Algorytmy rekurencyjne

Rekurencja

Rekurencja (inaczej **rekursja**) – odwołanie się funkcji lub definicji do samej siebie. Mówiąc inaczej, podejście rekurencyjne polega na tym, że rozwiązanie problemu wyraża się za pomocą rozwiązania tego samego problemu dla mniejszych danych wejściowych. Stosowanie rekurencji jest charakterystyczne dla algorytmów projektowanych metodą *dziel i zwyciężaj*.

Przykład

Typowym problemem, dla którego można zastosować rekurencję, jest obliczanie silni. Przypomnijmy, że silnia z *n* jest zdefiniowana jako *n!=1×2×…×n*. Funkcja ta może być równoważnie zapisana jako:

$$n! = (n-1)! * n \qquad dla\ n > 0$$
$$n! = 1 \qquad dla\ n = 0$$

W powyższym przykładzie górny wiersz jest ogólnym równaniem rekurencji, zaś dolny wiersz jest wartością brzegową. W języku C++ powyższa funkcja byłaby zapisana w poniższy sposób.

```
int silnia(int n)
{
    if (n > 0)
    {
        return n * silnia(n-1);
    }
    else
    {
        return 1;
    }
};
```

Przekształcenie postaci rekurencyjnej funkcji do **postaci zwartej** (tzn. takiej, która nie zawiera odwołania do samej siebie) jest określane jako **rozwiązanie rekurencji**. Metody rozwiązywania rekurencji są dostępne między innymi w książkach podanych w bibliografii.

Algorytmy stosujące rekurencję są zazwyczaj proste w implementacji. Jednocześnie wiążą się one z pewnymi problemami. Przy podejściu rekurencyjnym ta sama funkcja jest wywoływana wielokrotnie, co zużywa pamięć operacyjną (w skrajnych przypadkach może to spowodować przepełnienie stosu).

Bibliografia

- R.L. Graham, D.E. Knuth, O. Patashnik, *Matematyka konkretna*, Wydawnictwo Naukowe PWN , Warszawa, 2012, ISBN 9788301147648 [książka w księgarni Helion.pl].
- Z.J. Czech, S. Deorowicz, P. Fabian, *Algorytmy i struktury danych. Wybrane zagadnienia*, Wydawnictwo Politechniki Śląskiej, Gliwice, 2010, ISBN 9788373356689.

Algorytmy

program wyznaczający sumę n kolejnych liczb naturalnych.

Załóżmy, że na wejściu podaliśmy liczbę **5** (program ma wyznaczyć sumę **1+ 2+ 3 + 4 + 5**).

$$\boldsymbol{wynik} = suma(5);$$

Funkcja suma(**n**), wywołała się z argumentem równym **5**. Najpierw sprawdzamy, czy **n< 1 (5 < 1)**. Warunek jest fałszywy, przechodzimy więc do następnej linijki **return 5** + suma(**5 - 1**) . Funkcja suma wywołana została przez samą siebie z argumentem równym 4, a więc mamy:

$$\boldsymbol{wynik} = suma(5) = 5 + suma(4),$$

daną czynność powtarzamy do momentu, gdy argument osiągnie wartość **0**, wtedy funkcja zwróci **0** (**0 < 1**, **prawda**).

$$\begin{aligned}
&\boldsymbol{wynik} = \\
&= suma(5) = \\
&= 5 + suma(4) = \\
&= 5 + 4 + suma(3) = \\
&= 5 + 4 + 3 + suma(2) = \\
&= 5 + 4 + 3 + 2 + suma(1) = \\
&= 5 + 4 + 3 + 2 + 1 + suma(0) = \\
&= 5 + 4 + 3 + 2 + 1 + 0 = \\
&= 15;
\end{aligned}$$

Rozwiązanie w C++

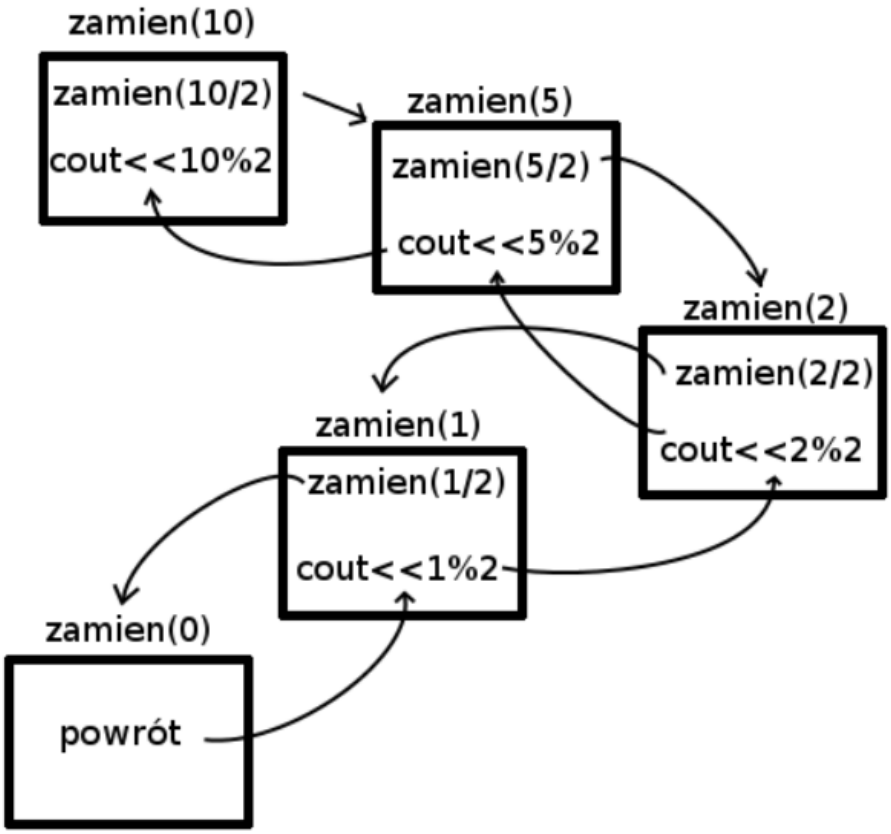
```
#include <iostream>
using namespace std;

long long suma(int n)
{
        if(n<1)
                return 0;

        return n+suma(n-1);
}

int main()
{
        int n;

        cout<<"Podaj liczbę: ";
        cin>>n;

        cout<<"Suma "<<n<<" kolejnych liczb naturalnych wynosi "
<<suma(n)<<endl;

        return 0;
}
```

