

# TEORETYCZNE PODSTAWY INFORMATYKI

22/10/2018

WFAiS UJ, Informatyka Stosowana  
I rok studiów, I stopień

# Wykład 3

2

Złożoność  
obliczeniowa  
algorytmów

- **Notacja „wielkie O”**
- **Notacja  $\Omega$  i  $\Theta$**
- **Algorytm Hornera**
- **Przykłady rzędów złożoności**
- **Klasy złożoności algorytmów**
- **Funkcje niewspółmierne**
- **Analiza czasu działania algorytmu**
  - ▣ **Instrukcje proste; instrukcje warunkowe; bloki instrukcji**
- **Efektywność algorytmu**

# Złożoność obliczeniowa

3

- **Złożoność obliczeniowa:**
  - **Jest to miara służąca do porównywania efektywności algorytmów.**
  - **Mamy dwa kryteria efektywności:**
    - **Czas,**
    - **Pamięć**
- **Do oceny efektywności stosujemy jednostki logiczne wyrażające związek między rozmiarem danych  $N$  (wielkość pliku lub tablicy) a ilością czasu  $T$  potrzebną na ich przetworzenie.**

# Złożoność asymptotyczna

4

- Funkcja wyrażająca zależność między  $N$  a  $T$  jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych
- Przybliżona miara efektywności to tzw. **złożoność asymptotyczna**.

# Które człony są ważne?

5

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

<b>n</b>	<b>f(n)</b>	<b><math>n^2</math></b>	<b><math>100 \cdot n</math></b>	<b><math>\log_{10} n</math></b>	<b>1000</b>
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%

- $n$  – rozmiar danych,
- $f(n)$  – ilość wykonywanych operacji

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

# Notacja „wielkie O”

6

## Definicja:

$f(n)$  jest  $O(g(n))$ , jeśli istnieją liczby dodatnie  $c$  i  $n_0$  takie że:  
 $f(n) < c \cdot g(n)$  dla wszystkich  $n \geq n_0$ .

## Przykład:

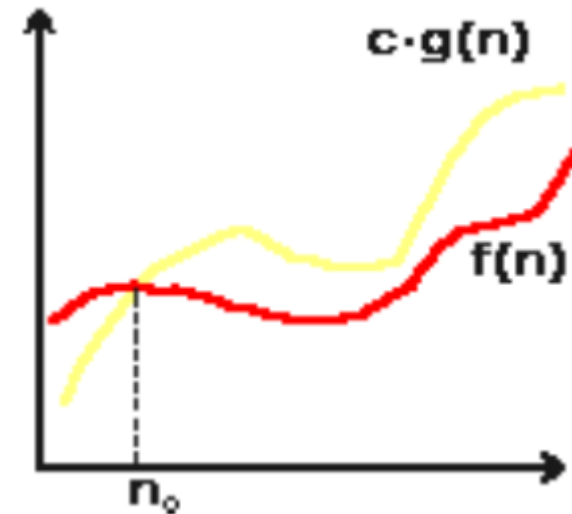
□  $f(n) = n^2 + 100n + \log_{10} n + 1000$

możemy przybliżyć jako:

$$f(n) \approx n^2 + 100n + O(\log_{10} n)$$

albo jako:

$$f(n) \approx O(n^2)$$



- Notacja „wielkie O” ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów.

# Własności notacji „wielkie O”

7

- **Własność 1** (przechodność):
  - 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  $O(n^{k+i})$  dla dowolnego dodatniego  $i$

# Własności notacji „wielkie O”

8

- Z tych wszystkich własności wynika, że dowolny wielomian jest „wielkie O” dla  $n$  podniesionego do najwyższej w nim potęgi, czyli :

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \text{ jest } O(n^k)$$

(jest też oczywiście  $O(n^{k+i})$  dla dowolnego dodatniego  $i$ )



# Własności notacji „wielkie O”

9

## □ 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  $a$

# Własności notacji „wielkie O”

10

- 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.

