**Algorytmy i struktury danych**

Instytut Sterowania i Systemów Informatycznych
Wydział Elektrotechniki, Informatyki i Telekomunikacji
Uniwersytet Zielonogórski

# Rekurencja

## 1 Cel ćwiczenia

Ćwiczenie ma na celu zapoznanie studentów z rekurencją. W szczególności omówiony zostanie wpływ rekurencji na wymagania pamięciowe, a co za tym idzie na efektywność programów.

## 2 Definicja rekurencji

Rekurencja (inaczej rekursja - ang. *recursion* ) oznacza odwoływanie się (np.funkcji lub definicji) do samej siebie:

Algorytm 1: *Rekurencja*

```
void rekurencja(parametry)
{
  ...
  rekurencja (parametry); // wyw. rekurencyjne
  ...
}
```

Ze względu na niebezpieczeństwo powtarzania wywołań funkcji w nieskończoność należy przestrzegać pewnych warunków zapewniających poprawność algorytmów zawierających funkcję rekurencyjną. Podstawowy warunek jest następujący: parametry *sterujące* rekurencją w kolejnych wywołaniach funkcji powinny tworzyć ciąg zbieżny do pewnej wartości granicznej, która nie spowoduje kolejnego odwołania rekurencyjnego.

## 3 Przykłady realizacji rekurencyjnej programów

Bazując na powyższych informacjach, przystępujemy do przedstawienia przykładowej realizacji rekurencyjnej programu. Posłużymy się tutaj przykładem obliczania silni. Realizacja bez rekurencji może mieć następującą postać

Algorytm 2: *Iteracyjne obliczanie silni*

```
unsigned long int silnia(unsigned int x)
{
unsigned long int wynik;
unsigned int i;

  wynik=1;
  for (i=1; i <= x; ++i)
    wynik = wynik * i;

  return wynik;
}
```

Następnie, posługując się następującą zależnością

$$x! = (x-1)! \cdot x,$$

która jest prawdziwa dla każdego $x > 0$, możemy przedstawić rekurencyjną wersję funkcji wyznaczającej silnię jej argumentu

Algorytm 3: *Rekurencyjne obliczanie silni*

```
unsigned long int silnia(unsigned int x)
{
unsigned long int wynik;

  if (x > 1) // warunek stopu
     wynik = x * silnia(x-1); // wyw. rekurencyjne
  else
     wynik = 1; // już bez rekurencji

  return wynik;
}
```

Powyższy kod zastosowany np. dla liczby $x = 4$ wykona następujące wywołania pokazane w Tabeli 1.

Kolejnym przykładem zastosowania rekurencji jest algorytm sortowania szybkiego tzw. Quicksort'u. Idea algorytmu QuickSort jest następująca:

1. Dla tablicy *xtab* określ (np. losowo) element $v$.
2. Podziel tablicę *xtab* na dwie podtablice, pierwszą o elementach mniejszych równych $v$, drugą o elementach większych równych $v$.
3. Wywołaj rekurencyjnie siebie niezależnie dla liczb na prawo i na lewo od $v$.

```
|silnia(4)  | pierwsze wywołanie
|4*silnia(3)|
            |silnia(3)   |
            |3*silnia(2) |
                          |silnia(2)   |
                          |2*silnia(1) |
                                       |silnia(1)   |
                                       |1*silnia(0) |
                                                    |silnia(0)|
                                                    |=1       |
                                       |1*1         |
                                       |=1          |
                          |2*1         |
                          |=2          |
            |3*2         |
            |=6          |
|4*6        |
|=24        |
```

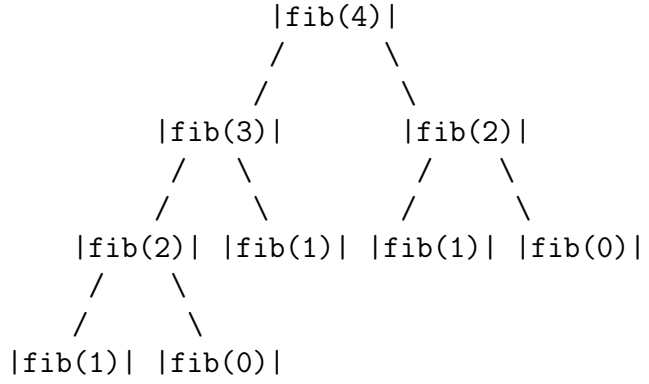
Tabela 1: Obraz wywołań rekurencyjnych dla liczby 4.

Przedstawione powyżej przykłady pokazują, że programy używające rekurencji są bardziej przejrzyste niż programy bez rekurencji. Z drugiej jednak strony, rekurencja prawie zawsze zwiększa pamięciowe zapotrzebowanie programu, przez co zmniejsza się efektywność i następuje wydłużenie czasu wykonywania programu.

W celu zilustrowania tej cechy rekurencji, rozważmy program wyznaczający wartości kolejnych wyrazów ciągu Fibonacciego według następujących reguł

$$fib(0) = 0;$$
$$fib(1) = 1;$$
$$fib(n) = fib(n-1) + fib(n-2), n \geqslant 2.$$

Zakładając, że rozważany program wyznacza wartość dla $n = 4$, to wtedy wykonywane są następujące wywoływania funkcji $fib()$



Łatwo zauważyć, iż w powyższym przykładzie obliczania $fib(4)$ niepotrzebnie jest dwukrotnie obliczana wartość $fib(2)$. Dlatego, ze względu na możliwość zwiększenia zapotrzebowania na pamięć oraz wydłużenie czasu wykonywania programu rekurencyjnego, stosuje się *derekursywację* czyli zamianę programów rekurencyjnych na iteracyjne.

# Literatura

[1] P. Wróblewski: *Algorytmy, struktury danych i techniki programowania*, HELION, 1996.

[2] T. H. Cormen i in.: *Wprowadzenie do algorytmów*, WNT, 2000.

[3] L. Banachowski, K. Diks, W. Rytter: *Algorytmy i struktury danych*, WNT, Warszawa, 1996.

[4] N. Wirth: *Algorytmy + struktury danych = programy*, WNT, Warszawa, 1989.