zamiana liczb w systemie dziesiętnym na system dwójkowy

algorytm wyjaśniony szerzej: https://www.algorytm.edu.pl/algorytmy-maturalne/pozycyjne-reprezentacje-liczb.html

Do rozwiązania problemu użyjemy rekurencji z nawrotami. Funkcja będzie wywoływać samą siebie, i w momencie gdy już "głębiej" się nie zagnieździ będzie powracać aby wykonać pozostałe instrukcje. Dzięki temu wypisane bity będą w prawidłowej kolejności.

Graficzne przedstawienie programu



Rozwiązanie w C++

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

void zamien(int n)
{
        //jesli n == 0 to zawracamy
        if(n==0)return;

        zamien(n/2); //zagnieżdżamy rekurencję

        cout<<n%2; //przy powrocie
}

int main()
{
        int n;
        cout<<"Podaj liczbę naturalną: ";
        cin>>n;
        cout<<"Postać binarna liczby "<<n<<": ";
        if(n==0)
                cout<<0;
        else
                zamien(n);

        cout<<endl;

        return 0;
}
```

Ciąg Fibonacciego

Ciąg Fibonacciego definiujemy następująco:

pierwszy i drugi element ciągu jest równy 1. Każdy następny otrzymujemy dodając do siebie dwa poprzednie. Matematycznie wygląda to następująco:

$$\mathrm{Fn} = \begin{cases} 1 & ,dla\ n = 1 \\ 1 & ,dla\ n = 2 \\ \mathrm{Fn}-2\ +\mathrm{Fn}-1 & ,\ \mathrm{dla\ n} > 2 \end{cases}$$

Inna definicja przedstawia zerowy numer ciągu jako wartość 0, pierwszy jako wartość 1, a każdy następny otrzymujemy dodając dwa poprzednie:

$$\mathrm{Fn} = \begin{cases} 1 & ,dla\ n = 0 \\ 1 & ,dla\ n = 1 \\ \mathrm{Fn}-2\ +\mathrm{Fn}-1 & ,\ \mathrm{dla\ n} > 1 \end{cases}$$

Kilka kolejnych wyrazów tego ciągu według pierwszej definicji przedstawia się następująco:

1, 1, 2, 3, 5, 8, 13, 21, ...

Pierwsze rozwiązanie zostanie przedstawione metodą iteracyjną, która jest wydajna i bez problemu wyznaczymy wszystkie wyrazy ciągu, które mieszczą się w dowolnym typie w C++.

Strategia jest następująca:

Zmienna $\boldsymbol{a}$ będzie przechowywać wyraz o numerze $\boldsymbol{n-2}$,
zmienna $\boldsymbol{b}$ wyraz o numerze $\boldsymbol{n-1}$.

W każdym przejściu pętli, zmienna b przeskoczy na element następny, czyli sumę elementów a i b

$$\boldsymbol{b = a + b},$$

natomiast zmienna a przechowa to co przechowywała zmienna b czyli

$$\boldsymbol{a = b - a} = (a + b) - a = \boldsymbol{b}.$$

Przykład w C++

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

void fibonacci(int n)
{
     long long a = 0, b = 1;

     for(int i=0;i<n;i++)
     {
            cout<<b<<" ";
            b += a; //pod zmienną b przypisujemy wyraz następny czyli a+b
            a = b-a; //pod zmienną a przypisujemy wartość zmiennej b
     }
}

int main()
{
    int n;

    cout<<"Podaj ile chcesz wypisać wyrazów ciągu fibonacciego: ";
    cin>>n;

    fibonacci(n);

    return 0;
}
```

Postać rekurencyjną przedstawimy do wyświetlenia pojedynczego wyrazu, ponieważ algorytm jest bardzo niewydajny i nadaje się tylko do wyznaczania niewielkiej ilości elementów ciągu.

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

int fib(int n)
{
        if(n<3)
                return 1;

        return fib(n-2)+fib(n-1);
}

int main()
{

        int n;

        cout<<"Podaj nr wyrazu ciągu: ";
        cin>>n;

        cout<<n<<" wyraz ciągu ma wartość "<<fib(n)<<endl;

        return 0;
}
```

Rozwiązanie rekurencyjne

Przeanalizujmy powyższy algorytm dla $n = 5$

$$wynik = fib(5) = \underbrace{fib(4)}_{\underbrace{fib(3)}_{\underbrace{fib(1)}_{1}+\underbrace{fib(2)}_{1}}+\underbrace{fib(2)}_{1}} + \underbrace{fib(3)}_{\underbrace{fib(1)}_{1}+\underbrace{fib(2)}_{1}} = 5$$

Pozostałe algorytmy

- algorytm Euklidesa
- wyszukiwanie binarne
- sortowanie szybkie
- sortowanie przez scalanie
- przeszukiwanie w głąb
- podnoszenie do potęgi
- wypisanie wyrazu wspak