# Teoretyczne podstawy informatyki

## Wykład 3a
## Złożoność obliczeniowa algorytmów

# Złożoność obliczeniowa i asymptotyczna

## Złożoność obliczeniowa:

Miara służąca do porównywania efektywności algorytmów.
Mamy dwa kryteria efektywności

- **czas**
- **pamięć**

Do oceny efektywności stosujemy jednostki logiczne wyrażające związek miedzy **rozmiarem danych** N (wielkość pliku lub tablicy) a **ilością czasu** T potrzebną na ich przetworzenie.

Funkcja wyrażająca zależność miedzy N a T jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych

➢ przybliżona miara efektywności czyli tzw. złożoność asymptotyczna

# Szybkość wzrostu poszczególnych składników funkcji

n – ilość wykonywanych operacji

**Funkcja:** $f(n) = n^2 + 100 \cdot n + \log_{10} n + 1000$

| n | f(n) | $n^2$ | $100 \cdot n$ | $\log_{10} n$ | 1000 |
|---|---|---|---|---|---|
| 1 | 1 101 | 0.1% | 9% | 0.0% | 91% |
| 10 | 2 101 | 4.8% | 48% | 0.05% | 48% |
| 100 | 21 002 | 48% | 48% | 0.001% | 4.8% |
| $10^3$ | 1 101 003 | 91% | 9% | 0.0003% | 0.09% |
| $10^4$ | | 99% | 1% | 0.0% | 0.001% |
| $10^5$ | | 99.9% | 0.1% | 0.0% | 0.0000% |

- Dla dużych wartości n, funkcja rośnie jak $n^2$, pozostałe składniki mogą być zaniedbane.

# Notacja „wielkie *O* "

(wprowadzona przez P. Bachmanna w 1894r)

**Definicja:**
f(n) jest *O* (g(n)), jeśli istnieją liczby dodatnie c i N takie że, $f(n) < c \bullet g(n)$ dla wszystkich $n \geq N$.

Przykład:
$f(n) = n^2 + 100*n + \log_{10} n + 1000$
możemy przybliżyć jako
$f(n) \sim n^2 + 100*n + O(\log_{10} n)$
albo jako
$f(n) \sim O(n^2)$

Notacja **„wielkie *O* "** ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów

# Własności notacji „wielkie O"

**Własność 1 (przechodniość)**
Jeśli f(n) jest $O(g(n))$ i g(n) jest $O(h(n))$, to f(n) jest $O(h(n))$

**Własność 2**
Jeśli f(n) jest $O(h(n))$ i g(n) jest $O(h(n))$, to f(n)+g(n) jest $O(h(n))$

**Własność 3**
Funkcja $an^k$ jest $O(n^k)$

**Własność 4**
Funkcja $n^k$ jest $O(n^{k+j})$ dla dowolnego dodatniego j

- Z tych wszystkich własności wynika, że dowolny wielomian jest **„wielkie *O*"** dla n podniesionego do najwyższej w nim potęgi, czyli
  $f(n) = a_k n^k + a_{k-1} n^{k-1} + ..... + a_1 n + a_0$ jest $O(n^k)$
  (jest też oczywiście $O(n^{k+j})$ dla dowolnego dodatniego j)

# Własności notacji „wielkie O"

**Własność 5**
Jeśli f(n)=c g(n), to f(n) jest $O(g(n))$

**Własność 6**
Funkcja $\log_a n$ jest $O(\log_b n)$ dla dowolnych a i b większych niż 1

**Własność 7**
$\log_a n$ jest $O(\log_2 n)$ dla dowolnego dodatniego a

Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest **funkcja logarytmiczna**. Jeżeli można wykazać że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry. Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak $O(\log_2 \log_2 n)$ czy $O(1)$ ma praktyczne znaczenie.

# Notacja Ω i Θ

Notacja „wielkie *O*” odnosi się do górnych ograniczeń funkcji. Istnieje symetryczna definicja dotycząca dolnych ograniczeń:

**Definicja**
f(n) jest $\Omega(g(n))$, jeśli istnieją liczby dodatnie c i N takie że, $f(n) \geq c\, g(n)$ dla wszystkich $n \geq N$.

**Równoważność**
f(n) jest $\Omega(g(n))$ wtedy i tylko wtedy, gdy g(n) jest $O(f(n))$

**Definicja**
f(n) jest $\Theta(g(n))$, jeśli istnieją takie liczby dodatnie $c_1$, $c_2$ i N takie że, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla wszystkich $n \geq N$.

# Możliwe problemy

Celem wprowadzonych wcześniej sposobów zapisu (notacji) jest porównanie efektywności rozmaitych algorytmów zaprojektowanych do rozwiązania tego samego problemu.
Jeżeli będziemy stosować tylko notacje „wielkie *O*" do reprezentowania złożoności algorytmów, to niektóre z nich możemy zdyskwalifikować zbyt pochopnie.

***Przykład***

*Załóżmy, że mamy dwa algorytmy rozwiązujące pewien problem, wykonywana przez nie liczba operacji to odpowiednio $10^8 n$ i $10n^2$. Pierwsza funkcja jest $O(n)$, druga $O(n^2)$.*

*Opierając się na informacji dostarczonej przez notacje „wielkie O" odrzucilibyśmy drugi algorytm ponieważ funkcja kosztu rośnie zbyt szybko.*

*To prawda ..... ale dopiero dla odpowiednio dużych n, ponieważ dla $n<10^7$ drugi algorytm wykonuje mniej operacji niż pierwszy.*

*Istotna jest więc też stała ($10^8$), która w tym przypadku jest zbyt duża aby notacja była znacząca.*

# Przykłady rzędów złożoności

Algorytmy można klasyfikować ze względu na złożoność czasową lub pamięciową. W związku z tym wyróżniamy wiele klas algorytmów.

