Algorytmy i struktury danych

program wykładu

algorytm i dane, struktury danych

notacja algorytmu

symbole oszacowań asymptotycznych

rekurencja i iteracja

Program wykładu

- Pojęcia wstępne
- Rekurencja i iteracja
- Elementarne struktury danych
- Sortowanie i statystyki pozycyjne
- Tablice z haszowaniem
- Struktury drzewiaste
- Drzewa zrównoważone
- Struktury zorientowane na pamięć zewnętrzną
- Złączalne struktury danych
- Przegląd złożonych struktur danych
- Metody konstrukcji algorytmów

Program wykładu

- Przegląd algorytmów:
  - ✓ Algorytmy grafowe
  - ✓ Wyszukiwanie wzorca
  - ✓ Algorytmy geometryczne
- Problemy NP, algorytmy dokładne i przybliżone
- Przeszukiwanie drzew decyzyjnych
- Heurystyki ogólne: przeszukiwanie tabu, symulowane żarzenie, algorytmy genetyczne, sieci neuronowe

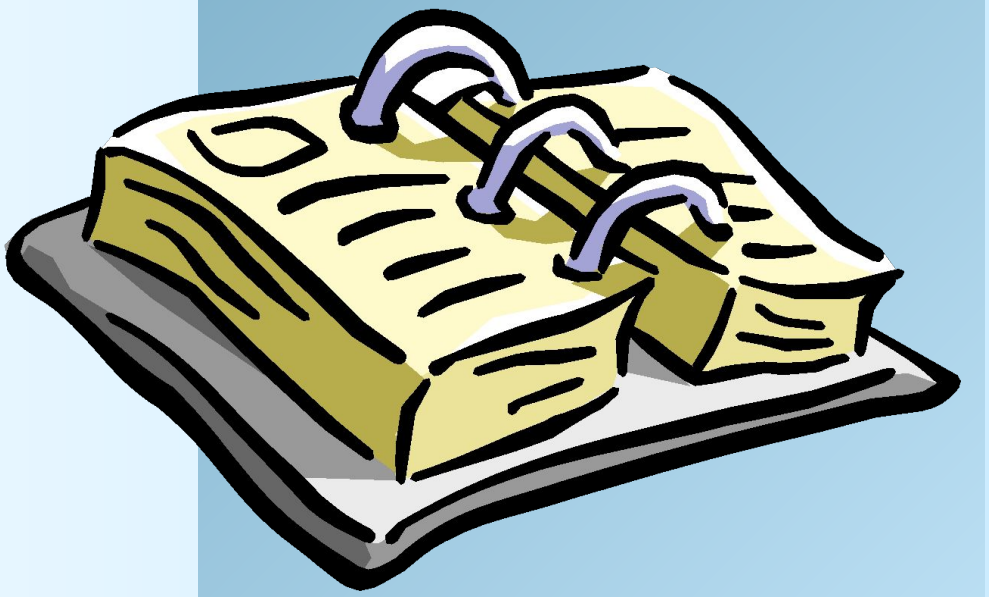
Literatura

- ✓ L.Banachowski i in. „*Algorytmy i struktury danych*”
- ✓ T.Cormen i in. „*Wprowadzenie do algorytmów*”
- ✓ N.Wirth „*Algorytmy+Struktury danych = programy*”

- ✓ L.Banachowski i in. „*Analiza algorytmów i struktur danych*”
- ✓ M.Sysło i in. „*Algorytmy optymalizacji dyskretnej*”
- ✓ Krzysztof Goczyła „*Struktury danych*”

