Temat: Rekurencja jako technika kodowania algorytmów.
Anatomia wywołania rekurencyjnego. Rodzaje rekurencji.
Wady i zalety kodowania rekurencyjnego.

Obiekt zwany jest rekurencyjnym (ang. *recursive*), jeżeli częściowo składa się z siebie samego lub jego definicja odwołuje się do jego samego.

Rekurencja (inaczej rekursja) jest silnym narzędziem w definicjach matematycznych.

Przykład 1
Funkcja silnia

$$n! = \begin{cases} 1 & dla \quad n = 0 \\ n \cdot (n-1)! & dla \quad n > 0 \end{cases}$$

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

1. Anatomia wywołania rekurencyjnego

Przykład 2
Funkcja obliczająca potęgę naturalną liczby rzeczywistej

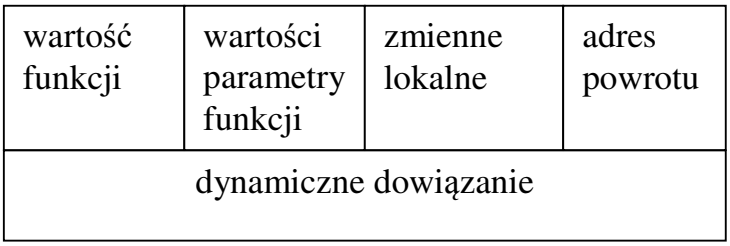
```
power(x, n)
   {
    if  (n==0) then return  1;
    else return x*power(x, n-1);
   }
….

x=power(5.6, 2);
```

call stack – stos wywołań funkcji, w którym zapamiętywane są tzw. rekordy aktywacji (inaczej rekordy wywołania funkcji)

Rekord wywołania (aktywacji) zawiera:

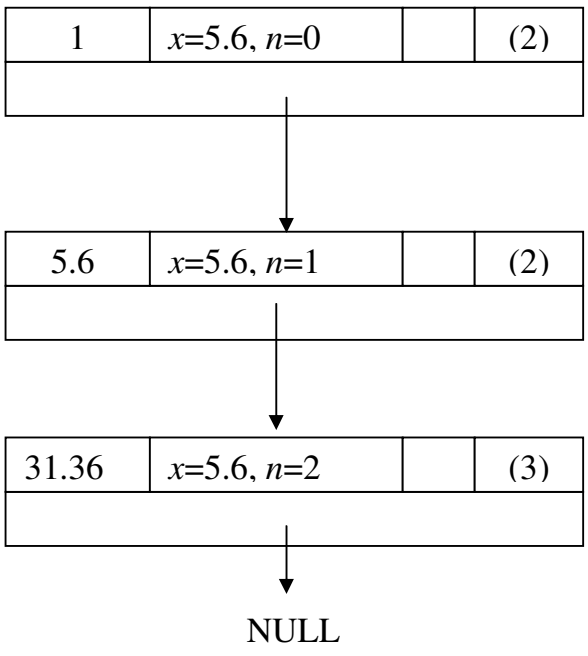
- wartości wszystkich parametrów funkcji,
- zmienne lokalne funkcji,
- adres powrotu (adres miejsca w kodzie programu, do którego wraca procesor po zakończeniu wywołania funkcji,
- dynamiczne dowiązanie (wskaźnik na poprzedni rekord aktywacji)
- wartość funkcji (o ile funkcja zwraca wartość)




```
x=5.6;
          power(x, n)
             {
(1)            if  (n==0) return  1;
(2)                else return x*power(x, n-1);
             }
          ….

(3)       x=power(5.6, 2);
```

Analiza wywołania power(5.6, 2) – stos wywołania



2. Rodzaje rekurencji

Rekursja końcowa nierozgałęziona - zawiera tylko jedno wywołanie rekurencyjne na końcu kodu funkcji. Taki rodzaj rekursji jest po prostu innym zapisem pętli i może być łatwo zastąpiony właśnie pętlą. Rekursja końcowa jest zazwyczaj nieefektywna ze względu na wyższy koszt czasowy i pamięciowy w porównaniu z wersją iteracyjną.