- **Algorytm stały:** czas wykonania pozostaje taki sam niezależnie od ilości przetwarzanych elementów.
- **Algorytm kwadratowy:** czas wykonania wynosi $O(n^2)$.
- **Algorytm logarytmiczny**: czas wykonania wynosi $O(\log n)$
- itd...

➢ **Analiza złożoności algorytmów jest niezmiernie istotna** i nie można jej lekceważyć argumentując potencjalną szybkością obliczeń komputera.
Nie sposób jej przecenić szczególnie w kontekście doboru struktury danych.

# Klasy algorytmów i ich czasy wykonania na komputerze działającym z szybkością 1 instrukcja/μs

| klasa | złożoność | liczba operacji i czas wykonania | | | |
|---|---|---|---|---|---|
| | n | 10 | | $10^3$ | |
| stały | $O(1)$ | 1 | 1μs | 1 | 1 μs |
| logarytm | $O(\log n)$ | 3.32 | 3μs | 9.97 | 10 μs |
| liniowy | $O(n)$ | 10 | 10μs | $10^3$ | 1ms |
| kwadratowy | $O(n^2)$ | $10^2$ | 100μs | $10^6$ | 1s |
| wykładniczy | $O(2^n)$ | 1024 | 10ms | $10^{301}$ | >> $10^{16}$ lat |

# Funkcje niewspółmierne

- Bardzo wygodna jest możliwość porównywania dowolnych funkcji f(n) i g(n) z pomocą notacji „duże O”
  - albo f(n) = O(g(n))
  - albo g(n) = O(f(n))

  Albo jedno i drugie czyli f(n) = $\Theta$(g(n)).
- Istnieją pary funkcji niewspółmiernych (ang. *incommensurate*), z których żadne nie jest „dużym O” dla drugiej.

**Przykład**

Rozważmy funkcję f(n) równą *n* dla nieparzystych *n* oraz $n^2$ dla parzystych *n. Oznacza to, że*

*f(1)=1, f(2)=4, f(3)=3, f(4)=16, f(5)=5 itd..*

*Podobnie, niech g(n) będzie równe $n^2$ dla nieparzystych n oraz n dla parzystych n. W takim przypadku, funkcja f(n) nie może być O(g(n)) ze względu na parzyste argumenty n, analogicznie g(n) nie może być O(f(n) ze względu na nieparzyste elementy n.*

*Obie funkcje mogą być ograniczone jako O($n^2$).*

# Analiza czasu działania programu

Mając do dyspozycji definicję „duże O” oraz własności (1)-(7) będziemy mogli, wg. kilku prostych zasad, skutecznie analizować czasy działania większości programów spotykanych w praktyce.

Efektywność algorytmów ocenia się przez szacowanie ilości czasu i pamięci potrzebnych do wykonania zadania, dla którego algorytm został zaprojektowany.

Najczęściej jesteśmy zainteresowani ***złożonością czasową***, mierzoną zazwyczaj liczbą przypisań i porównań realizowanych podczas wykonywania programu.

Bardzo często interesuje nas tylko ***złożoność asymptotyczna***, czyli czas działania dla dużej ilości analizowanych zmiennych.

# Czas działania instrukcji prostych

## *Czas dzialania instrukcji prostych*

Przyjmujemy zasadę że czas działania pewnych prosty operacji na danych wynosi O(1), czyli jest niezalezny od rozmiaru danych wejściowych.

- Operacje arytmetyczne, np.. (+), (-)
- Operacje logiczne (&&)
- Operacje porównania (<=)
- Operacje dostępu do struktur danych, no. Indeksowanie tablic (A[i])
- Proste przypisania, np.. Kpopiowanie wartości do zmiennej.
- Wywołania funkcji biblotecznych, np.. Scanf lub printf

Każda z tych operacji można wykonać za pomocą pewnej (niewielkiej) liczby rozkazów maszynowych.