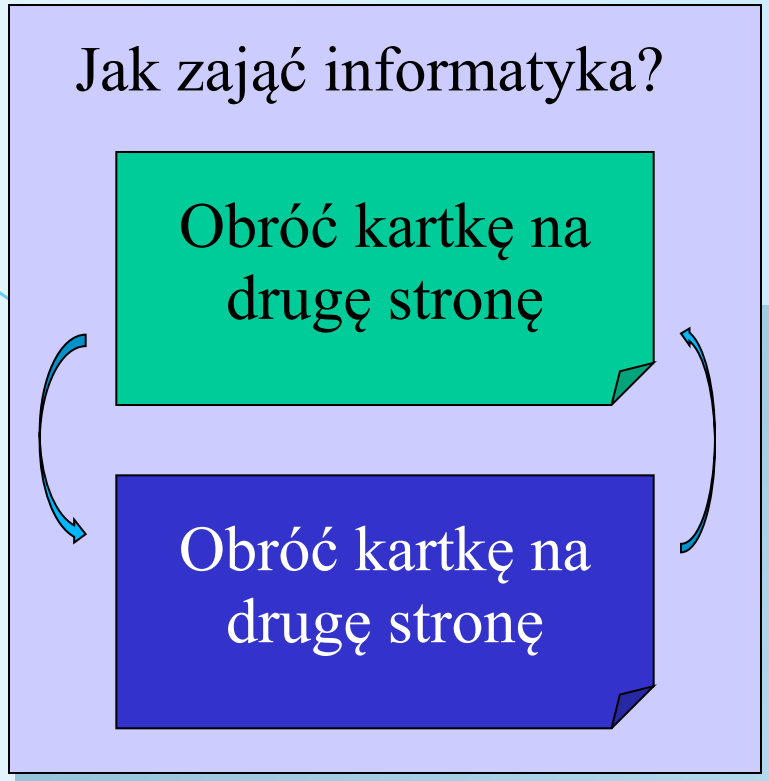
Struktury danych

- ✓ Sposób organizacji danych
- ✓ Przykłady struktur:
  - Tablica
  - Lista
  - Kolejka
  - Stos
  - Lista dwukieunkowa
  - Drzewo
  - Kopiec



Algorytm

skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania



Algorytm:

po skończonej liczbie kroków (czy zawsze i po ilu)

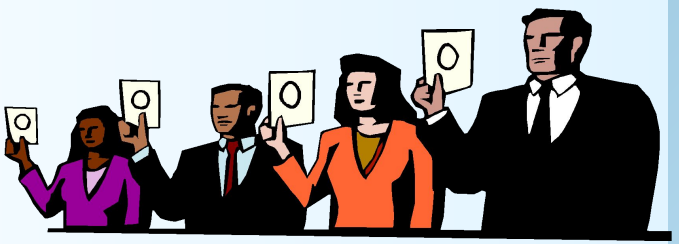
daje poprawny (czy zawsze) i

dokładny (na ile) wynik

dla określonych (czy każdych) danych wejściowych

Notacja algorytmu – język naturalny

Wczytaj dane. Policz sumę wczytanych wartości i podziel przez ich liczbę. Wynik czyli średnią wypisz.