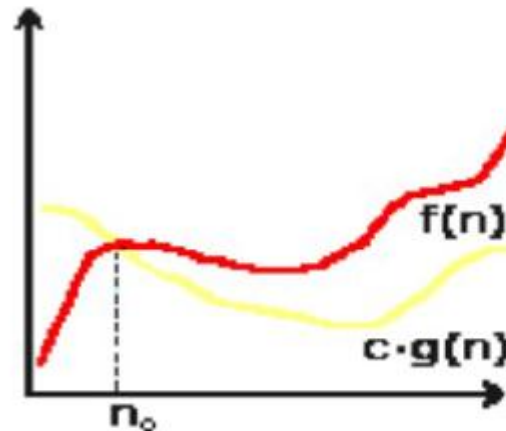
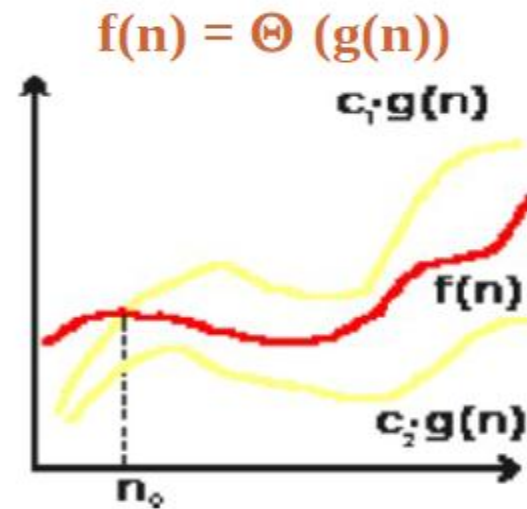
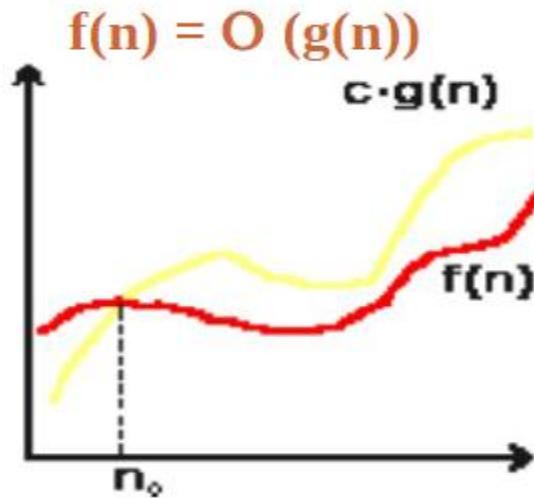
# Notacja $\Omega$ i $\Theta$

11

- 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  $n_0$  takie że,  $f(n) \geq c g(n)$  dla wszystkich  $n \geq n_0$ .
- **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  $n_0$  takie że,  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  dla wszystkich  $n \geq n_0$ .

# Notacja $O$ , $\Omega$ i $\Theta$

12



$f(n) = \Omega(g(n))$

# Algorytm Hornera

13

- Załóżmy, że mamy policzyć wartość wielomianu postaci:

$$f(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$$

dla danych liczb  $a_0, a_1, \dots, a_n$ , w danym punkcie  $x_0$ .

- Algorytm polegający na **bezpośrednim liczeniu** ze wzoru wymaga  **$n$  dodawań** i  **$(n-2)$  potęgowań** lub  **$(2n-1)$  mnożeń** co w wyniku daje niedokładności (błąd względny i bezwzględny).
  - ▣ Warto poszukać innego rozwiązania.

# Algorytm Hornera

14

Przedstawiamy wielomian w postaci:

$$f(x) = (\dots((a_0x + a_1)x + a_2)x + \dots + a_{n-1})x + a_n$$

to otrzymujemy następującą metodę na obliczanie wielomianu:

$$\begin{aligned}b_0 &= a_0 \\b_1 &= b_0x_0 + a_1 \\b_2 &= b_1x_0 + a_2\end{aligned}$$

$$b_n = b_{n-1}x_0 + a_n$$

gdzie  $b_i$  oznacza wartość  $i$ -tego nawiasu dla  $x$  równego  $x_0$ , a  $b_n$  szukaną wartość wielomianu.

Algorytm wymaga  **$n$  dodawań** i  **$n$  mnożeń**.

# Algorytm Hornera

15

- Otrzymana metoda to tzw. **Algorytm Hornera** obliczania wartości wielomianu.
- Algorytm ten jest numerycznie poprawny i jest jedynym algorytmem który minimalizuje liczbę dodawań i mnożeń przy obliczaniu wartości wielomianu wg. podanej postaci.

# O czym należy pamiętać

16

- 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 pochośnie.



# Pamiętaj o dużych stałych

17

## 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 notację „**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

18

- **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 zastanawiając się nad doбором struktury danych.**

# Najczęstsze złożoności

19

- **$\log(n)$**  – złożoność logarytmiczna
- **$n$**  – złożoność liniowa
- **$n \log(n)$**  – złożoność liniowo-logarytmiczna
- **$n^k$**  – złożoność wielomianowa
- **$2^n$**  – złożoność wykładnicza
- **$n!$**  – złożoność wykładnicza ponieważ  $n! > 2^n$  już dla  $n=4$ .

# Klasy złożoności algorytmów

20

- **Czasy wykonania logarytmów na komputerze działającym z szybkością 1 instrukcja /  $\mu\text{s}$ .**

klasa	złożoność	liczba operacji i czas wykonania			
		n	10	10 <sup>3</sup>	
<b>stały</b>	$O(1)$	1	1 $\mu\text{s}$	1	1 $\mu\text{s}$
<b>logarytmiczny</b>	$O(\log n)$	3.32	3 $\mu\text{s}$	9.97	10 $\mu\text{s}$
<b>liniowy</b>	$O(n)$	10	10 $\mu\text{s}$	10 <sup>3</sup>	1ms
<b>kwadratowy</b>	$O(n^2)$	10 <sup>2</sup>	100 $\mu\text{s}$	10 <sup>6</sup>	1s
<b>wykładniczy</b>	$O(2^n)$	1024	10ms	10 <sup>301</sup>	>> 10 <sup>16</sup> lat

# Funkcje niewspółmierne

21

- **Bardzo wygodna jest możliwość porównywania dowolnych funkcji  $f(n)$  i  $g(n)$  za pomocą notacji „wielkie 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 „wielkim O” dla drugiej.**

# Funkcje niewspółmierne

22

## □ Przykład:

Rozważmy funkcję  $f(n)=n$  dla nieparzystych  $n$  oraz  $f(n)=n^2$  dla parzystych  $n$ .