# Czas działania pętli „for"

***Przykład 1:*** *Prosta pętla*

*for (i=sum=0; i<n; i++) sum+=a[i];*

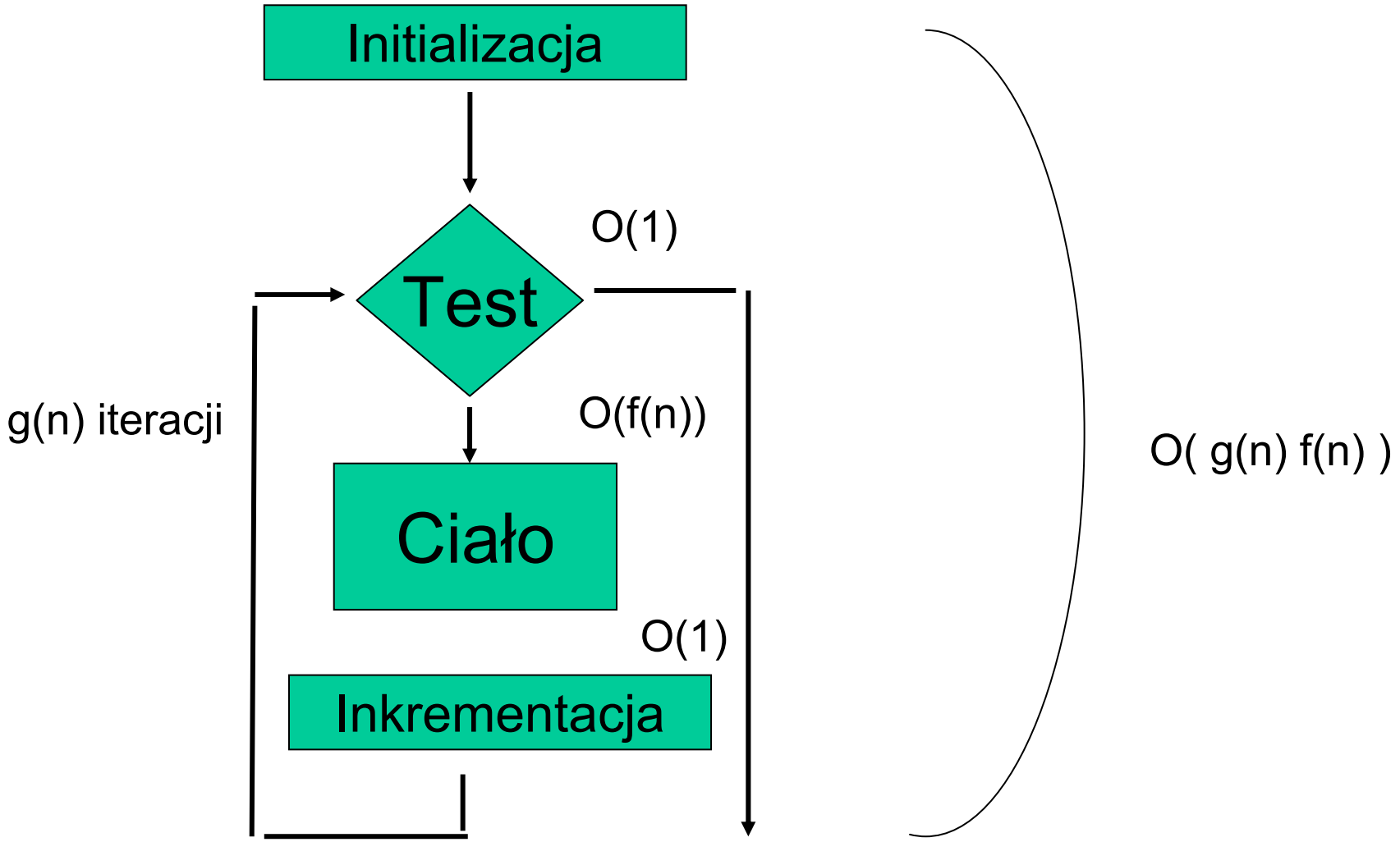
Powyższa **pętla powtarza się n razy**, podczas każdego jej przebiegu realizuje **dwa przypisania**:

- aktualizujące zmienną „sum"
- zmianę wartości zmiennej „i"

Mamy zatem 2n przypisań podczas całego wykonania pętli;

Złożoność asymptotyczna algorytmu **jest *O*(n).**

# Czas działania instrukcji „for"



# Czas działania pętli „for"

***Przykład 2:*** *Pętla zagnieżdżona*

```
for  (i=0; i<n; i++)     {
        for (j=1, sum=a[0]; j<=i; j++)
             sum+=a[j];  }
```

Na samym początku zmiennej „i" nadawana jest wartość początkowa. **Pętla zewnętrzna powtarza się n razy**, a w każdej jej iteracji wykonuje się wewnętrzna pętla oraz instrukcja przypisania wartości zmiennym „i", „ j", „ sum".
Pętla wewnętrzna wykonuje się „i" razy dla każdego $i \in \{1,...,n-1\}$, a na każdą iteracje przypadają dwa przypisania: jedno dla „sum", jedno dla„j".

Mamy zatem: $1+3n+2(1+2+...+n-1) = 1+3n+n(n-1) = O(n)+O(n^2) = O(n^2)$ przypisań wykonywanych w całym programie.
**Złożoność asymptotyczna algorytmu jest $O(n^2)$.** Pętle zagnieżdżone maja zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.

# Czas działania pętli „for"

***Przykład 3***

*Znajdź najdłuższą podtablice zawierającą liczby uporządkowane rosnąco.*

```
for (i=0; len=1; i<n-1; i++) {
        for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);
            if(len < i2-i1+1)  len=i2-i1+1;
}
```

Jeśli liczby w tablicy są uporządkowane malejąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się tylko raz. **Złożoność asymptotyczna algorytmu jest więc $O(n)$.**

Jeśli liczby w tablicy są uporządkowane rosnąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się i razy dla $i \in \{1,...,n-1\}$. **Złożoność asymptotyczna algorytmu jest więc $O(n^2)$.**

# Czas działania pętli „for"

Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą ale bardzo istotna.
Staramy się wyznaczy złożoność

- w „przypadku optymistycznym",
- „przypadku pesymistycznym"
- oraz „przypadku średnim".

Często posługujemy się przybliżeniami opartymi o notacje „wielkie $O$, $\Omega$ i $\Theta$".