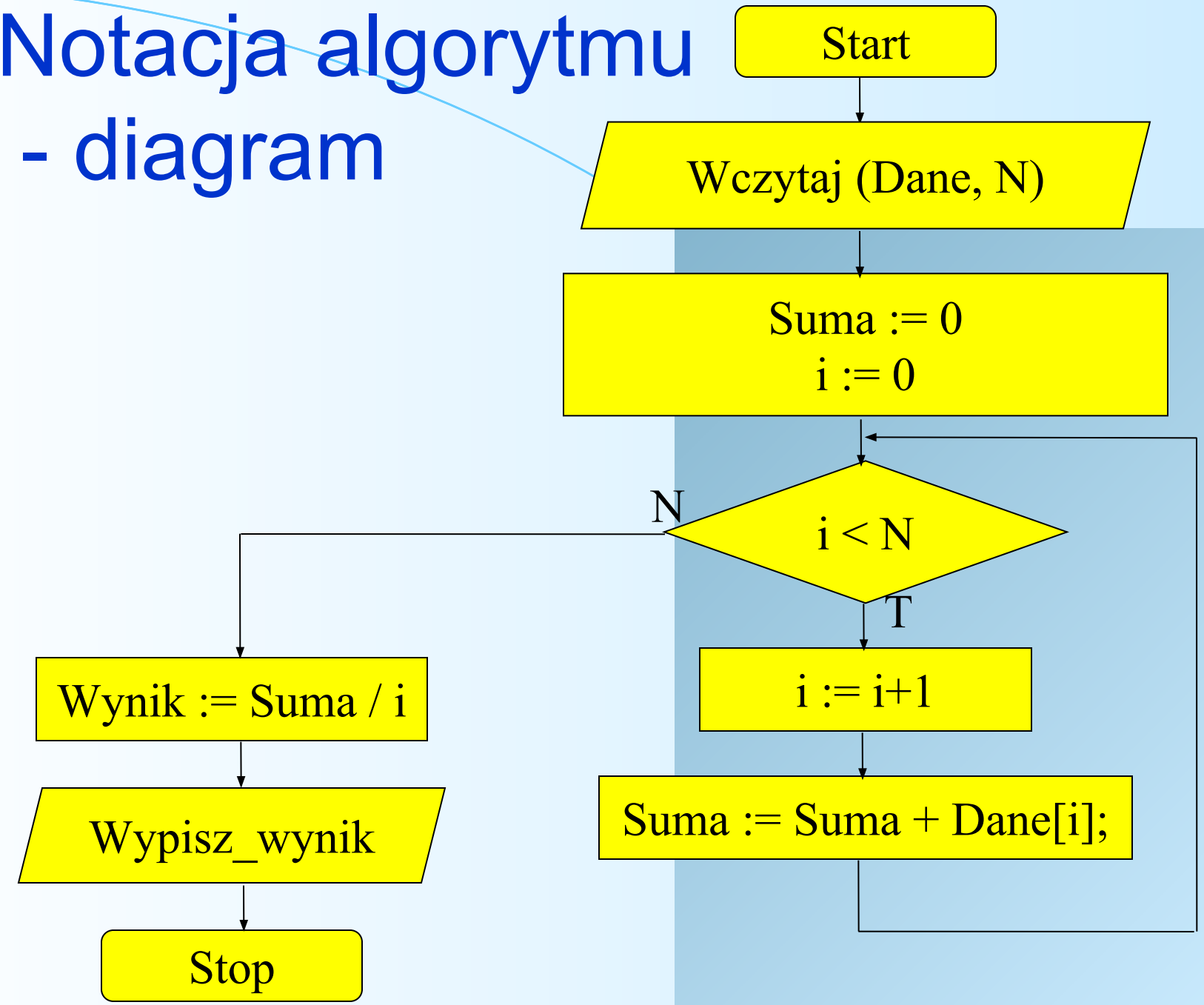
Notacja algorytmu - pseudokod

```
Dane := WczytajDane();
Suma := Sumuj_liczby(Dane);
Licznik := Zlicz_liczby(Dane);
Srednia := Suma / Licznik;
Wypisz_wynik(Srednia);
```

```
Function PoliczSrednia(Dane)
Begin
    Suma := Sumuj_liczby(Dane);
    Licznik := Zlicz_liczby(Dane);
    PoliczSrednia := Suma / Licznik;
End;
```

```
int PoliczSrednia(Dane)
{
    Suma = Sumuj_liczby(Dane) ;
    Licznik = Zlicz_liczby(Dane) ;
    return Suma / Licznik ;
}
```

Notacja algorytmu - diagram



Pseudokod

- podstawienie **=** , porównianie **==**
- instrukcje złożone kończymy **:**
- bloki zaznaczamy przez wcięcie (zamiast {/}, begin/end)
- linie kontynuujemy przez **** na końcu
- komentarz zaczynamy od **#**
- instrukcja warunkowa **if** warunek : czynność
  if warunek : czynność **else** : czynność
- dla skrótu else : if oznaczać będziemy przez elif
- pętla iteracyjna **for** zmienna **in** lista : czynność
- pętla **while** warunek: czynność **else**: czynność
  else oznacza kod wykonany po niespełnionym warunku
- przerwanie pętli **break**, kontynuacja pętli **continue**
- nic nie rób (np. po if) **pass**
- *instrukcje opisowe*

Przykład algorytmu

- ✓ Czy liczba m jest potegą n czyli czy $m = n^k$, gdzie m, n>=2, k∈C

```
def IsPower(m, n):
    p=1
    while p<m:
        p = p*n
    return p==m
```

Czy algorytm ma własność stopu?

Czy algorytm zwraca poprawny wynik?

Formalny dowód poprawności jest czesto bardzo trudny ...

Czasem nawet nie wiadomo

- ... czy algorytm się kończy?

- Problem Colatza – czy poniższy algorytm się kończy dla dowolnej liczby naturalnej

```
def Collatz(x):
    while x>1:
        if x%2==0:
            x = x//2
        else:
            x = x*3+1
```

Po co są algorytmy

...

aby

rozwiązywać

problemy

☺

Po co wymyślać nowe algorytmy

...

aby

rozwiązywać

problemy

lepiej

szybciej, dokładniej, łatwiej

Przykład: dane kontrahentów I

- ✓ Struktura danych : nieuporządkowany “plik“ kartek

```
def AddNew(phoneBook, newEntry):
    append_new_entry(phoneBook, newEntry)

def Find(phoneBook, findWhat):
    current = None
    while not is_end(phoneBook) and \
            current != findWhat:
        current = get_next(phoneBook)
    if current == entryTofind:
        return current
    else:
        return None
```

Czy to działa?