Przykład 3
Funkcja, która wypisuje liczby od i do 1

```
pisz(i)
{
   if (i>0)
        {
           "drukuj i";
          pisz(i-1);
        }
}
```

```
iter_pisz(i)
  {
     for (j=i; j>=1; j--)
        „drukuj j";
  }
```

Rekursja niekońcowa nierozgałęziona- zawiera jedno wywołanie rekurencyjne, które nie jest ostatnim rozkazem funkcji. Iteracyjna wersja takiej funkcji wymaga użycia struktury pomocniczej.

Przykład 4
Funkcja, która wypisuje znaki wprowadzone z klawiatury w rewersie

```
reverse()
{
    char ch;
   „pobierz ch";
    if (ch!=ENTER)
         reverse();
  "drukuj ch";
 }
```

```
iter_reverse()
{
    char ch[1000]; int i=0;
     do{
           „pobierz ch";
             i++;
        } while (ch[i-1]!=ENTER);
     for (int j=i-1;j>=0;j--)
         "drukuj ch[j]";
  }
```

Rekursja rozgałęziona końcowa – zawiera więcej niż jedno wywołanie rekurencyjne na końcu kodu funkcji

Przykład 5

Funkcja, która wyszukuje liczbę w ciągu uporządkowanym $A=\{a_1, a_2, \ldots, a_n\}$. *Zakładamy, że liczby w ciągu się nie powtarzają.*

```
search(l, r)
{
  int m;
  if (l>r) return -1;
  else {
            m=(l+r)/2;
            if (x==a_m) return m;
            else if (x< a_m) return search(l, m-1);
                  else return search(m+1, r);
          }
}
...
wynik=search(0, n-1);

search_iter(A, x, l, r)
{
  int m;
  while  (l<=r)
   {
        m=(l+r)/2;
        if (x== a_m)
              return m;
         else if (x< a_m) r=m-1;
               else return l=m+1;
      }
    return -1;
}
...
wynik=search_iter(A, 5, 0, n-1);
```

Rekursja rozgałęziona niekońcowa – zawiera więcej niż jedno wywołanie rekurencyjne, po niektórych z wywołań albo po każdym z nich znajduje się instrukcje.

Przykład 6

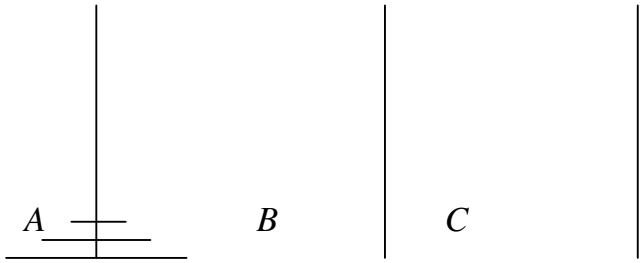
Problem wież Hanoi

Załóżmy, że mamy trzy wieże (kołki) *A*, *B*, *C*. Na pierwszym kołku *A* znajduje się *n* krążków nanizanych w porządku malejących wielkości, podczas, gdy pozostałe kołki są puste.

Należy przenieść krążki z kołka *A* na *B*, być może używając kołka *C*. Reguły przenoszenia krążków są następujące:

- Krążki możemy przenosić po jednym na raz.
- Krążek większy nie może być umieszczony na wierzchu mniejszego.

Rozwiązanie dla $n = 3$



$A \rightarrow B$; $A \rightarrow C$; $B \rightarrow C$; $A \rightarrow B$; $C \rightarrow A$; $C \rightarrow B$; $A \rightarrow B$

Algorytm rozwiązujący problem wież Hanoi

n – liczba krążków, *X* ,*Y, Z* - stojaki

```
Hanoi(n, X, Y, Z)
{
   if (n = =1)  "drukuj X →Y";
   else {
            Hanoi(n-1, X, Z, Y);
            "drukuj X →Y";
            Hanoi(n-1, Z, Y, X);
         }
}
```

Przykładowe wywołanie

Hanoi(3, *A*, *B*, *C*);

Zalety rekurencji:

- upraszcza kodowanie
- zwiększa czytelność kodu

Wady rekurencji:

- spowolnienie wykonania funkcji
- większa zajętość pamięciowa i czasowa – przechowywanie na stosie dodatkowej informacji (rekord aktywacji). Jeśli rekursja jest zbyt głęboka, to może się zdarzyć sytuacja określana jako przepełnienie stosu.