- Oznacza to, że  $f(1)=1$ ,  $f(2)=4$ ,  $f(3)=3$ ,  $f(4)=16$ ,  $f(5)=5$  itd...
  - Podobnie, niech  $g(n)=n^2$  dla nieparzystych  $n$  oraz  $g(n)=n$  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

23

- Mając do dyspozycji definicję „**wielkie 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

24

- Przyjmujemy zasadę że czas działania pewnych prosty operacji na danych wynosi  $O(1)$ , czyli jest niezależny od rozmiaru danych wejściowych.
  - ▣ Operacje arytmetyczne, np. (+), (-)
  - ▣ Operacje logiczne (&&)
  - ▣ Operacje porównania (<=)
  - ▣ Operacje dostępu do struktur danych, np. indeksowanie tablic (A[i])
  - ▣ Proste przypisania, np. kopiowanie wartości do zmiennej.
  - ▣ Wywołania funkcji bibliotecznych, np. `scanf` lub `printf`
- Każdą z tych operacji można wykonać za pomocą pewnej (niewielkiej) liczby rozkazów maszynowych.



# Czas działania pętli „for”

25

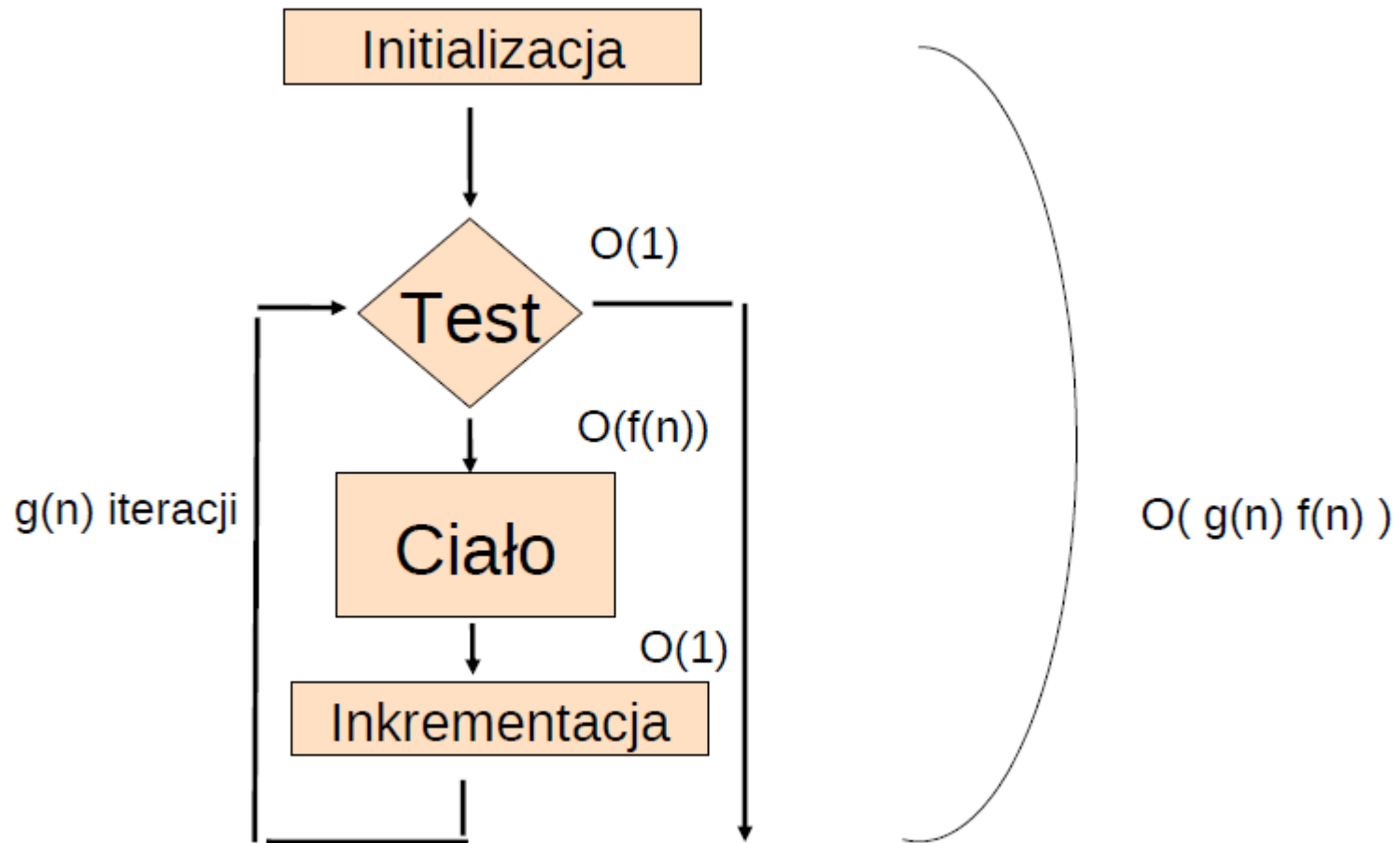
- **Przykład 1: Prosta pętla**

```
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 pętli „for”

26



# Czas działania pętli „for”

27

## □ Przykład 2: Pętla zagnieżdżona

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

- **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  $i \in \{1, \dots, n-1\}$ , a na każdą iterację przypadają dwa przypisania: jedno dla „sum”, jedno dla „j”.
- **Mamy zatem:  $1+3n+2(1+2+\dots+n-1) = 1+3n+n(n-1) = O(n)+O(n^2) = O(n^2)$**  przypisań wykonywanych w całym programie.
