Podstawy Programowania

Wykład X

Złożoność obliczeniowa

Robert Muszyński
Katedra Cybernetyki i Robotyki, PWr

Zagadnienia: efektywność programów/algorytmów, sposoby zwiększania efektywności algorytmów, zasada 80–20, ocena efektywności algorytmów, ocena złożoności obliczeniowej, rzeczywista a teoretyczna złożoność obliczeniowa, porównanie metod sortowania, efektywność w praktyce, sfera problemów algorytmicznych, problemy nieobliczalne.

– Skład FoilTEX –

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność
- przenośność

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność
- przenośność
- łatwość utrzymania

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność
- przenośność
- łatwość utrzymania
- efektywność

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność
- przenośność
- łatwość utrzymania
- efektywność
  - ⋆ szybkość działania
  - ⋆ wielkość

} kompromis

Sprawność programów (algorytmów)

Jakość programów (algorytmów):

- poprawność/niezawodność
- przenośność
- łatwość utrzymania
- efektywność
  - ⋆ szybkość działania } kompromis
  - ⋆ wielkość
    - ∗ objętość kodu
    - ∗ wielkość struktur danych

Sposoby zwiększania efektywności programów

- na etapie planowania programu (np. dobór algorytmu)

Sposoby zwiększania efektywności programów

- na etapie planowania programu (np. dobór algorytmu)
- na etapie uruchamiania programu (np. wpisanie procedury w miejsce wywołania)

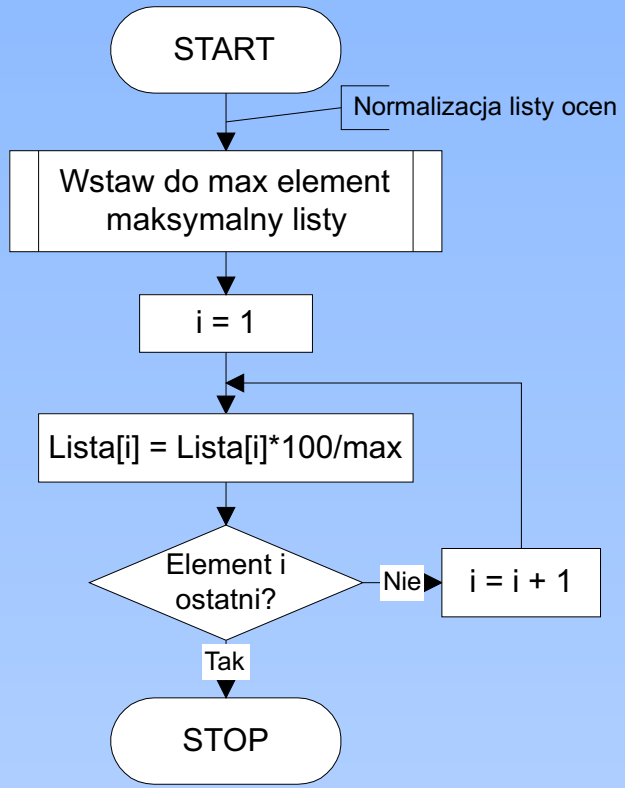
Sposoby zwiększania efektywności programów

- na etapie planowania programu (np. dobór algorytmu)
- na etapie uruchamiania programu (np. wpisanie procedury w miejsce wywołania)