3. Przykład nieuzasadnionego użycia rekurencji

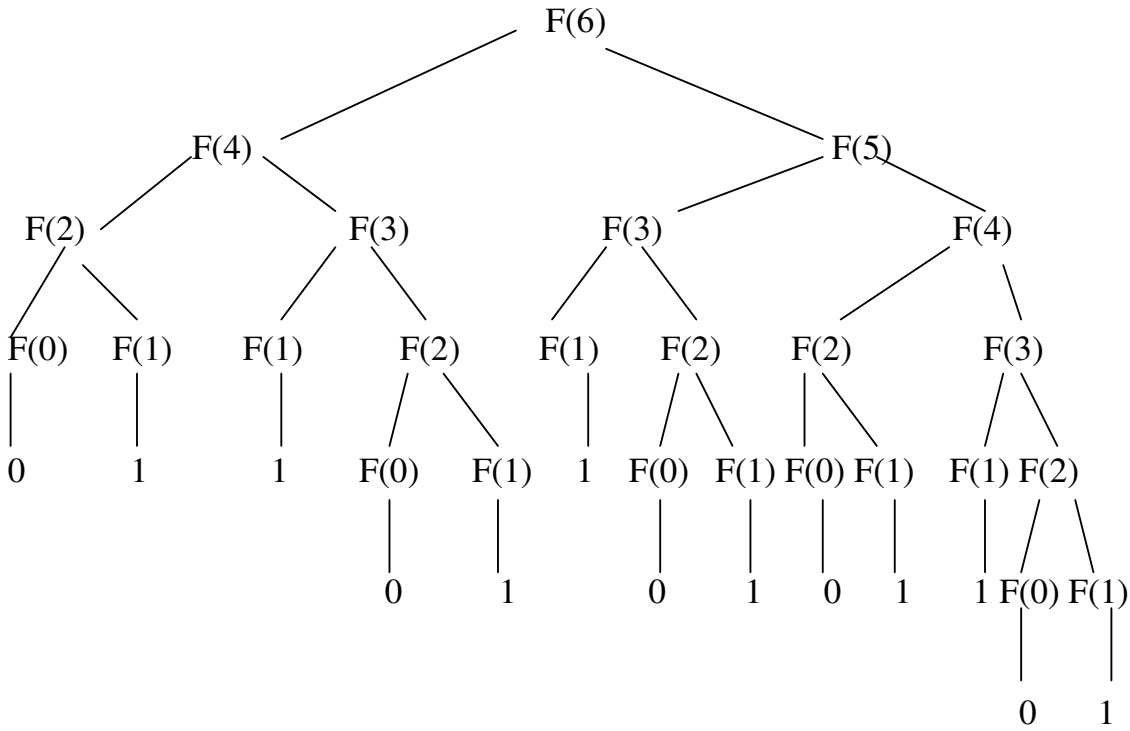
Przykład 7

Funkcja obliczająca wartości elementów ciągu Fibonacciego
Ciąg Fibonacciego: $F(0)=0$
$F(1)=1$
$F(n)=F(n-1)+F(n-2)$ dla $n>1$

Algorytm I (rozwiązanie rekurencyjne)

```
F(n)
{
   if (n==0) return 0;
   else if (n==1) return 1;
   else return F(n-1)+F(n-2);
}
```

Złożoność czasowa: $\Theta(2^n)$



Liczba operacji dodawania i liczba wywołań rekurencyjnych przy obliczaniu liczby Fibonacciego

n	Liczba dodawań	Liczba wywołań
6	12	25
10	88	177
15	986	1973
20	10945	21891
25	121392	242785
30	1346268	2692537

Algorytm II (wersja z tablicą)

```
F(n, T)
{
  T[0]=0;   T[1]=1;
  for (i=2;i<=n;i++)
     T[i]=T[i-1]+T[i-2];
 return T[n];
}
```

Koszt czasowy: $\Theta(n)$

Koszt pamięciowy: $\Theta(n)$

Algorytm III (wersja bez tablicy)

```
F(n)
{
 int a=0,b=1,c;
 for (i=2;i<=n;i++)
   {
      c=a+b;
      a=b;
      b=c;
   }
 return c;
}
```

Koszt czasowy: $\Theta(n)$
Koszt pamięciowy: $\Theta(1)$

Rekurencję stosujemy tylko wtedy, gdy nie wpływa ona znacząco na podniesienie kosztów czasowych i pamięciowych algorytmu, a jest prostsza do zastosownia i czytelniejsza.