# Czas działania instrukcji warunkowych

Instrukcje warunkową if-else zapisuje się w postaci

```
if (<warunek>)
 <blok-if>
else
 <blok-else>
```

Gdzie

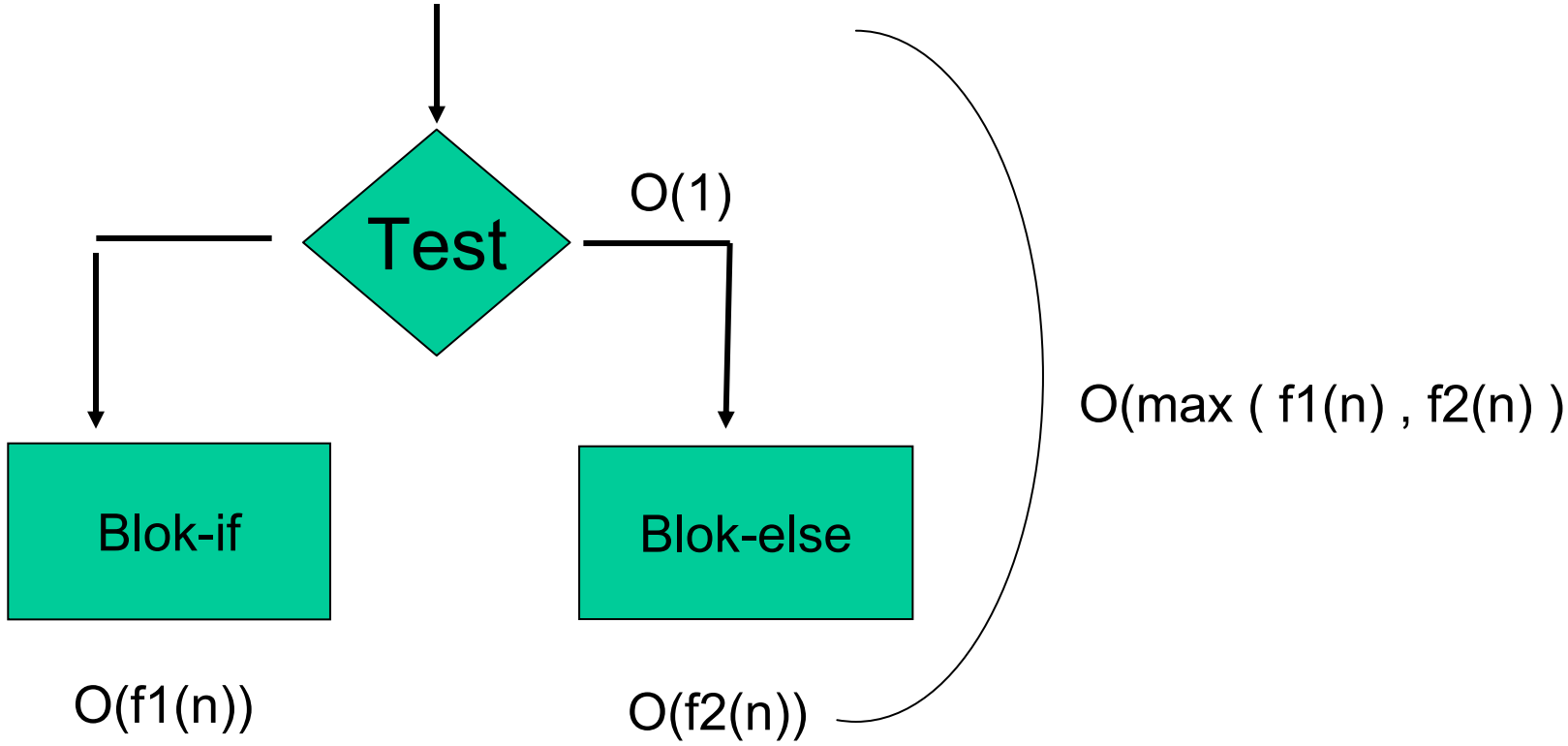
(1) **<warunek>** jest wyrażeniem które trzeba obliczyć. Warunek niezależnie od tego jak skomplikowany wymaga wykonania stałej liczby operacji chyba ze zawiera wywolanie funkcji, więc czasu **O(1)**

(2) **<blok-if>** zawiera insrukcje wykonywane tylko w przypadku gdy warunek jest prawdziwy, czas działania **f(n)**