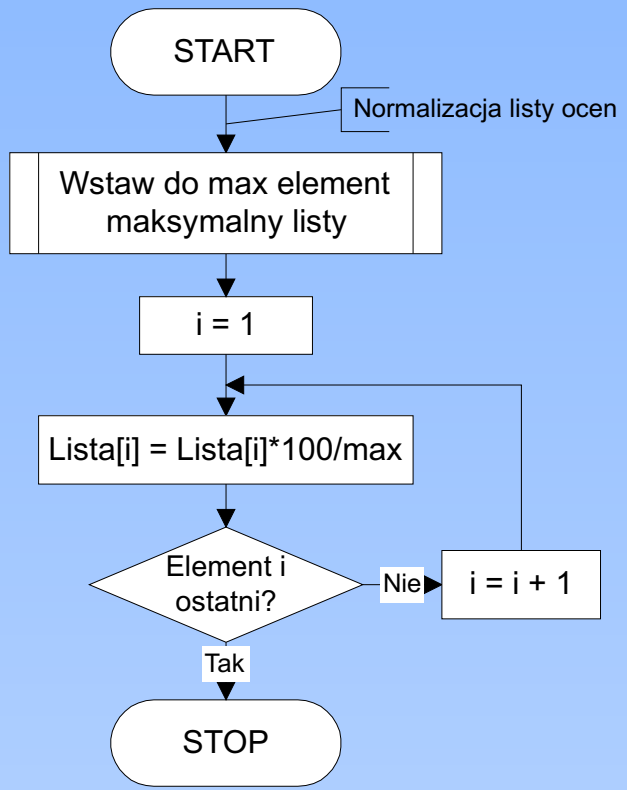
Zasada 80–20

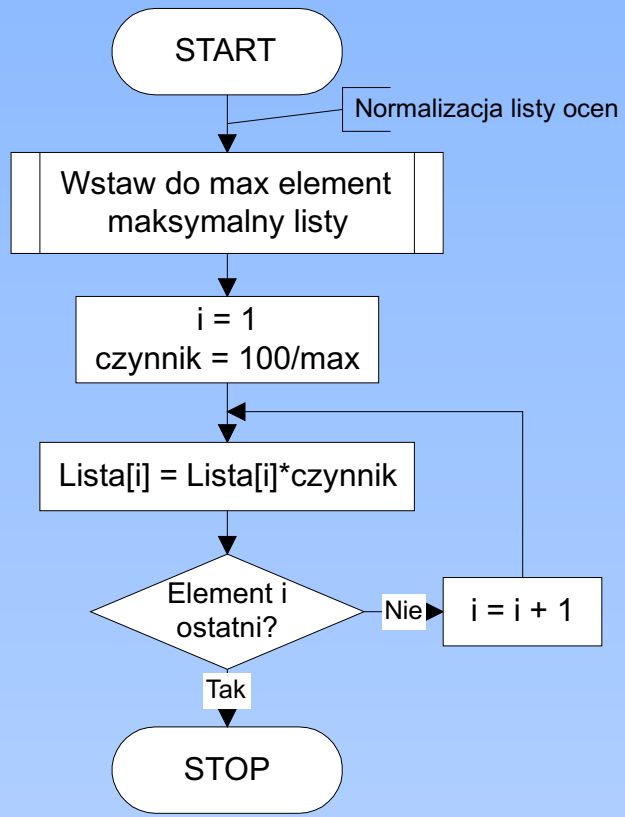
Program spędza 80% czasu w 20% swojego kodu.