Zamiast

- `current == findWhat`

w praktyce powinno być

- `current.name == nameToFind`
- `current.key == keyToFind`
- `is_like(current,findWhat)`

- Wyzerowanie stanu phoneBook

Przykład: dane kontrahentów I

- ✓ Struktura danych : nieuporządkowany “plik“ kartek

```
def AddNew(phoneBook,newEntry):
    append_new_entry(phoneBook,newEntry)

def Find(findWhat):
    current = get_first(phoneBook)
    while not is_end(phoneBook) \
        and current != findWhat:
         current = get_next(phoneBook)
    if current != findWhat:
        return None
    else:
        return current
```

Przykład: dane kontrahentów II

- ✓ Struktura danych : posortowane kartki

```
def SearchBin(phoneBook, minEl, maxEl, whatFind):
      if minEl >= maxEl :
    if minEl >= len(phoneBook):
       return minEl
    if phoneBook[minEl] >= whatFind :
          return minEl
         else:
       return minEl +1
    mid = (maxEl + minEl) // 2
      if whatFind < phoneBook[mid] :
         return SearchBin(phoneBook, minEl, mid-1,
  whatFind)
      elif whatFind > phoneBook[mid] :
         return SearchBin(phoneBook, mid+1, maxEl,
  whatFind)
```

Przykład: dane kontrahentów II

```
def AddNew(phoneBook, newEntry):
    position = Find(phoneBook, newEntry)
    if position < len(phoneBook) and \
            phoneBook[position] == newEntry:
        print("kontrahent już istnieje“)
    else:
        insert_new_entry(phoneBook, position, newEntry)


def Find(phoneBook, whatFind):
    return SearchBin(phoneBook,0,len(phoneBook),whatFind)
```

Porównanie algorytmów

- ✓ Pierwszy algorytm jest prostszy
- ✓ Drugi algorytm wymaga struktury o dostępie swobodnym (np. tablicy, drzewa) lub wymaga przepisywania elementów

A szybkość?

Porównanie algorytmów

- ✓ Ktory algorytm jest szybszy?
  - Czasy wykonania zależą od procesora, jakości kompilatora itd.
  - Liczba operacji uniezależnia nas od szybkości komputera
- ✓ Liczba operacji dla pierwszego algorytmu:
  - Dodawanie elementu => 1 operacja
  - Przeszukiwania z sukcesem => średnio N/2, najgorzej N
  - Przeszukiwania z porażką => N operacji
- ✓ Liczba operacji dla drugiego algorytmu:
  - Przeszukiwania z porażką => log N operacji
  - Przeszukiwania z sukcesem => log N operacji
  - Dodawanie elementu => log N lub log N + N/2, w zależności od czasu wstawanie elementu

Co mogą znaczyć te liczby

- ✓ Algorytm I
  - Dla 100 i 10 poszukiwań
    - Tworzenie pliku: T = 100
    - Szukanie: T <= 10*100 = 1000
  - Dla 100 i 100 poszukiwań
    - Tworzenie pliku: T=100
    - Szukanie: T <= 100*100 = 10000
- ✓ Algorytm II
  - Dla 100 i 10 poszukiwań
    - Tworzenie pliku: $T <= 100*\log_2 100 \sim 650$
    - Szukanie: $T <= 10* \log_2 100 \sim 65$
  - Dla 100 i 100 poszukiwań
    - Tworzenie pliku: $T <= 100*\log_2 100 \sim 650$
    - Szukanie: $T <= 100* \log_2 100 \sim 650$

Dlaczego tylko mogą znaczyć ?

Mamy 2 algorytmy, które wykonają odpowiednio
100 x operacji oraz $10\ x^2$ operacji.

Który jest lepszy?

$$100\ c_1\ x \quad ? \quad 10\ c_2\ x^2$$

$$c_1'\ x \quad ? \quad c_2'\ x^2$$

Tempo wzrostu

- ✓ Zamiast pytania - co jest większe:
  - $10X + 100$
  - $100X + 10$
  - X^2
- ✓ Pytamy co szybciej rośnie, np:
  - $\ln x$
  - X
  - X^2

W kontekście algorytmu pytamy jak wzrośnie czas wykonania jeśli dane wzrosną o 1 lub 2 razy itd.

