктів через самого себе з використанням раніше заданих окремих правил.

Визначення, що використовує рекурсію, називається індуктивним. Одним із прикладів подібного визначення є аксіоматична побудова множини натуральних чисел.

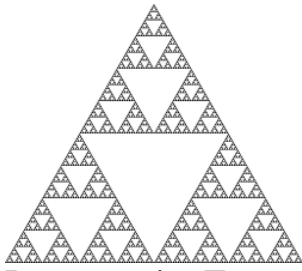


Рисунок 4 – Трикутник Серпінського

– Факторіал цілого ненегативного числа  $n$  (позначається  $n!$ ) визначається як  $n! = n \cdot (n - 1)!$  при  $n > 0$  і  $n! = 1$  при  $n = 0$

– Числа Фібоначі визначаються за допомогою рекурентного співвідношення:

Перше й друге числа Фібоначі рівні 1

Для  $n > 2$ ,  $n$ -е число Фібоначі дорівнює сумі  $(n - 1)$ -го й  $(n - 2)$ -го чисел Фібоначі

– Практично всі геометричні фрактали задаються у формі нескінченної рекурсії (наприклад, трикутник Серпінського).

– Рекурсивні акроніми: GNU (GNU Not Unix), PHP (PHP: Hypertext Preprocessor) і т.д.

### **Рекурсія в програмуванні**

#### **Функції**

У програмуванні рекурсія — виклик функції (процедури) з її ж самої, безпосередньо (проста рекурсія) або через інші функції (складна рекурсія), наприклад, функція  $A$  викликає функцію  $B$ , а функція  $B$  — функцію  $A$ . Кількість вкладених викликів функції або процедури називається глибиною рекурсії.

Перевага рекурсивного визначення об'єкта полягає в тому, що таке кінцеве визначення теоретично здатне описувати нескінченно велику кількість об'єктів. За допомогою рекурсивної програми ж можливо описати нескінченне обчислення, причому без явних повторень частин програми.

**Реалізація рекурсивних викликів функцій у практично використовуваних мовах і середовищах програмування, як правило, опирається на механізм стека викликів — адреси повернення й локальних змінних функцій записуються в стек, завдяки чому кожний наступний рекурсивний виклик цієї функції користується своїм набором локальних змінних і за рахунок цього працює коректно.** Зворотним боком цього досить простого за структурою механізму є те, що на кожний рекурсивний виклик потрібна деяка кількість оперативної пам'яті комп'ютера, і при надмірно великій глибині рекурсії може наступити переповнення стека викликів. Внаслідок цього, звичайно рекомендується уникати рекурсивних програм, які приводять (або в деяких умовах можуть приводити) до занадто великої глибини рекурсії.

Втім, є спеціальний тип рекурсії, називаний «хвостовою рекурсією». Інтерпретатори й компілятори функціональних мов програмування, що підтримують оптимізацію коду (вихідного й/або що виконується), автоматично перетворюють хвостову рекурсію до ітерації, завдяки чому забезпечується

виконання алгоритмів із хвостовою рекурсією в обмеженому обсязі пам'яті. Такі рекурсивні обчислення, навіть якщо вони формально нескінченні (наприклад, коли за допомогою рекурсії організується робота командного інтерпретатора, що приймає команди користувача), ніколи не приводять до вичерпання пам'яті. Однак, далеко не завжди стандарти мов програмування чітко визначають, яким саме умовам повинна задовольняти рекурсивна функція, щоб транслятор гарантовано перетворив її в ітерацію. Один з рідкісних винятків – мова Scheme (діалект мови Lisp), опис якої містить всі необхідні відомості.

*Будь-яку рекурсивну функцію можна замінити циклом і стеком.*

### **Дані**

Опис типу даних може містити посилання на самого себе. Подібні структури використовуються при описі списків і графів. Приклад опису списку (C++):

```
class element_of_list
{
    element_of_list *next; /* посилання на наступний елемент того ж типу */
    int data; /* якісь дані */
};
```

Рекурсивна структура даних найчастіше спричиняє застосування рекурсії для обробки цих даних.

### **Корекурсія**

**Корекурсія** – у теорії категорій та інформатиці тип операції, дуальний до рекурсії. Звичайно корекурсія використовується (разом з механізмом ледачих обчислень) для генерації нескінченних структур даних.

### **Загальні зауваження**

Правило використання корекурсії на коданних дуальне правилу застосування рекурсії на даних. Замість *згортання* структури даних виходячи з початкового значення аргументу, корекурсія *розгортає* результат на основі початкового значення аргументу. Необхідно відзначити, що корекурсія *створює* потенційно нескінченні структури даних, у той час як звичайна рекурсія *аналізує* (розбирає) по необхідності кінцеві структури даних. Звичайна рекурсія незастосовна до коданних, оскільки процес аналізу може ніколи не зупинитися. Відповідно, корекурсія не може робити дані, оскільки дані завжди кінцеві.

### **Приклади**

Приклад використання механізму корекурсії мовою Haskell (обчислення нескінченного списку чисел Фібоначі):

```
fibs = 0:1 : zipWith (+) fibs (tail fibs)
Інший приклад - обчислення нескінченного списку простих чисел:
primes = eratosthenes [2..]
    where
        eratosthenes (x:xs) = x:eratosthenes (filter ((/= 0).('mod' x)) xs)
```

Дана функція реалізує алгоритм «решето Ератосфена», причому робить це самим безпосереднім чином.

Наведені приклади мовою Haskell не зовсім коректні, оскільки в мові немає ідіоми коданних. У зазначених прикладах коданні тільки емулюються за допомогою нескінченного списку.