Temat: Rekurencja i jej zastosowanie

Słowo **rekurencja** (również rekursja) wywodzi się z języka łacińskiego - *recurrere = biec z powrotem*. W algorytmice mówimy, że dany algorytm jest rekurencyjny, jeśli do rozwiązania pewnego problemu wykorzystuje on sam siebie. W programowaniu dana funkcja jest rekurencyjna, jeśli wywołuje samą siebie. Kolejne wywołania takiej funkcji nazywamy rekurencyjnym ciągiem wywołań. Ciąg ten nie może być nieskończony - każde wywołanie funkcji powoduje umieszczenie w pamięci komputera adresu powrotu, czyli miejsca w programie, do którego wraca procesor, gdy zakończy wykonywać kod funkcji. Ponieważ pamięć jest skończona, to nie można w niej umieścić nieskończenie wiele adresów powrotnych. Dlatego w rekurencji bardzo ważne jest określenie warunku, który kończy rekurencję - np. funkcja przestaje już dalej wywoływać samą siebie.

Rekurencyjna silnia

Silnia n! (ang. factorial of n) jest iloczynem kolejnych liczb naturalnych mniejszych lub równych n. Możemy ją zapisać w sposób rekurencyjny:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n \geq 1 \end{cases}$$

Wykorzystując ten wzór policzmy np. 5!:

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0!$$

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1$$

$$5! = 120$$

Zwróć uwagę, iż w obliczeniach cofamy się od n = 5 do n = 0. Gdy osiągniemy 0, to rekurencja ustaje.

Przykład:

```
#include <iostream>

using namespace std;

unsigned silnia(unsigned n)
{
    if(n == 0) return 1;
    else       return n * silnia(n - 1);
}

int main()
{
    unsigned n;

    cin >> n;

    cout << n << "! = " << silnia(n) << endl << endl;
```

```
    return 0;
}
```

Silnia rośnie bardzo szybko. Już 13! przekroczy dopuszczalny zakres dla liczb typu unsigned (0...4mld). Zwróć uwagę, iż funkcja rekurencyjna jest bardzo prosta - to jest właśnie zaleta rekurencji.

Rekurencyjny Największy Wspólny Dzielnik

Algorytm Euklidesa możemy również zapisać w sposób rekurencyjny:

$$NWD(a,b) = \begin{cases} a, & b = 0 \\ NWD(b, a \bmod b), & b \neq 0 \end{cases}$$

Może się to wydawać na pierwszy rzut oka dziwne, lecz tak właśnie działa algorytm Euklidesa z dzieleniem, który omawialiśmy wcześniej. Traktuje on liczbę b jako resztę z dzielenia a przez b. Jeśli reszta z dzielenia jest równa 0, to kończy zwracając a. Jeśli nie, to a zastępuje przez b, czyli obecną resztą z dzielenia, a b zastępuje nową resztą z dzielenia i znów wykonuje sam siebie aż do pożądanego skutku.

Przykład:

```
#include <iostream>

using namespace std;

unsigned NWD(unsigned a, unsigned b)
{
    if(b == 0) return a;
    else       return NWD(b, a % b) ;
}

int main()
{
    unsigned a, b;

    cin >> a >> b;

    cout << "NWD(" << a << "," << b << ") = " << NWD(a, b)
<< endl << endl;

    return 0;
}
```

Liczby Fibonacciego

Liczby Fibonacciego powstają również rekurencyjnie:

$$fib_n = \begin{cases} n, & n \leq 1 \\ fib_{n-2} + fib_{n-1}, & n > 1 \end{cases}$$

Oto kilka początkowych liczb Fibonacciego:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_2 = fib_0 + fib_1 = 0 + 1 = 1$$
$$fib_3 = fib_1 + fib_2 = 1 + 1 = 2$$
$$fib_4 = fib_2 + fib_3 = 1 + 2 = 3$$
$$fib_5 = fib_3 + fib_4 = 2 + 3 = 5$$
$$fib_6 = fib_4 + fib_5 = 3 + 5 = 8$$
$$\ldots$$

Za wyjątkiem dwóch pierwszych, każda kolejna liczba Fibonacciego powstaje jako suma dwóch poprzednich liczb. Jeśli dla tych liczb zastosujemy metodę rekurencyjną, to, owszem, funkcja tworząca będzie prosta, lecz liczba wywołań rekurencyjnych może prześcignąć nasze wyobrażenia. Spróbujmy rozwinąć rekurencyjnie fib6 (na czerwono zaznaczono liczby Fibonacciego, które się dalej rozkładają rekurencyjnie):