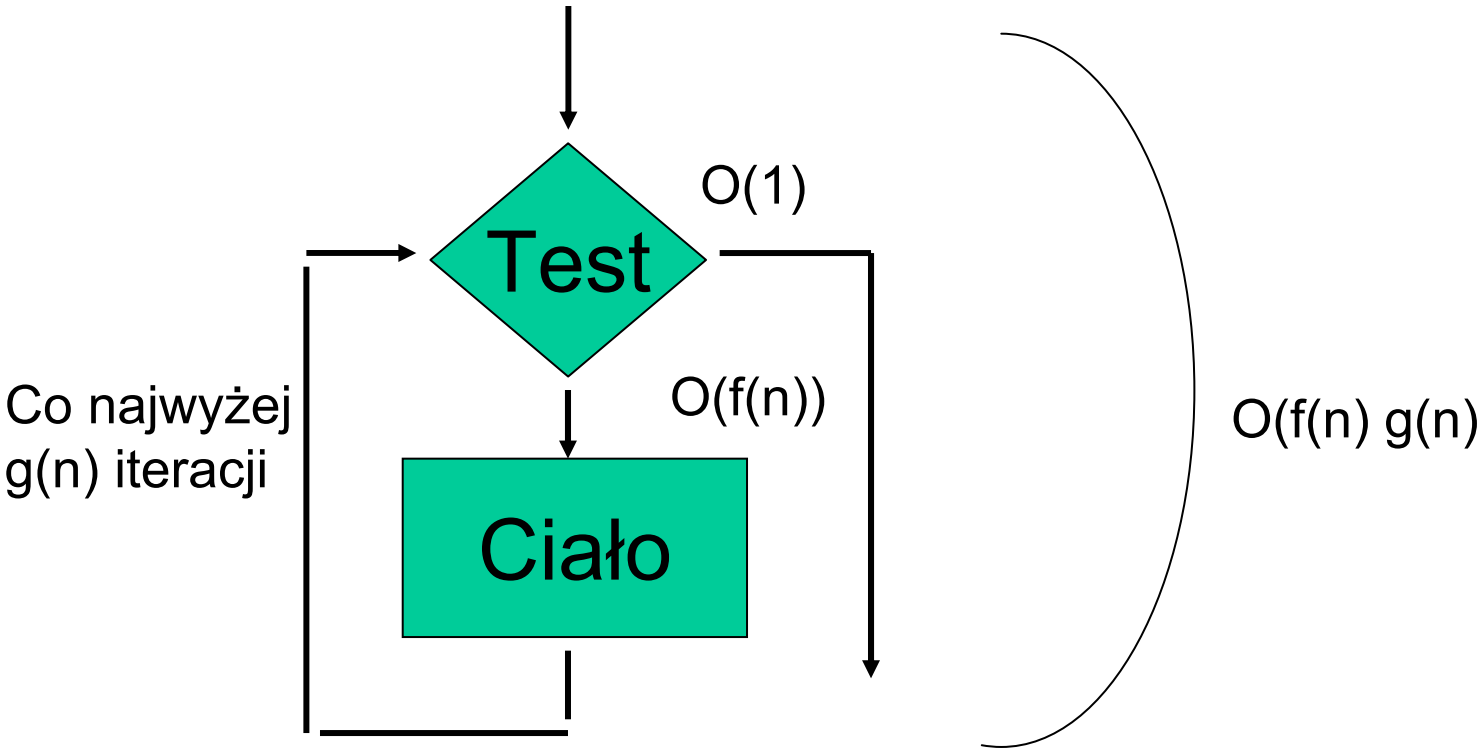
(3) **<blok-else>** wykonywany jest tylko w przypadku gdy warunek jest fałszywy, czas działania **g(n)**

Czas działania instrukcji warunkowej należy zapisać jako **O(max (f(n), g(n) )**

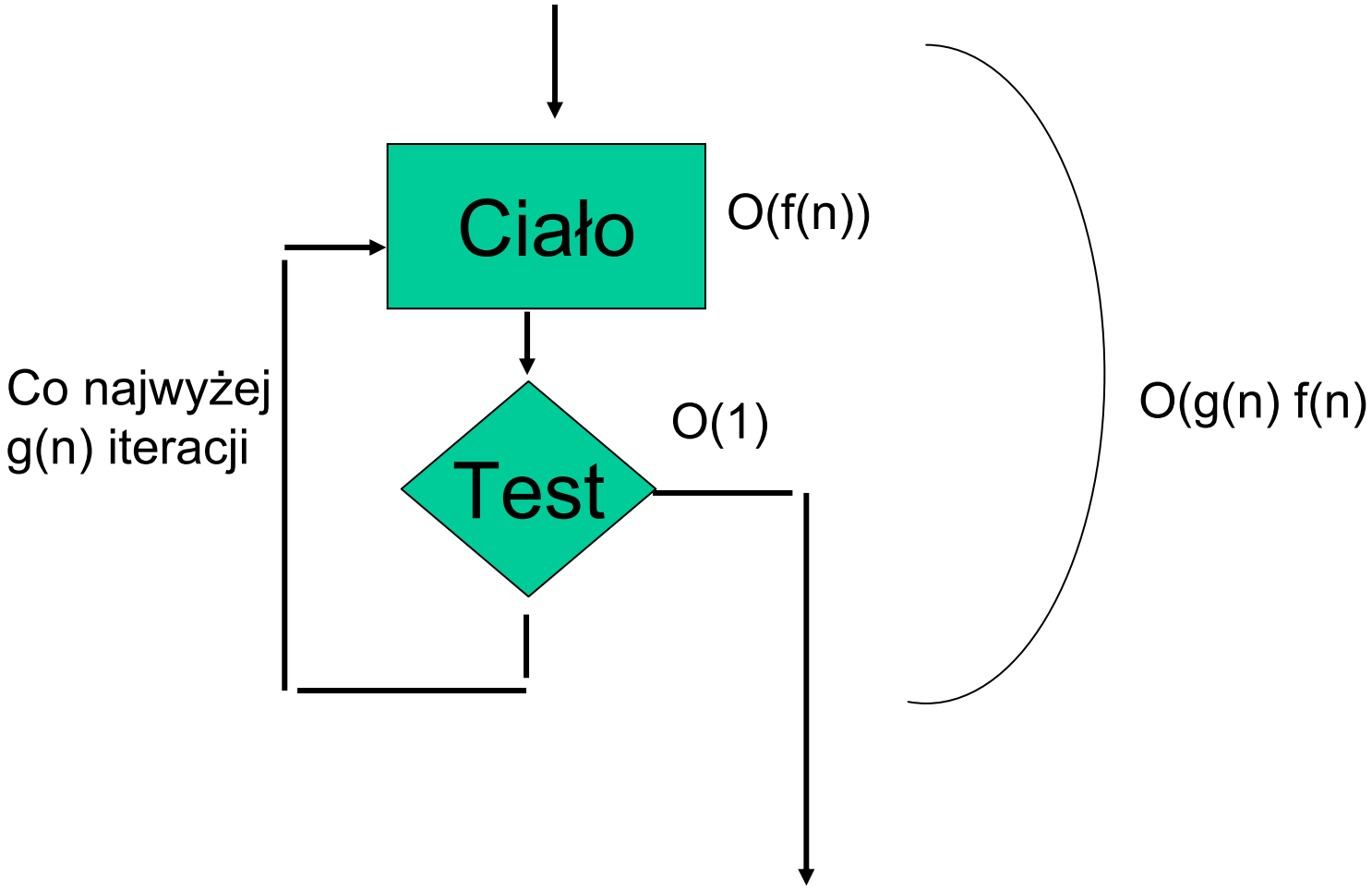
# Czas działania instrukcji „if"