Poprawa efektywności — przykład

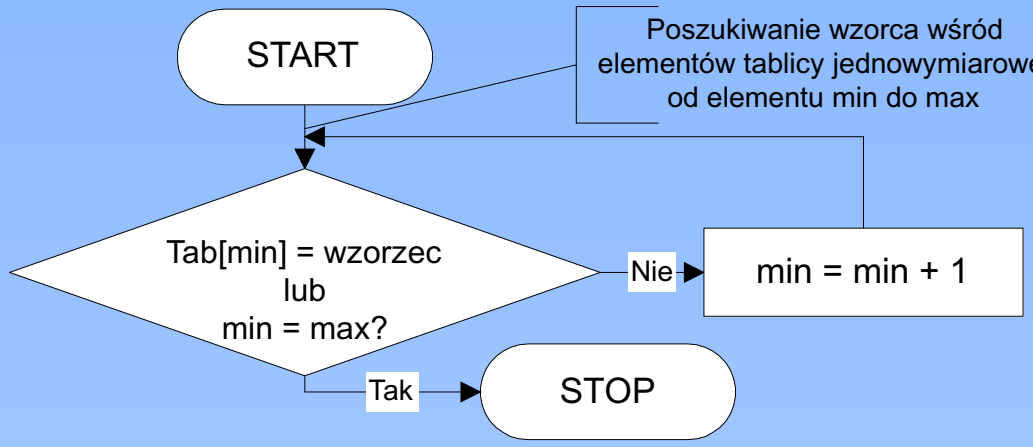


Poprawa efektywności — przykład

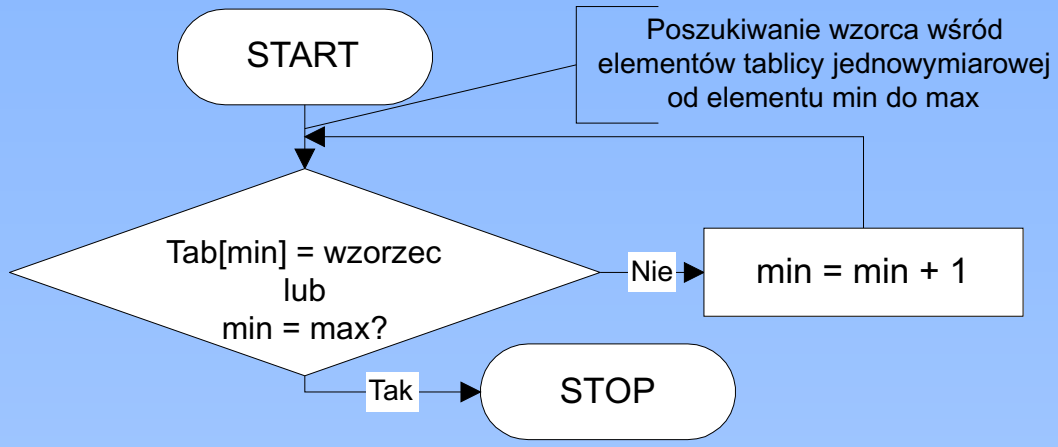




Poprawa efektywności — inny przykład



Poprawa efektywności — inny przykład



Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa
- Przypadki: najgorszy, najlepszy, średni

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa
- Przypadki: najgorszy, najlepszy, średni
- Rodzaj komputera } niezależna

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa
- Przypadki: najgorszy, najlepszy, średni
- Rodzaj komputera
- Język programowania

} niezależna

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa
- Przypadki: najgorszy, najlepszy, średni
- Rodzaj komputera
- Język programowania
- Rodzaj kompilatora

} niezależna

Ocena efektywności programów (algorytmów)

- Funkcja wielkości zbioru danych wejściowych
- Wyrażona liczbą elementarnych operacji/jednostek pamięci — złożoność obliczeniowa (czasowa)/pamięciowa
- Przypadki: najgorszy, najlepszy, średni
- Rodzaj komputera
- Język programowania
- Rodzaj kompilatora

} niezależna

- Zbiór danych wejściowych

Rodzaj komputera

Przykładowe czasy sortowania

8170 liczb

Typ komputera	Czas
komputer osobisty	51.915s

Rodzaj komputera

Przykładowe czasy sortowania

8170 liczb

Typ komputera	Czas
komputer osobisty	51.915s
stacja robocza	11.508s

Rodzaj komputera

Przykładowe czasy sortowania

8170 liczb

Typ komputera	Czas
komputer osobisty	51.915s
stacja robocza	11.508s
minikomputer	2.382s
mainframe	0.431s
superkomputer	0.087s

Rodzaj komputera

Przykładowe czasy sortowania

8170 liczb

Typ komputera	Czas
komputer osobisty	51.915s
stacja robocza	11.508s
minikomputer	2.382s
mainframe	0.431s
superkomputer	0.087s

n liczb

n	k. osobisty	s. robocza
125	12.5ms	2.8ms
250	49.3ms	11.0ms
500	195.8ms	43.4ms
1000	780.3ms	172.9ms
2000	3114.9ms	690.5ms