- **Złożoność asymptotyczna algorytmu jest  $O(n^2)$** . Pętle zagnieżdżone mają zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.

# Czas działania pętli „for”

28

- **Przykład 3:** Znajdź najdłuższą podtablicę 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 **1-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  $i \in \{1, \dots, n-1\}$ .

**Złożoność asymptotyczna algorytmu jest więc  $O(n^2)$ .**

# Czas działania pętli „for”

29

- Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą ale bardzo istotną.
- Staramy się wyznaczyć złożoność
  - w „przypadku optymistycznym”,
  - w „przypadku pesymistycznym”
  - oraz w „przypadku średnim”
- Często posługujemy się przybliżeniami opartymi o notacje „wielkie  $O$ ,  $\Omega$  i  $\Theta$ ” .

# Czas działania instrukcji warunkowych

30

Instrukcje warunkową **if-else** zapisuje się w postaci:

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

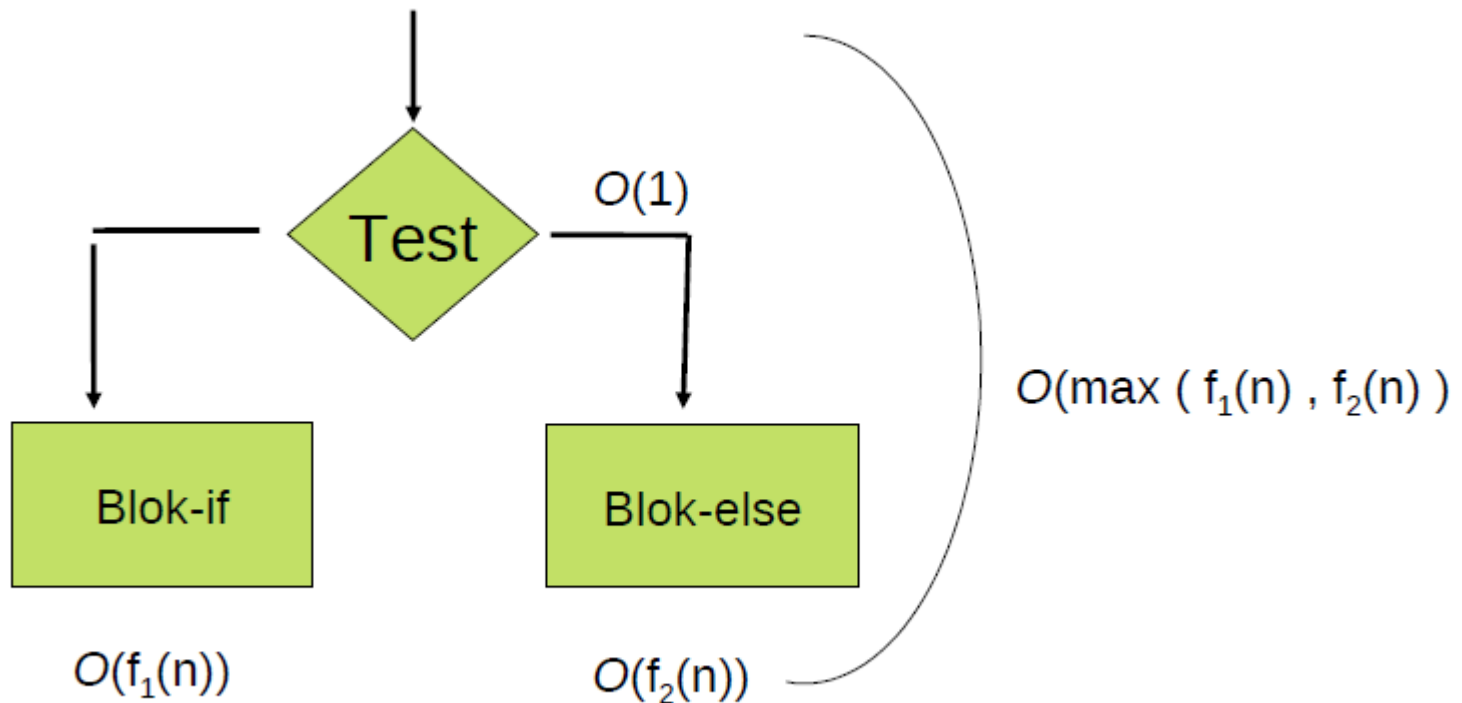
Gdzie

- **<warunek>** jest wyrażeniem które trzeba obliczyć. Warunek niezależnie od tego jak skomplikowany wymaga wykonania stałej liczby operacji (więc czasu  $O(1)$ ) chyba że zawiera wywołanie funkcji, .
- **<blok-if>** zawiera instrukcje wykonywane tylko w przypadku gdy warunek jest prawdziwy, czas działania  $f(n)$ .
- **<blok-else>** wykonywany jest tylko w przypadku gdy warunek jest fałszywy, czas działania  $g(n)$ .

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

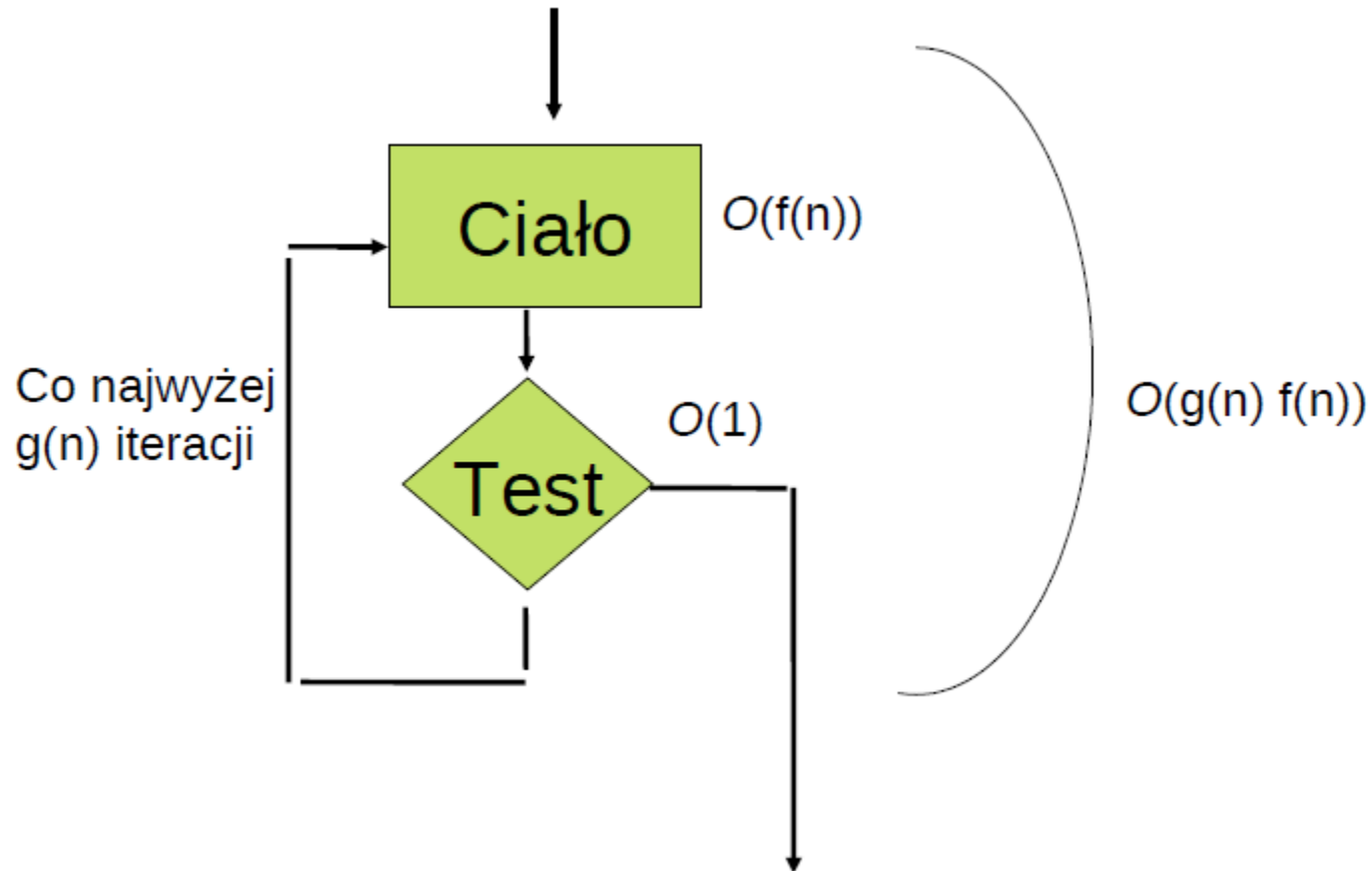
# Czas działania instrukcji „if”

31



# Czas działania instrukcji „while”

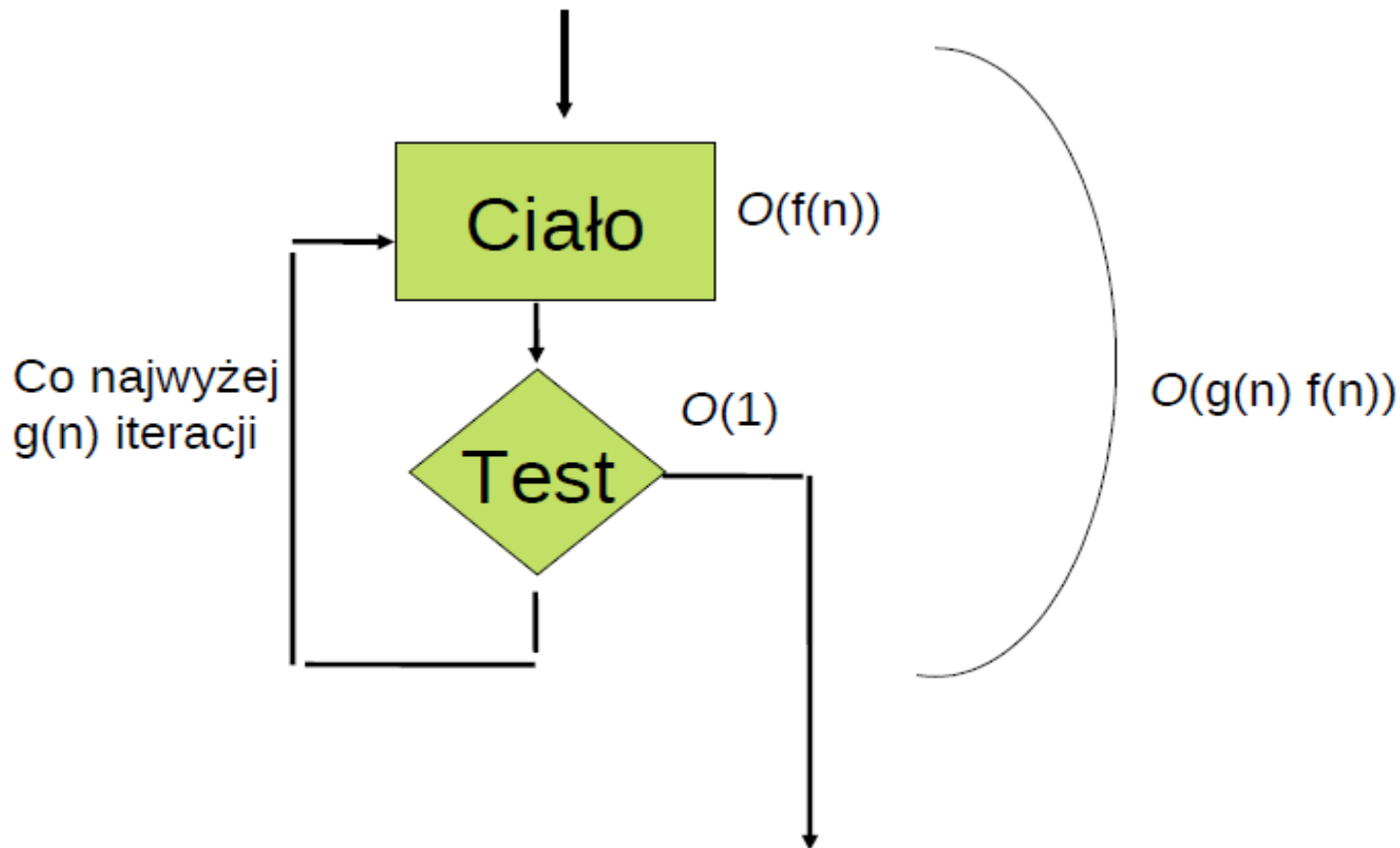
