Розділ 2. Рекурсія та рекурсивні алгоритми

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.

Рекурсія – це одна з фундаментальних концепцій у математиці й програмуванні. Рекурсія – це одна з форм мислення, цей поужний засіб, ще дозволяє будувати елегантні й виразні алгоритми.

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 ланцюжка посилань. Фактично це спосіб визначення множини об'єктів через самого себе з використанням раніше заданих окремих правил.

Визначення, які використовують рекурсію, називаються *індуктивними*. Одним із прикладів подібного визначення є аксіоматична побудова множини натуральних чисел.

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

– Числа Фібоначі визначаються за допомогою рекурентного співвідношення:

Перше й друге числа Фібоначі рівні 1.

Для $n > 2$, $n - e$ 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.

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, яка 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, для цього у програмі (функції) повинен бути присутнім ще один важливий елемент – так звана термінальна умова, тобто умова за якої програма припиняє рекурсивний процес.

Рекурсія в програмуванні. Функції

У програмуванні рекурсія – виклик функції (процедури) з неї 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 скінченне визначення теоретично здатне описувати нескінченно велику кількість об'єктів. За допомогою рекурсивної програми ж можливо описати нескінченне обчислення, причому без явних повторень частин програми.

На практиці реалізація рекурсивних викликів функцій у мовах і середовищах програмування, як правило, опирається на механізм стеку 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, зазвичай рекомендується уникати рекурсивних програм, які приводять (або при деяких умовах можуть приводити) до занадто великої глибини рекурсії.

Втім, є спеціальний тип рекурсії, що називається *«хвостовою рекурсією»*. Інтерпретатори й компілятори функціональних мов програмування, що підтримують оптимізацію коду (вихідного та/або що виконується), автоматично перетворять 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 організовується 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.

Рекурентності

Рекурентність – це рекурсивне визначення функції. Вона широко поширена в математиці. Можливо, найбільш знайома Вам з такого роду функцій, як факторіал. Факторіал – це

добуток натуральних чисел від одиниці до деякого даного натурального числа. Він визначається формулою:

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

Це прямо відповідає нижченаведеному псевдокоду рекурсивної програми:

```
function int factorial(int N)
 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, за якою вона прямо обчислює результат.

Необхідно також пам'ятати про те, що це – програма, а не формула: наприклад ні формула, ні програма не працюють із від'ємними *N*, але згубні наслідки спроби зробити обчислення для від'ємного числа більш помітні для програми, ніж для формули. Виклик factorial(-1) призведе до нескінченного рекурсивного циклу. Тому перед викликом даної програми потрібно робити перевірку умови незаперечності.

Друге добре відоме рекурентне співвідношення – співвідношення для визначення чисел Фібоначі. Числа Фібоначі – це елементи числової послідовності 1, 1, 2, 3, 5, 8, ..., у яких кожний наступний елемент дорівнює сумі попередніх.

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

І знову, рекурентність відповідає простій рекурсивній програмі, яку можна реалізувати на основі наступного псевдокоду:

```
function int fibonacci(int N)
 begin
    if N <= 1 then
        fibonacci = 1
    else
   fibonacci = fibonacci(N - 1) + fibonacci(N - 2)
 end
```

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

```
const max = 25
var int i; array of int F[0..max]

procedure fibonacci()
 begin
    F[0] = 1
    F[1] = 1
    for i = 2 to max do
        F[i] = F[i - 1] + F[i - 2]
 end
```

Ця програма обчислює перші max чисел Фібоначі, використовуючи масив розміром max. Цей метод називається *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 в порівнянні з їхніми ітераційними аналогами.

Поділ навпіл

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

Як приклад, давайте розглянемо завдання нанесення поділок 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 (рис. 2.1).

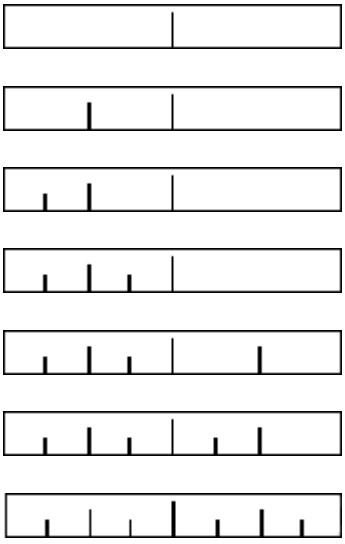
Рисунок 2.1 – Лінійка

Ми також припускаємо, що в нас є 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 – висоту (*n* – 1) одиниць, і так далі. Наступний псевдокод рекурсивної програми ілюструє прямий шлях досягнення нашої мети:

```
procedure rule(int l, int r, int h)
 var int m
 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 в її середині. Це ділить її на дві рівні частини. 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 перевіряємо, коли ми повинні нанести мітку висотою 0. Розглянемо приклад, як результат 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 дорівнювати 0. В остаточному підсумку ми вертаємося до rule і розмічаємо праву половину подібним чином.



```
rule(0,8,3)
     mark(4,3)
     rule(0,4,2)
         mark(2,2)
         rule(0,2,1)
             mark(1,1)
             rule(0,1,0)
             rule(1,2,0)
         rule(2,4,1)
             mark(3,1)
             rule(2,3,0)
             rule(3,4,0)
     rule(4,8,2)
        mark(6,2)
        rule(4,6,1)
           mark(5,1)
           rule(4,5,0)
           rule(5,6,0)
        rule(6,8,1)
           mark(7,1)
           rule(6,7,0)
```