Rodzaj komputera

Przykładowe czasy sortowania

8170 liczb

Typ komputera	Czas
komputer osobisty	51.915s
stacja robocza	11.508s
minikomputer	2.382s
mainframe	0.431s
superkomputer	0.087s

n liczb

n	k. osobisty	s. robocza
125	12.5ms	2.8ms
250	49.3ms	11.0ms
500	195.8ms	43.4ms
1000	780.3ms	172.9ms
2000	3114.9ms	690.5ms

↑ ↑

parabole

Zbiór danych wejściowych

Działanie algorytmów przeszukiwania dla n elementów

		liniowo			binarnie		
	n=	7	13	14	7	13	14

Zbiór danych wejściowych

Działanie algorytmów przeszukiwania dla n elementów

element		liniowo			binarnie		
szukany	n=	7	13	14	7	13	14
1-szy		1	1	1	3	3	3

Zbiór danych wejściowych

Działanie algorytmów przeszukiwania dla n elementów

element szukany	n=	liniowo			binarnie		
		7	13	14	7	13	14
1-szy		1	1	1	3	3	3
n-ty		7	13	14	3	4	4

Zbiór danych wejściowych

Działanie algorytmów przeszukiwania dla n elementów

element szukany	n=	liniowo 7	liniowo 13	liniowo 14	binarnie 7	binarnie 13	binarnie 14
1-szy		1	1	1	3	3	3
n-ty		7	13	14	3	4	4
[n/2]+1		4	7	8	1	1	4
[n/2]		3	6	7	3	4	1

Zbiór danych wejściowych

Działanie algorytmów przeszukiwania dla n elementów

element szukany	n=	liniowo			binarnie		
		7	13	14	7	13	14
1-szy		1	1	1	3	3	3
n-ty		7	13	14	3	4	4
[n/2]+1		4	7	8	1	1	4
[n/2]		3	6	7	3	4	1

Przypadki: najgorszy, najlepszy, średni

Zbiór danych wejściowych

Jak poprzednio, inne kryterium układu tabeli

przypadek	liniowo			binarnie		
n=	7	13	n	7	13	n
najgorszy	7	13	n	3	4	$[\log_2 n+1]$

Zbiór danych wejściowych

Jak poprzednio, inne kryterium układu tabeli

przypadek	liniowo			binarnie		
n=	7	13	n	7	13	n
najgorszy	7	13	n	3	4	$[\log_2 n+1]$
najlepszy	1	1	1	1	1	1

Zbiór danych wejściowych

Jak poprzednio, inne kryterium układu tabeli

przypadek	liniowo			binarnie		
n=	7	13	n	7	13	n
najgorszy	7	13	n	3	4	$[\log_2 n+1]$
najlepszy	1	1	1	1	1	1
średni	3.5	6.5	n/2	2.8	3.7	$\log_2 n$

Zbiór danych wejściowych

Jak poprzednio, inne kryterium układu tabeli

przypadek	liniowo			binarnie		
n=	7	13	n	7	13	n
najgorszy	7	13	n	3	4	$[\log_2 n+1]$
najlepszy	1	1	1	1	1	1
średni	3.5	6.5	n/2	2.8	3.7	$\log_2 n$

Dla $n=10^9$ wartość $[\log_2 n+1]$ wynosi 30.

Porównanie metod sortowania

Liczba porównań (Po) i przesunięć (Pr) obiektów w metodach sortowania

algorytm\przypadek		najgorszy	najlepszy	średni
proste wstawianie	Po=	$(n^2-n)/2-1$	$n-1$	$(n^2+n-2)/4$
	Pr=	$(n^2+3n-4)/2$	$2(n-1)$	$(n^2-9n-10)/4$
proste wybieranie	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$n^2/4+3(n-1)$	$3(n-1)$	$n(\ln n+0.57)$
sortowanie bąbelkowe	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$3(n^2-n)/2$	0	$3(n^2-n)/4$