32





# Czas działania instrukcji „do while”

33



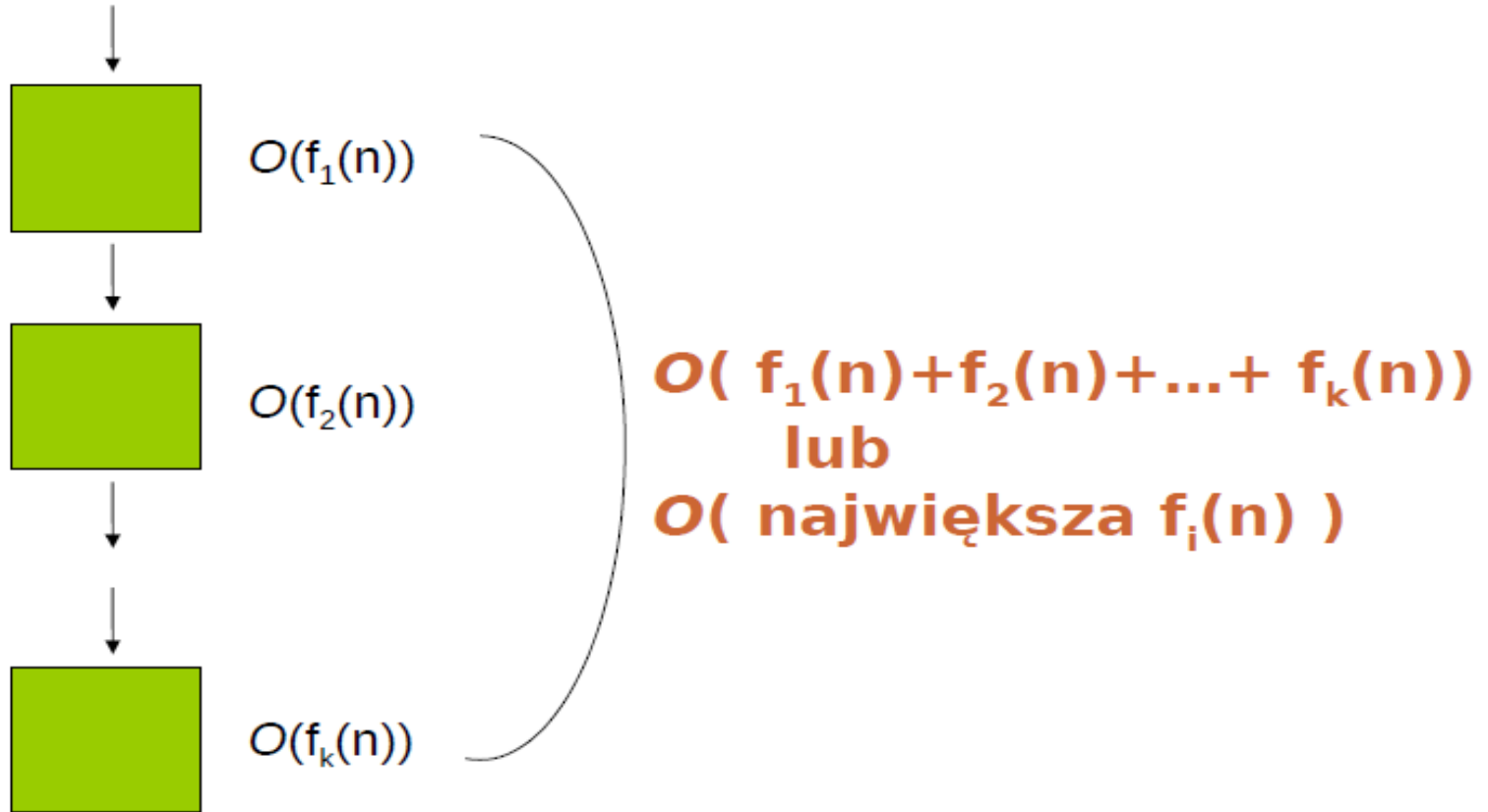
# Czas działania bloków

34

- Sekwencja instrukcji przypisań, odczytów i zapisów, z których każda wymaga czasu  $O(1)$ , potrzebuje do swojego wykonania łącznego czasu  $O(1)$ .
- Pojawiają się również instrukcje złożone, jak instrukcje warunkowe i pętle.
  - ▣ **Sekwencję** prostych i złożonych **instrukcji** nazywa się **blokiem**.
- Czas działania bloku obliczymy sumując górne ograniczenia czasów wykonania poszczególnych instrukcji, które należą do tego bloku.

# Czas działania bloku instrukcji

35

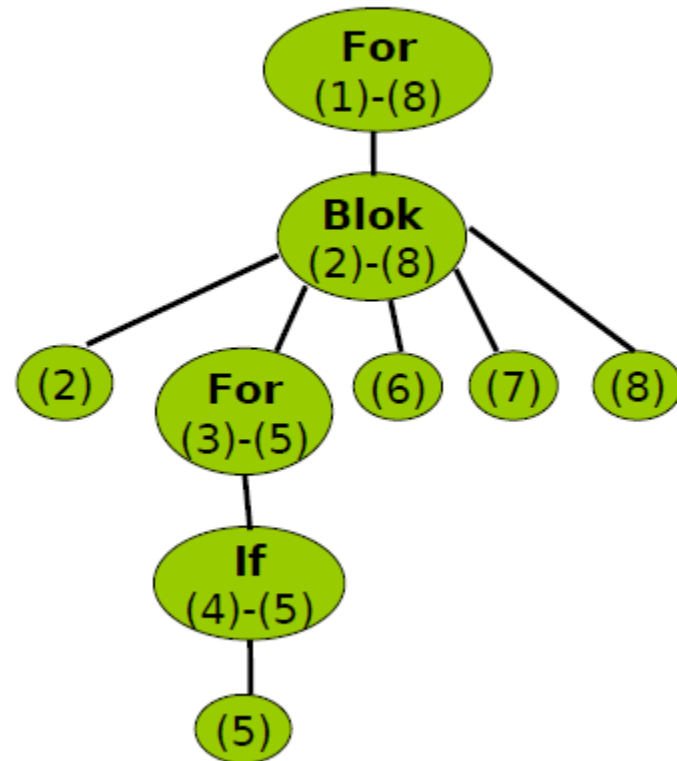


# Przykład: „sortowanie przez wybieranie”

36

## Drzewo reprezentujące grupowanie instrukcji