Ćwiczenie

- ✓ Uszerguj wg. tempa wzrostu:
  - $\ln x$, $\lg x$, $\log_2 x$

- ✓ Uszerguj wg. tempa wzrostu:
  $\log x$, x, $\sqrt{x}$, x^2, $x\sqrt{x}$, 2^x, e^x, $x!$, x^x

Symbol Θ

Θ($g(x)$) oznacza zbiór takich funkcji $f(x)$, że

istnieją dodatnie stałe c_1, c_2, n_0,
takie że dla wszystkich $n >= n_0$

$$0 <= c_1 * g(n) <= f(n) <= c_2 * g(n)$$

Mówimy, że $f(x) \in \Theta(g(x))$ lub w skrócie $f(x) = \Theta(g(x))$

Przykłady: $10x^2 + 5x - 7 = \Theta(x^2)$, $\ln x^2 = \Theta(\log x)$

Symbol O

O($g(x)$) oznacza zbiór takich funkcji $f(x)$, że

istnieją dodatnie stałe c, n_0,
takie, że dla wszystkich $n >= n_0$

$$0 <= f(n) <= c*g(n)$$

Mówimy, że $f(x) \in O(g(x))$ lub w skrócie $f(x) = O(g(x))$

Przykłady: $x^2 = O(x^2)$, $x = O(x^2)$

Symbol Ω

$\Omega(g(x))$ oznacza zbiór takich funkcji $f(x)$, że

istnieją dodatnie stałe c, n_0,
takie, że dla wszystkich $n >= n_0$

$$0 <= c*g(n) <= f(n)$$

Mówimy, że $f(x) \in \Omega(g(x))$ lub w skrócie $f(x) = \Omega(g(x))$

Przykłady: $x^2 = \Omega(x^2)$, $x^2 * \ln x = O(x^2)$

Symbol o

o($g(x)$) oznacza zbiór takich funkcji $f(x)$, że

dla każdej dodatniej stałej c istnieje dodatnie n_0, takie, że dla wszystkich $n >= n_0$

$$0 <= f(n) < c*g(n)$$

Mówimy, że $f(x) \in o(g(x))$ lub w skrócie $f(x) = o(g(x))$

Przykłady: $x^2 + x \neq o(x^2)$, $\ln x = o(x)$

Symbol ω

ω($g(x)$) oznacza zbiór takich funkcji $f(x)$, że

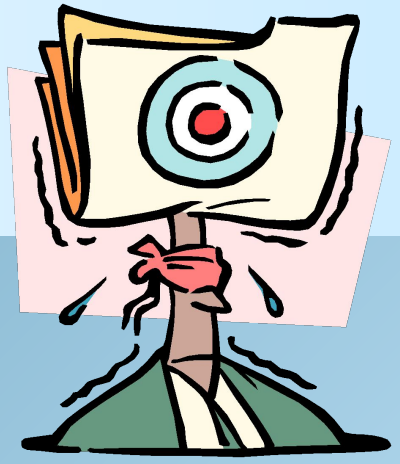
dla każdej dodatniej stałej c istnieje dodatnie n_0, takie, że dla wszystkich $n >= n_0$

$0 <= c*g(n) < f(n)$

Mówimy, że $f(x) \in \omega(g(x))$ lub w skrócie $f(x) = \omega(g(x))$

Przykłady: $x^2 \neq \omega(x^2)$, $x^2 * \ln x = \omega(x^2)$

Symbole oszacowań - podsumowanie



- ✓ o –> wolniej niż
- ✓ O –> nie szybciej niż
- ✓ ω -> szybciej niż
- ✓ Ω -> nie wolniej niż
- ✓ Θ -> tak samo

- ✓ $f(n) = o(g(n))$ oznacza, że $\lim f(n)/g(n) = 0$ dla $n \to \infty$
- ✓ $f(n) = \omega(g(n))$ oznacza, że $\lim f(n)/g(n) = \infty$ dla $n \to \infty$

Czym różnią się algorytmy ?