$fib_6 = \mathbf{fib_4} + \mathbf{fib_5}$
$fib_6 = \mathbf{fib_2} + \mathbf{fib_3} + \mathbf{fib_3} + \mathbf{fib_4}$
$fib_6 = fib_0 + fib_1 + fib_1 + \mathbf{fib_2} + fib_1 + \mathbf{fib_2} + \mathbf{fib_2} + \mathbf{fib_3}$
$fib_6 = fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 +$
$fib_0 + fib_1 + fib_1 + \mathbf{fib_2}$
$fib_6 = fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 +$
$fib_0 + fib_1 + fib_1 + fib_0 + fib_1$

A teraz rozwinięcie rekurencyjne dla fib7:

$fib_7 = \mathbf{fib_5} + \mathbf{fib_6}$
$fib_7 = \mathbf{fib_3} + \mathbf{fib_4} + \mathbf{fib_4} + \mathbf{fib_5}$
$fib_7 = fib_1 + \mathbf{fib_2} + \mathbf{fib_2} + \mathbf{fib_3} + \mathbf{fib_2} + \mathbf{fib_3} + \mathbf{fib_3} + \mathbf{fib_4}$
$fib_7 = fib_1 + fib_0 + fib_1 + fib_0 + fib_1 + fib_1 + \mathbf{fib_2} + fib_0 +$
$fib_1 + fib_1 + \mathbf{fib_2} + fib_1 + \mathbf{fib_2} + \mathbf{fib_2} + \mathbf{fib_3}$
$fib_7 = fib_1 + fib_0 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 +$
$fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_0 +$
$fib_1 + fib_1 + \mathbf{fib_2}$
$fib_7 = fib_1 + fib_0 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 +$
$fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_1 + fib_0 + fib_1 + fib_0 +$
$fib_1 + fib_1 + fib_0 + fib_1$

Jak widzimy, liczba wywołań lawinowo rośnie wraz ze wzrostem n. Taki przyrost nazywamy przyrostem wykładniczym. Niestety, ta cecha powoduje, iż obliczenie metodą rekurencyjną liczb Fibonacciego może być operacją bardzo czasochłonną dla dużych n.

Przykład:

```
#include <iostream>

using namespace std;

unsigned long long fib(int n)
{
    if(n <= 1) return n;
    else       return fib(n - 2) + fib(n - 1);
}

int main()
{
    int n;

    cin >> n;

    cout << "fib(" << n << ") = " << fib(n) << endl <<
endl;

    return 0;
}
```

Zadania

Zad.1 Poniżej zdefiniowany jest pewien ciąg, którego kolejne wyrazy generowane są w sposób rekurencyjny:

$$a_n = \begin{cases} 1, & dla\ n = 1 \\ 0,5, & dla\ n = 2 \\ -a_{n-1} \cdot a_{n-2}, & dla\ n > 2 \end{cases}$$

Napisz program, który znajdzie wartość *n-tego* wyrazu ciągu.

Zad. 2. Napisz program, który wyznaczy sumę cyfr liczby naturalnej z zakresu [0...1020]. Rozwiąż zadanie metodą rekurencyjną.

Rozwiązanie

Aby wyłuskać cyfrę jedności danej liczby należy wykonać operację:

cyfra=n mod 10

gdzie mod oznacza resztę z dzielenia. Następnym krokiem jest skrócenie liczby o jedną cyfrę wykonując operację

n=n div 10

gdzie div oznacza dzielenie całkowite. Powtarzamy te operacje do momentu otrzymania liczby 0.

$$sumacyfr(123) = \overbrace{123 \; mod \; 10}^{3} + \underbrace{sumacyfr(\overbrace{123 \; div \; 10}^{12})}$$

$$\overbrace{12 \; mod \; 10}^{2} + \underbrace{sumacyfr(\overbrace{12 \; div \; 10}^{1})}$$

$$\overbrace{1 \; mod \; 10}^{1} + \underbrace{sumacyfr(\overbrace{1 \; div \; 10}^{0})}$$

$$\underbrace{0}$$

$$= 3 + 2 + 1 + 0 = 6$$

Zad. 3. Napisz program, który zapisze podaną liczbę dziesiętną naturalną w systemie binarnym. Rozwiąż zadany problem rekurencyjnie.