Teoretyczne podstawy informatyki

Rekursja (rekurencja)

Plan wykładu

1. **Definicje rekurencyjne**
2. **Silnia**
3. **Liczby Fibonacciego**

Definicje rekurencyjne

Definiowanie indukcyjne (rekursja/rekurencja)

definiujemy nieznane przez nieznane, które będzie znane

Definicja nie może być niezrozumiała ani paradoksalna

Definicja rekurencyjna musi zawierać:

- **jedną lub więcej reguł podstawowych definiujących obiekty proste**
- **jedną lub więcej reguł indukcyjnych definiujących większe obiekty w oparciu o mniejsze z tego samego zbioru**

Używanie rekurencji w programach jest możliwe dzięki podprogramom (procedurom, funkcjom) ale nie każdy język programowania, który wspiera podprogramy pozwala na korzystanie z rekurencji (Fortran).

Rekurencja może być:

- **bezpośrednia**

  w ciele P wywoływana jest P (P → P):

```
procedure P;
begin
  { ... }
  P;
  { ... }
end { P };
```

- **pośrednia**

  w ciągu aktywacji podprogramów jest więcej niż jedna instancja P (P → … → … → Q → … → P):

```
procedure P; forward; { deklaracja }

procedure Q;
begin
  { ... }
  P;
  { ... }
end { Q };

procedure P; { definicja }
begin
  { ... }
  Q;
  { ... }
end { P };
```


Silnia

Dekompozycja problemu

```
S(0) = 1                    ⇒ S(n)=1          dla n=0
S(n) = 1*2*3* ... *(n-1)*n
     = S(n-1) * n           ⇒ S(n)=S(n-1)*n   dla n>0
```

Program w Pascalu

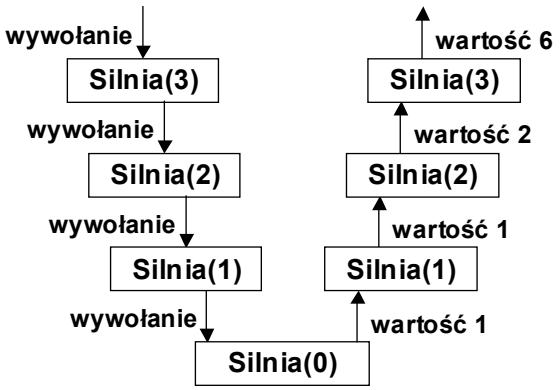
```
dla n = 0:
   S(n)=1
dla n > 0:
   S(n)=S(n-1)*n
```

```
function S(n:integer):integer;
begin
  if n = 0
    then S := 1
    else S := S(n-1) * n;
end { S };

var
  i : integer;

begin
  read(i);
  writeln(S(i));
end { Program }.
```

Ciąg instancji procedur



Wizualizacja wywołań - program w Pascalu

```
var
  i : integer;
function Silnia(n : integer) : integer;
begin
  if n = 0
    then begin
           writeln(' └─> 1':7-(n-i));
           Silnia := 1
         end
    else begin
           writeln(' └─> Silnia(':13-(n-i),
                   n-1,')');
           Silnia := Silnia(n-1) * n;
         end;
end { Silnia };
begin
  writeln(#13#10#13#10);
  read(i);
  write('n=>',i,#13#10,Silnia(i));
end { Program }.
```

Wejście:

```
7
```

Wyjście:

```
n=>7
  └─> Silnia(6)
    └─> Silnia(5)
      └─> Silnia(4)
        └─> Silnia(3)
          └─> Silnia(2)
            └─> Silnia(1)
              └─> Silnia(0)
                └─> 1
```

Obsługa błędnych parametrów wywołania

Co się stanie przy wywołaniu S(-2) ?

S(-2) → S(-3) → S(-4) → S(-5) → S(-6) → S(-7) → … → Stack overflow

podejście 0.1:

```
function S(n:integer):integer;
begin
  if n < 0
    then begin
           writeln('niepoprawne dane');
           Halt(1);
         end
    else if n = 0
           then S := 1
           else S := S(n-1) * n;
end { S };
```

podejście 1.0:

```
function S(n:integer):integer;
  function LiczS(n:integer):integer;
  begin
    if n = 0
      then LiczS := 1
      else LiczS := LiczS(n-1) * n;
  end { LiczS } ;
begin
  if(n < 0)
    then begin
           writeln('niepoprawne dane');
           Halt(1);
         end
    else if n = 0
           then S := 1
           else S := LiczS(n-1) * n;
end { S };
```


Kiedy nie stosować rekursji ?