# Czas działania instrukcji „while"



# Czas działania instrukcji „do-while"

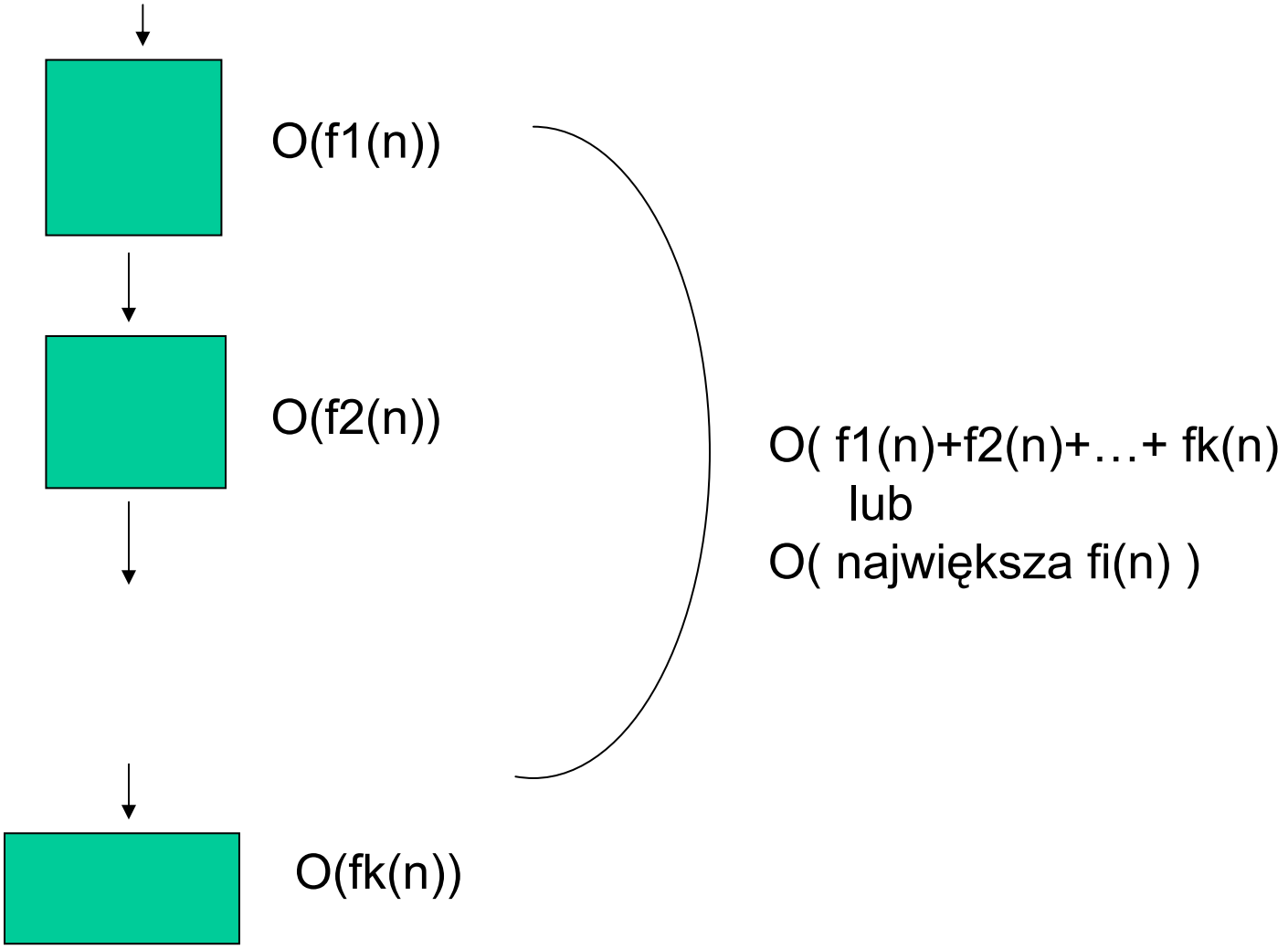


# Czas działania bloków

Sekwencja instrukcji przypisań, odczytów i zapisów, z których każda wymaga czasu O(1), potrzebuje do swojego wykonania łacznego czasu O(1).
Pojawiają się również instrukcje złożone, jak instrukcje warunkowe i pętle.
Sekwencję prostych i złożonych instrukcji nazywa się blokiem.