Porównanie metod sortowania

Liczba porównań (Po) i przesunięć (Pr) obiektów w metodach sortowania

algorytm\przypadek		najgorszy	najlepszy	średni
proste wstawianie	Po=	$(n^2-n)/2-1$	$n-1$	$(n^2+n-2)/4$
	Pr=	$(n^2+3n-4)/2$	$2(n-1)$	$(n^2-9n-10)/4$
proste wybieranie	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$n^2/4+3(n-1)$	$3(n-1)$	$n(\ln n+0.57)$
sortowanie bąbelkowe	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$3(n^2-n)/2$	0	$3(n^2-n)/4$

Wady przedstawionego sposobu porównywania algorytmów:

- niedokładne (pominięte np. sterowanie wykonywaniem pętli)

Porównanie metod sortowania

Liczba porównań (Po) i przesunięć (Pr) obiektów w metodach sortowania

algorytm\przypadek		najgorszy	najlepszy	średni
proste wstawianie	Po=	$(n^2-n)/2-1$	$n-1$	$(n^2+n-2)/4$
	Pr=	$(n^2+3n-4)/2$	$2(n-1)$	$(n^2-9n-10)/4$
proste wybieranie	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$n^2/4+3(n-1)$	$3(n-1)$	$n(\ln n+0.57)$
sortowanie bąbelkowe	Po=	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	Pr=	$3(n^2-n)/2$	0	$3(n^2-n)/4$

Wady przedstawionego sposobu porównywania algorytmów:

- niedokładne (pominięte np. sterowanie wykonywaniem pętli)
- niemożliwe do przeprowadzenia dla wielu z algorytmów

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu.

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu. Stąd klasy złożoności obliczeniowej algorytmów:
  - logarytmiczne — $O(\log_2 n)$

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu. Stąd klasy złożoności obliczeniowej algorytmów:
  - logarytmiczne — $O(\log_2 n)$
  - liniowe — $O(n)$

Złożoność obliczeniowa i jej ocena

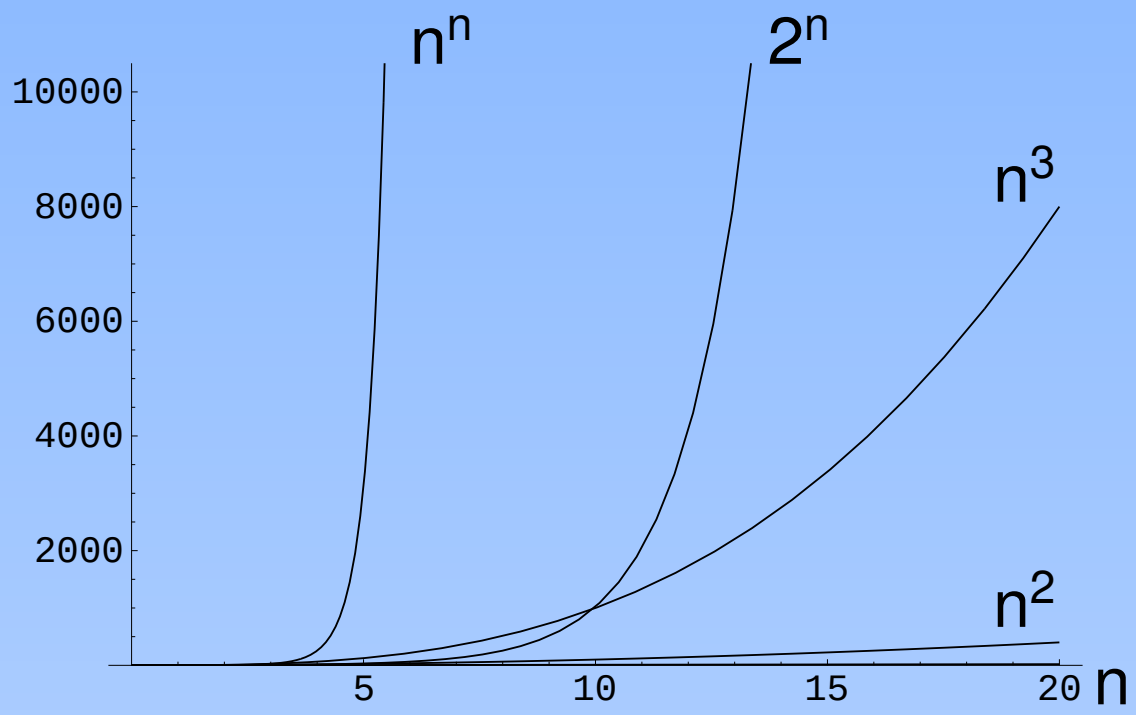
- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu. Stąd klasy złożoności obliczeniowej algorytmów:
  - ⋆ logarytmiczne — $O(\log_2 n)$
  - ⋆ liniowe — $O(n)$
  - ⋆ wielomianowe — $O(n^2)$, $O(n^3)$, $O(n^4)\ldots$