Wydajność (pamięć i czas) vs prostota zapisu.

Jeżeli program ma strukturę:

P : instrukcja(e)_bez_wywołania_rekurencyjnego
if warunek then instrukcja(e)_z_wywołaniem_rekurencyjnym

to zwykle łatwo go przekształcić na wersję iteracyjną (np. silnia).

Niekiedy można efektywnie przekształcić złożone algorytmy (liczby Fibonacciego) ale niektóre mocno się komplikują (quicksort).

Usuwanie rekursji

dodatkowa zmienna(zmienne) = pamięć „ukryta”

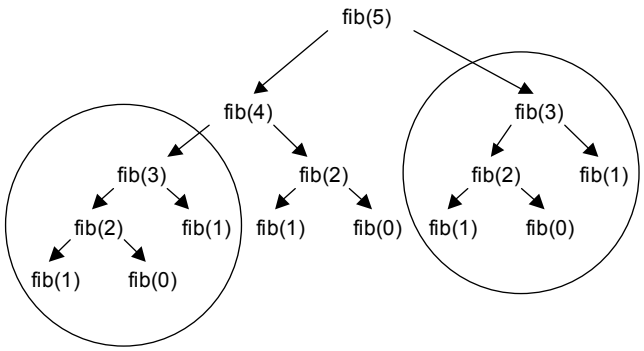
dla n = 0:
S(n)=1
dla n > 0:
S(n)=S(n-1)*n
	function S(n:integer):integer;
	var res : integer;
	begin
	res:=1;
	while n > 0 do { n > 1 }
	begin
	res:=n*res; { S:=n*S; }
	n := n - 1;
	end;
	S:=res;
	end { S };

Liczby Fibonacciego (wady rekursji)

rozwiązanie rekurencyjne:

f(0) = 1,
f(1) = 1,
f(i+2) =
f(i+1) + f(i)
	function fib(j:integer):integer;
	begin
	case j of
	0 : fib:=1;
	1 : fib:=1;
	else fib := fib(j-1)+fib(j-2);
	end;
	end { fib };

Ciąg wywołań



j	wartość funkcji fib(j)	liczba wołań ncall	j	wartość funkcji fib(j)	liczba wołań ncall
1	1	1	11	144	287
2	2	3	12	233	465
3	3	5	13	377	753
4	5	9	14	610	1219
5	8	15	15	987	1973
6	13	25	16	1597	3193
7	21	41	17	2584	5167
8	34	67	18	4181	8361
9	55	109	19	6765	13529
10	89	177	20	10946	21891

rozwiązanie z pomocniczą tablicą

fibArr	1	1	2	3	5	8	13	21	34	55	89	144	…

```
const
  MaxFib = 20;
var
  fibArr : array [ 0 .. MaxFib ] of integer;
function fiba(n : integer) : integer;
var
  i : integer;
begin
  case n of
    0 : fiba := 1;
    1 : fiba := 1;
    else begin
           fibArr[0] := 1;
           fibArr[1] := 1;
           for i := 2 to n do
             fibArr[i]:=fibArr[i-1]+
                        fibArr[i-2];
           fiba := fibArr[n];
         end;
  end;
end { fiba };
var
  n : integer;
begin
  read(n);
  if n <= MaxFib
    then writeln(fiba(n))
    else writeln('Wartosc n zbyt duza');
end { program }.
```

rozwiązanie iteracyjne

```
function fibi(j : integer) : integer;
var
  f, f0, f1 : integer;
begin
  case j of
  0 : f := 1;
  1 : f := 1;
  else
    f0 := 1;
    f1 := 1;
    while j >= 2 do
      begin
        f := f0 + f1;
        f0 := f1;
        f1 := f;
        j := j - 1;
      end;
  end;
  fibi := f;
end { fibi };
```


porównanie kosztów poszczególnych metod (takty CPU)

n	fibr	fiba	fibi
0	500	152	152
1	96	96	96
2	248	204	192
3	348	484	136
4	432	536	132
5	644	288	164
6	1020	288	172
7	1468	292	188
8	2096	284	188
9	3136	276	168
10	4860	304	172
11	7604	208	176
12	12060	216	180
13	19472	224	184
14	31340	224	188
15	50580	240	196
16	81788	248	196
17	132160	272	200
18	213800	284	204
19	345872	304	192