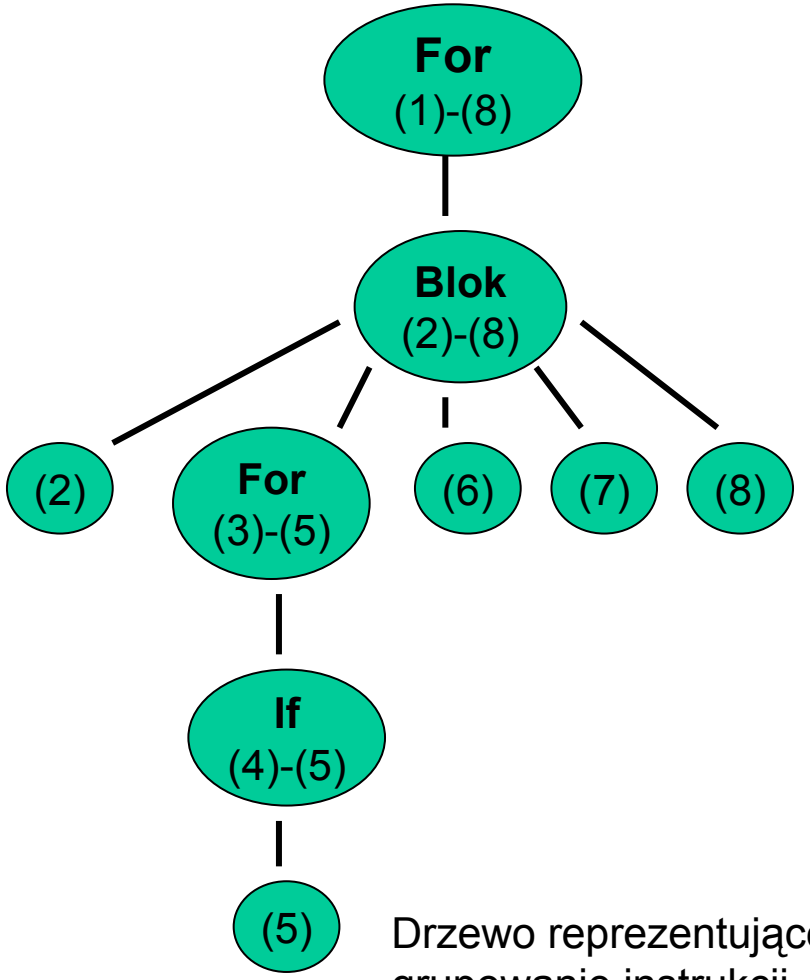
Czas działania bloku obliczmy sumując górne ograniczenia czasów wykonania poszczególnych instrukcji, które należą do tego bloku.

# Czas działania bloku instrukcji



# Przykład: „sortowanie przez wybieranie"

```
(1)  For (i=0; i< n-1; i++ ) {
(2)    small = i;
(3)    for (j=i+1; j<n; j++ )
(4)        if( A[j] < A[small] )
(5)        small =j;
(6)    temp = A[small];
(7)    A[small] A[i];
(8)    A[i] = temp;
       }
```



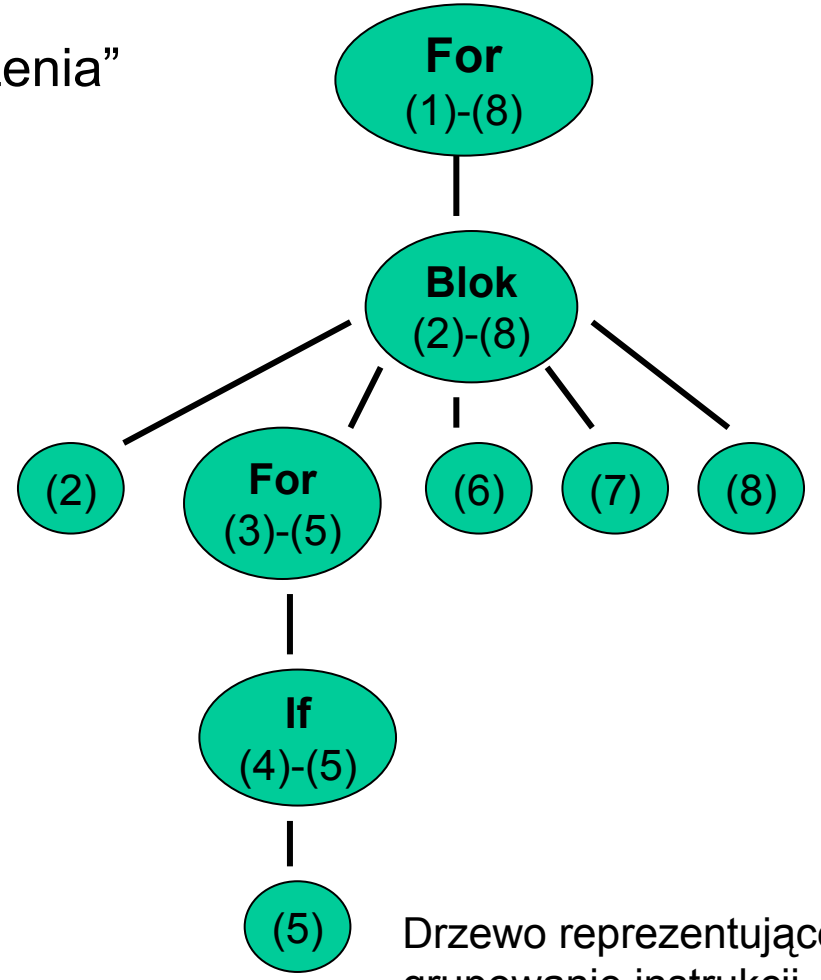

Drzewo reprezentujące grupowanie instrukcji

# Przykład: „sortowanie przez wybieranie"

Rozpoczynamy analizę „od liścia do korzenia"