```
(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; }
```



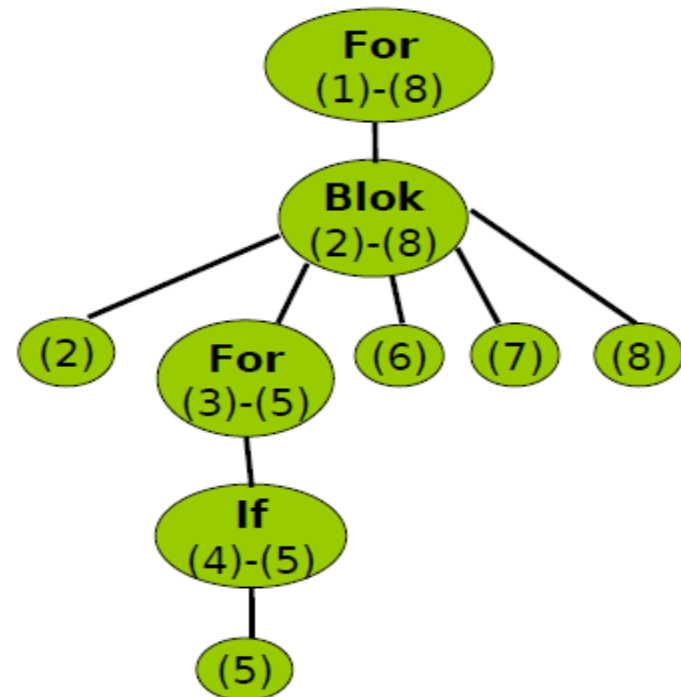
# Przykład: „sortowanie przez wybieranie”

37

- 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



# Przybliżone lub precyzyjne ograniczenie

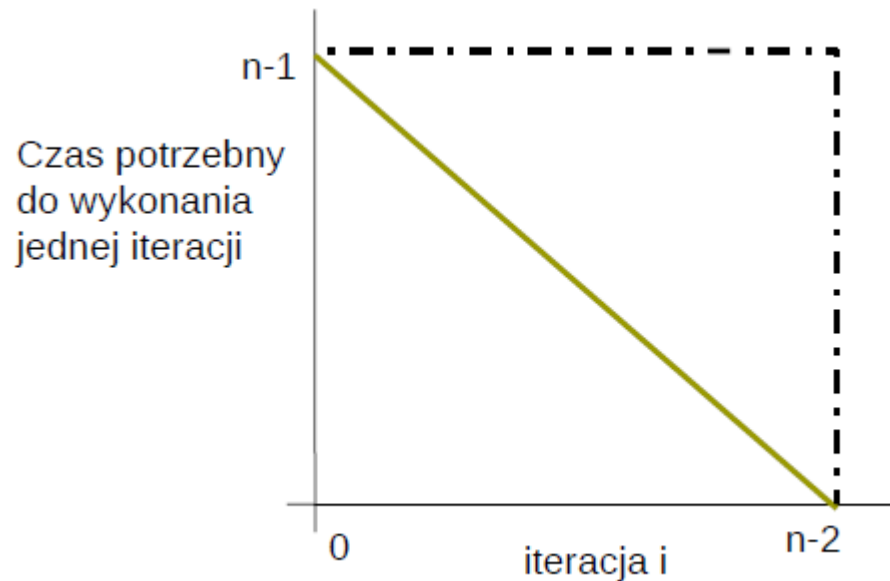
38

- Dotychczas rozważaliśmy szacowanie czasu działania pętli używając ujednoliczonego górnego ograniczenia, mającego zastosowanie w każdej iteracji pętli.
- Dla sortowania przez wybieranie, takie przybliżone ograniczenie prowadziło do szacowania czasu wykonania pętli  $O(n^2)$ .
- Można jednak dokonać bardziej szczegółowej analizy pętli i dokonać sumowania górnych ograniczeń poszczególnych iteracji. Część 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 niezbędne do wykonania wszystkich iteracji wynosi:

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

# Przybliżone lub precyzyjne ograniczenie

39



- Górne ograniczenie czasu niezbędne do wykonania wszystkich iteracji wynosi:

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

# Efektywność algorytmu

40

## □ Czas działania:

- Oznaczamy przez funkcję  $T(n)$  liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze  $n$ .
- Funkcje te nazywamy **czasem działania**. Dość często czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru. W takim przypadku, funkcję  $T(n)$  definiuje się jako **najmniej korzystny przypadek** z punktu widzenia kosztów czasowych. Inną wyznaczaną wielkością jest też **czas średni**, czyli średni dla różnych danych wejściowych.



# Nie przejmuj się efektywnością algorytmu...

41

- **Nie przejmuj się efektywnością algorytmu... wystarczy poczekać kilka lat.**
- Taki pogląd funkcjonuje czasem 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.

# Nie przejmuj się efektywnością algorytmu...

42

- 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 wyobrazić, **wytyczy nowe granice możliwości wykorzystania komputerów**.

# Uwagi końcowe

43

- Na wybór najlepszego algorytmu dla tworzonego programu wpływa wiele czynników, najważniejsze to:
  - **prostota,**
  - **łatwość implementacji**
  - **efektywność**
- Do problemu systematycznej analizy czasu działania programu powrócimy jeszcze na wykładzie za kilka tygodni...

# Pytania do wykładu

44

1. **Podaj definicję notacji: „wielkie O”,  $\Omega$ ,  $\Theta$ .**
2. **Podaj własności notacji „wielkie O”.**
3. **Wymień znane Ci klasy złożoności algorytmu.**
4. **Co to są funkcje niewspółmierne? Podaj przykład.**
5. **Co to jest złożoność obliczeniowa średnia?  
Uzasadnij pojęcie dla dowolnie wybranego algorytmu.**
6. **Co to jest złożoność obliczeniowa asymptotyczna?  
Uzasadnij pojęcie dla dowolnie wybranego algorytmu.**