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu. Stąd klasy złożoności obliczeniowej algorytmów:
  - logarytmiczne — $O(\log_2 n)$
  - liniowe — $O(n)$
  - wielomianowe — $O(n^2)$, $O(n^3)$, $O(n^4)\ldots$
  - wykładnicze — $O(2^n)$,

Złożoność obliczeniowa i jej ocena

- proporcjonalna do liczby operacji elementarnych
- istotny składnik najszybciej rosnący ze wzrostem rozmiaru problemu. Stąd klasy złożoności obliczeniowej algorytmów:
  - logarytmiczne — $O(\log_2 n)$
  - liniowe — $O(n)$
  - wielomianowe — $O(n^2)$, $O(n^3)$, $O(n^4)\ldots$
  - wykładnicze — $O(2^n)$,
  - inne — $O(n\log_2 n)$, $O(n^{1.2})$, $O(n!)$, $O(n^n)$, $O\left(n^{n^n}\right)\ldots$



Przykładowe przebiegi funkcji.

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

liczba protonów w znanym wszechświecie ma 126 cyfr

liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

liczba protonów w znanym wszechświecie ma 126 cyfr

liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry

Zapotrzebowanie na czas dla wybranych algorytmów

(przy prędkości 1 MIPS)

n	10	20	50	100	300
$O(n^2)$	1/10000 s	1/2500 s	1/400 s	1/100 s	9/100 s

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

liczba protonów w znanym wszechświecie ma 126 cyfr

liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry

Zapotrzebowanie na czas dla wybranych algorytmów

(przy prędkości 1 MIPS)

n	10	20	50	100	300
$O(n^2)$	1/10000 s	1/2500 s	1/400 s	1/100 s	9/100 s
$O(n^5)$	1/10 s	3.2 s	5.2 min	2.8 h	28.1 dnia

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

liczba protonów w znanym wszechświecie ma 126 cyfr

liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry

Zapotrzebowanie na czas dla wybranych algorytmów

(przy prędkości 1 MIPS)

n	10	20	50	100	300
$O(n^2)$	1/10000 s	1/2500 s	1/400 s	1/100 s	9/100 s
$O(n^5)$	1/10 s	3.2 s	5.2 min	2.8 h	28.1 dnia
$O(2^n)$	1/1000 s	1 s	35.7 lat	4×10^{16} lat	10^{77} lat