- ✓ Szybkość działania (złożoność obliczeniowa czasowa)
- ✓ Zajętość pamięci (złożoność pamięciowa)
- ✓ Dokładność
- ✓ Dziedzina (dane dla których alg. jest poprawny)
- ✓ Komplikacja algorytmu
- ✓ Stabilność numeryczna

Złożoność obliczeniowa

- ✓ Czasowa: Liczba operacji wyrażona jako funkcja rozmiaru danych
- ✓ Pamieciowa: Rozmiar dodatkowej pamięci (bez danych wejściowych) wymagany przez algorytm wyrażony jako funkcja rozmiaru danych

Iteracja

- ✓ Klasyczne algorytmy wykorzystujące pętle, zmienne chwilowe itd, ale nie zawierające wołania „samego siebie”

```
def FactorialIter(n):
    ret = 1
    while n>1:
        ret = ret * n
        n = n-1
    return ret
```

- ✓ Liczba operacji (złożoność) odpowiada liczbie wykonań najbardziej zagnieżdzonej pętli

Rekurencja

- ✓ Funkcja wywołuje samą siebie
  1. Rekurencja jawna - podprogram woła samego siebie

```
def f(n):
    f(n-1)
```

  2. Rekurencja niejawna

```
def a(n):
    b(n/2)
def b(n):
    a(n-1)
```

- ✓ Liczba operacji?

Algorytmy rekurencyjne

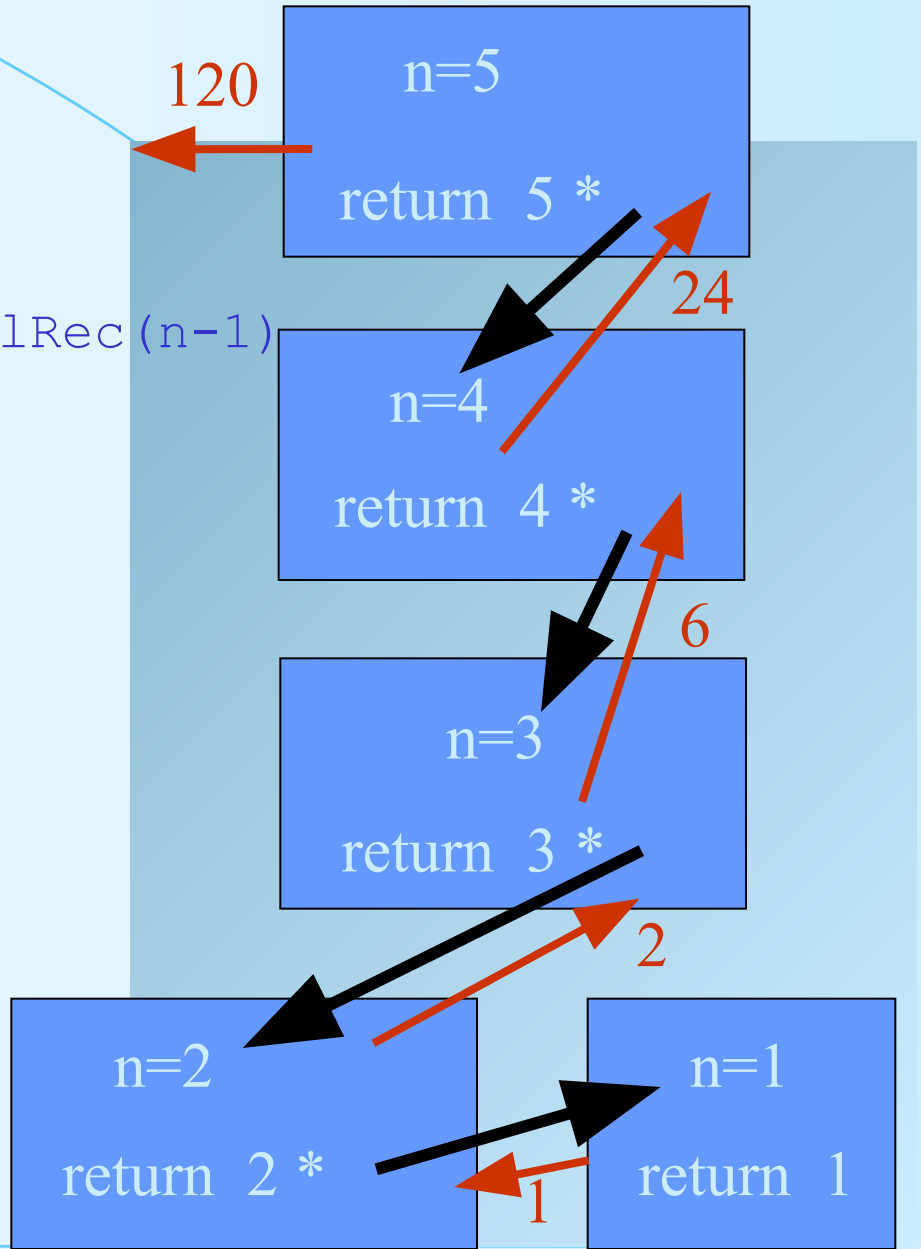
- ✓ Często algorytmy rekurencyjne mają charakter bardziej zwarty niż analogiczne alg. iteracyjne