- Każda instrukcja przypisania (liście) wymaga czasu O(1)
- Instrukcja „if" (4-5) wymaga czasu O(1)
- Instrukcja 'for" (3)-(5) wymaga czasu O(n-i-1) oraz i<n
- Instrukcja „for" (2)-(8) może być dalej ograniczona przez O(n-1)
- Instrukcja „for" (1)-(8) może być ograniczona przez O(n (n-1) ),

Odrzucając wyraz mniej znaczący otrzymujemy oszacowanie czasu działania jako **$O(n^2)$**




Drzewo reprezentujące grupowanie instrukcji

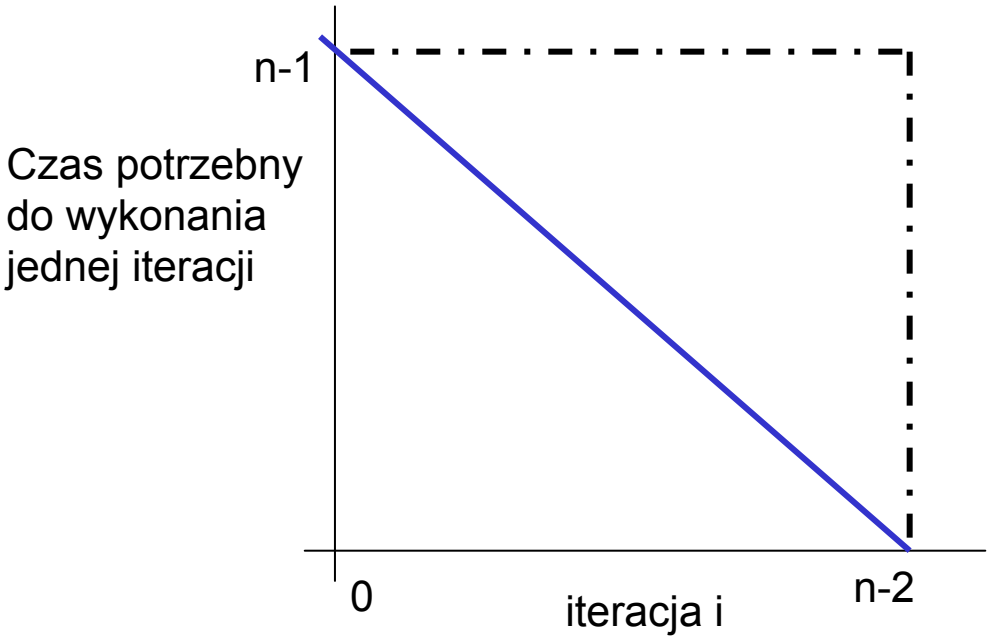
# Proste lub precyzyjne ograniczenie

Dotychczas rozważaliśmy szacowanie czasu działania pętli używając ujednoliconego górnego ograniczenia, mającego zastosowanie w każdej iteracji pętli. Dla sortowania przez wybieranie, takie proste ograniczenie prowadziło do szacowania czasu wykonania $O(n^2)$.
Można jednak dokonać bardziej uważnej analizy pętli i traktować wszystkie jej iteracje osobno. Można wówczas dokonać sumowania górnych ograniczeń poszczególnych iteracji.
Czas działania pętli z wartością *i* zmiennej indeksowej *i* wynosi $O(n-i-1)$, gdzie *i* przyjmuje wartości od *0* do *n-2*.

*Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi*

$$O\left(\sum_{n=0}^{n-2}(n-i-1)\right) = O(n(n-1)/2)$$

# Proste lub precyzyjne ograniczenie




*Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi*

$$O\left(\sum_{n=0}^{n-2}(n-i-1)\right) = O(n(n-1)/2)$$

## *Nie przejmuj się efektywnością algorytmu… wystarczy poczekać kilka lat.*

*Taki pogląd funkcjonuje w środowisku programistów, nie określono przecież granicy rozwoju mocy obliczeniowych komputerów. Nie należy się jednak z nim zgadzać w ogólności. Należy zdecydowanie przeciwstawiać się przekonaniu o tym, że ulepszenia sprzętowe uczynią pracę nad efektywnymi algorytmami zbyteczną.*

*Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych jest teoretycznie możliwe, ale praktycznie przekracza możliwości istniejących technologii. Przykładem takiego problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy "inteligentna" komunikacja pomiędzy komputerami a ludźmi na rozmaitych poziomach.*
*Kiedy pewne problemy stają się "proste"… Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrażać, wytyczy nowe granice możliwości wykorzystania komputerów.*

- Do problemu systematycznej analizy czasu działania programu powrócimy jeszcze na wykładzie w styczniu …