Tempo wzrostu wybranych funkcji

n	10	50	100	300	1 000
$[\log_2 n]$	3	5	6	8	9
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	(7 cyfr)	(8 cyfr)	(10 cyfr)
2^n	1024	(16 cyfr)	(31 cyfr)	(91 cyfr)	(302 cyfry)
$n!$	(7 cyfr)	(65 cyfr)	(161 cyfr)	(623 cyfry)	niewyobrażalna
n^n	(11 cyfr)	(85 cyfr)	(201 cyfr)	(744 cyfry)	niewyobrażalna

liczba protonów w znanym wszechświecie ma 126 cyfr

liczba mikrosekund od „wielkiego wybuchu” ma 24 cyfry

Zapotrzebowanie na czas dla wybranych algorytmów

(przy prędkości 1 MIPS)

n	10	20	50	100	300
$O(n^2)$	1/10000 s	1/2500 s	1/400 s	1/100 s	9/100 s
$O(n^5)$	1/10 s	3.2 s	5.2 min	2.8 h	28.1 dnia
$O(2^n)$	1/1000 s	1 s	35.7 lat	4×10^{16} lat	10^{77} lat

„wielki wybuch” był w przybliżeniu 1.5×10^{10} lat temu

słońce wypali się za około 5×10^{9} lat

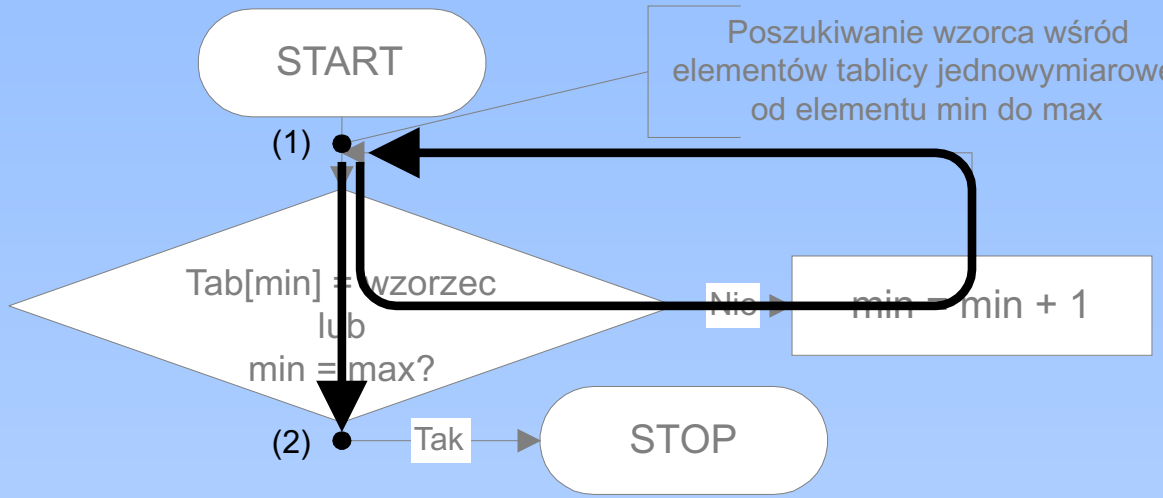
Szacowanie złożoności obliczeniowej

Do oszacowania złożoności obliczeniowej wystarczy policzyć liczbę porównań występujących w programie.

Szacowanie złożoności obliczeniowej

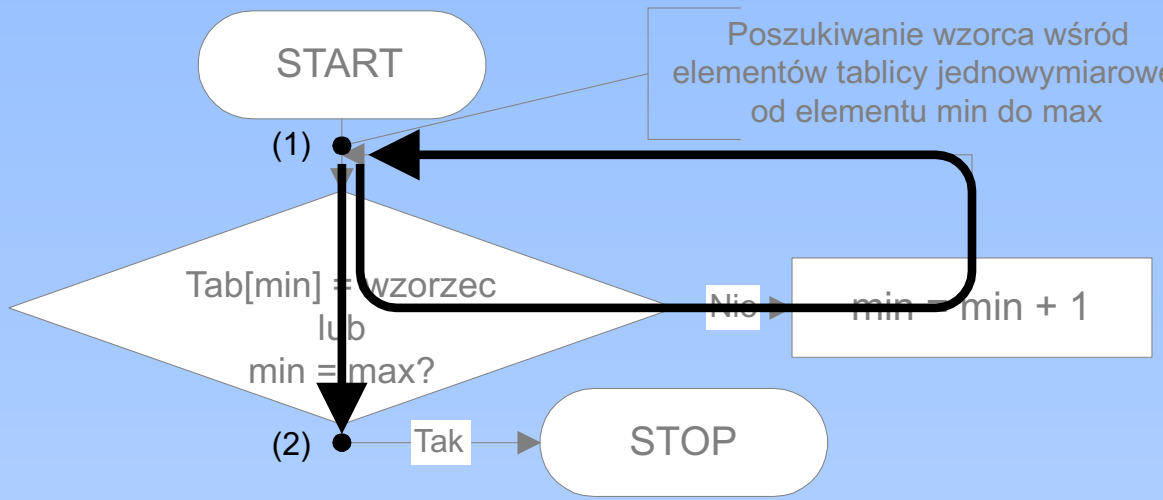
Do oszacowania złożoności obliczeniowej wystarczy policzyć liczbę porównań występujących w programie.