UWAGA na warunek wyjścia

```
def f():
    f() # brak warunku zakonczenia =
        # rekurencja nieskończona
```

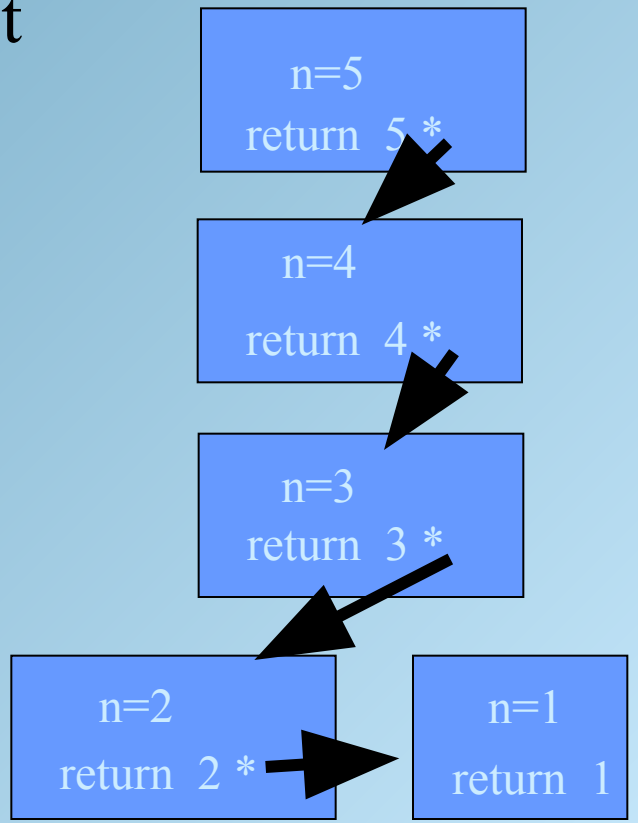
Algorytmy rekurencyjne - silnia

```
def FactorialRec (n):
    if n <=1 :
        return 1
    else:
        return n * FactorialRec(n-1)
```



Głębokość rekurencji

- ✓ Każda instancja funkcji zajmuje miejsce na stosie
- ✓ Instancja = zmienne lokalne + adres powrotu
- ✓ Rozmiar stosu w C/C++ jest silnie ograniczony
- ✓ Problem mogą stanowić
  - duże zmienne lokalne
  - duże parametry (struktury są kopiowane)
  - tablice lokalne



Przykład algorytmu

- ✓ Największy wspólny dzielnik liczb m i n, gdzie m>=n>0 oraz m,n$\in$C

```
def GCD(m,n):
    while n!=0:
        tmp = m % n
        m = n
        n = tmp
    return m
```

Czy algorytm ma własność stopu?

Czy algorytm zwraca poprawny wynik?

Przykład algorytmu

- ✓ Największy wspólny dzielnik liczb m i n, gdzie m>=n>0, m,n∈C

```
def NWD(m,n):
    if m%n == 0:
        return n
    else:
        return NWD(n, m % n)
```

Czy algorytm ma własność stopu?

Czy algorytm zwraca poprawny wynik?

Rekurencja vs. iteracja

- ✓ Każdy algorytm rekurencyjny mozna zapisać w sposób iteracyjny i odwrotnie

- ✓ Potencjalne problemy związane z rekurencją:
  - głębokość rekurencji (→ złożoność pamięciowa)
  - nadmiarowe obliczenia (→ CZAS OBLICZEŃ)

Rekurencja vs. iteracja

```
def FibRec(n):
    if n<=2:
       return 1
    else:
       return FibRec(n-1)+FibRec(n-2)


def FibIter(n)
    cur=1
    last=1
    while n>2:
       n=n-1
       last, cur = cur, cur+last
    return cur
```

Która wersja jest szybsza proszę sprawdzić w domu np. dla 500?