Рисунок 2.2 – Ілюстрація послідовності рекурсивних викликів

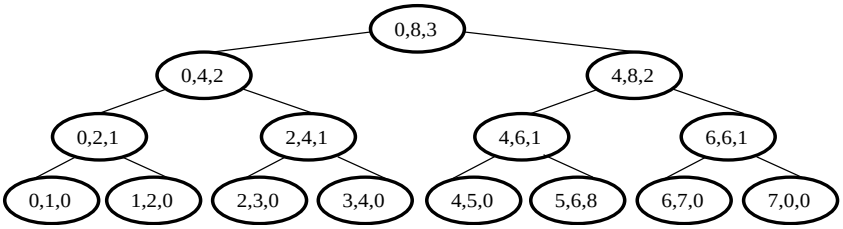


Рисунок 2.3 – Дерево рекурсивних викликів для малювання лінійки

Інший нерекурсивний алгоритм полягає в тому, щоб малювати спершу найкоротші мітки, потім трохи менш короткі й так далі, як показано в наступному, досить компактному, нерекурсивному псевдокоді:

```
procedure draw(int l, int r, int h)
 var  int i, int j
 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
```

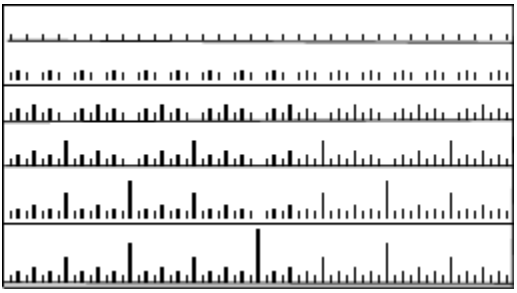


Рисунок 2.4 – Нерекурсивне малювання лінійки

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

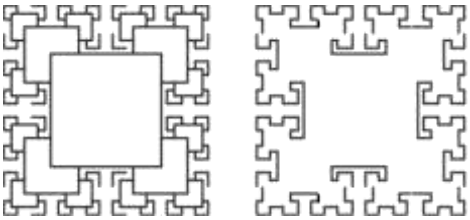


Рисунок 2.5 – Фрактальна зірка (ліворуч) і її обриси (праворуч)

Приклад псевдокоду алгоритму створення фрактальної зірки:

```
var int r

 procedure star(int x, int y, int r)
  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)
         drawFilledRrectangle(x - r, y - r, x + r, y + r)
     end
  end

 begin

  ...  // тут повинне бути очищення екрану

     write('Enter the length of star: ')
     read(r)

...  // тут повинні бути функції налаштування графіки
```

```
  do
     star(getMaxX() div 2, getMaxY() div 2, r)
     repeat
  until (not keyPressedEsc)
end
```

Примітив для малювання тут – просто процедура, яка малює квадрат розміром 2*r* 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 дійсній і комплексній площинах), то можна розробити візерунки вражаючої краси й складності. Відомим прикладом фракталу є трикутник Серпінського (рис. 2.6).

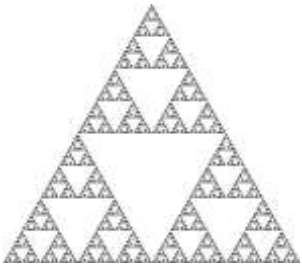


Рисунок 2.6 – Трикутник Серпінського

Рекурсивні структури даних

Опис типу даних може містити посилання на самого себе. Подібні структури використовуються при описі списків і графів. Спосіб створення рекурсивних типів даних дуже залежить від засобів тієї чи іншої мови програмування і, зокрема, реалізації у ній вказівників. Наведемо приклад створення рекурсивної структури даних не на псевдокоді, а на коді реальної мови програмування.

Приклад 2.1. Опис списку на мові C++

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

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

Корекурсія

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

Загальні зауваження

Правило використання корекурсії на коданих (прикладом яких є потоки) 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 скінченні структури даних. Звичайна рекурсія незастосовна до коданих, оскільки процес аналізу може ніколи не зупинитися. Відповідно, корекурсія не може продукувати дані, оскільки дані завжди скінченні.

Приклади корекурсії

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 коданих. У зазначених прикладах кодані тільки емулюються за допомогою нескінченного списку.

Питання до розділу:

1. В чому різниця між явно і побічно рекурсивними процедурами?
2. Який важливий елемент повинен бути присутнім у програмі (функції) для її коректної роботи?
3. Які широко поширені рекурентності Ви знаєте з математики?
4. Чому перед викликом програми для обчислення факторіалу потрібно робити перевірку умови незаперечності?
5. Які переваги й недоліки використання рекурсивних процедур?
6. На Ваш погляд, який із наведених вище прикладів для малювання лінійки (а саме рекурсивний розподіл навпіл чи нерекурсивний алгоритм) кращий і чому?
7. Дайте означення фракталу.
8. Дайте означення глибини рекурсії.
9. Чому рекомендують уникати рекурсивних програм?
10. Опишіть порядок роботи “хвостової рекурсії”.
11. Дайте визначення корекурсії.