Dlaczego?



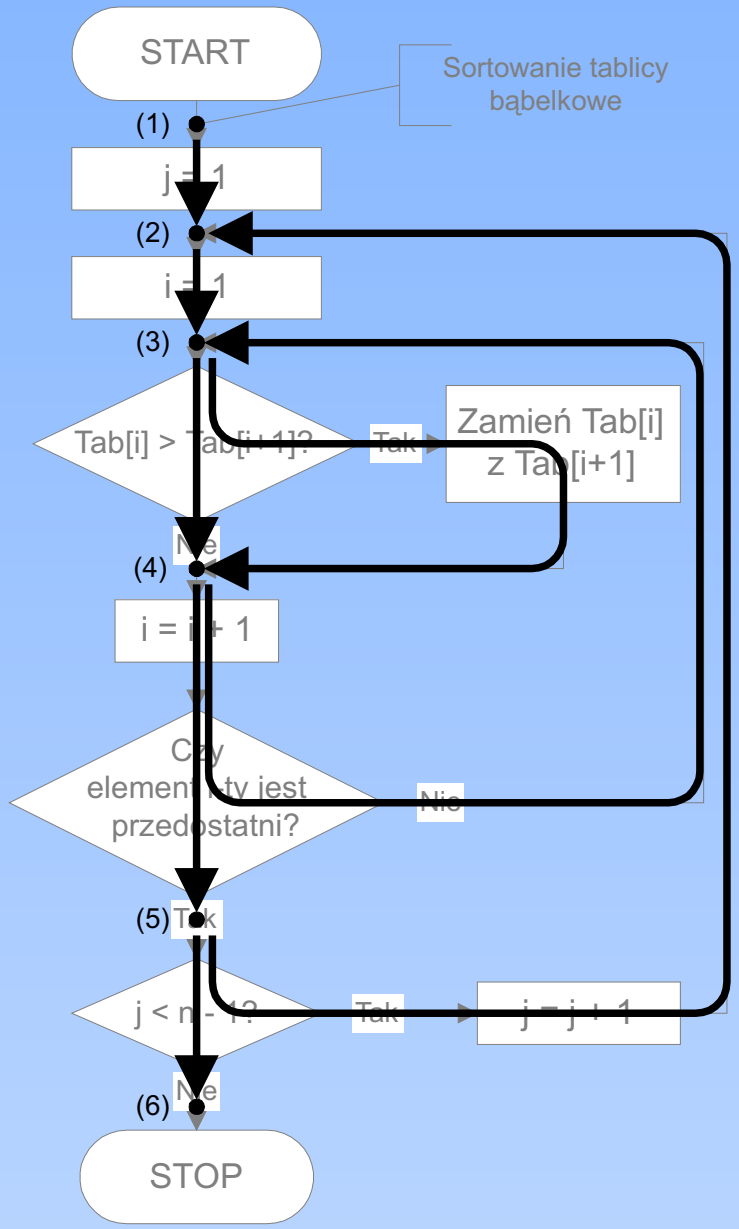
Szacowanie złożoności obliczeniowej

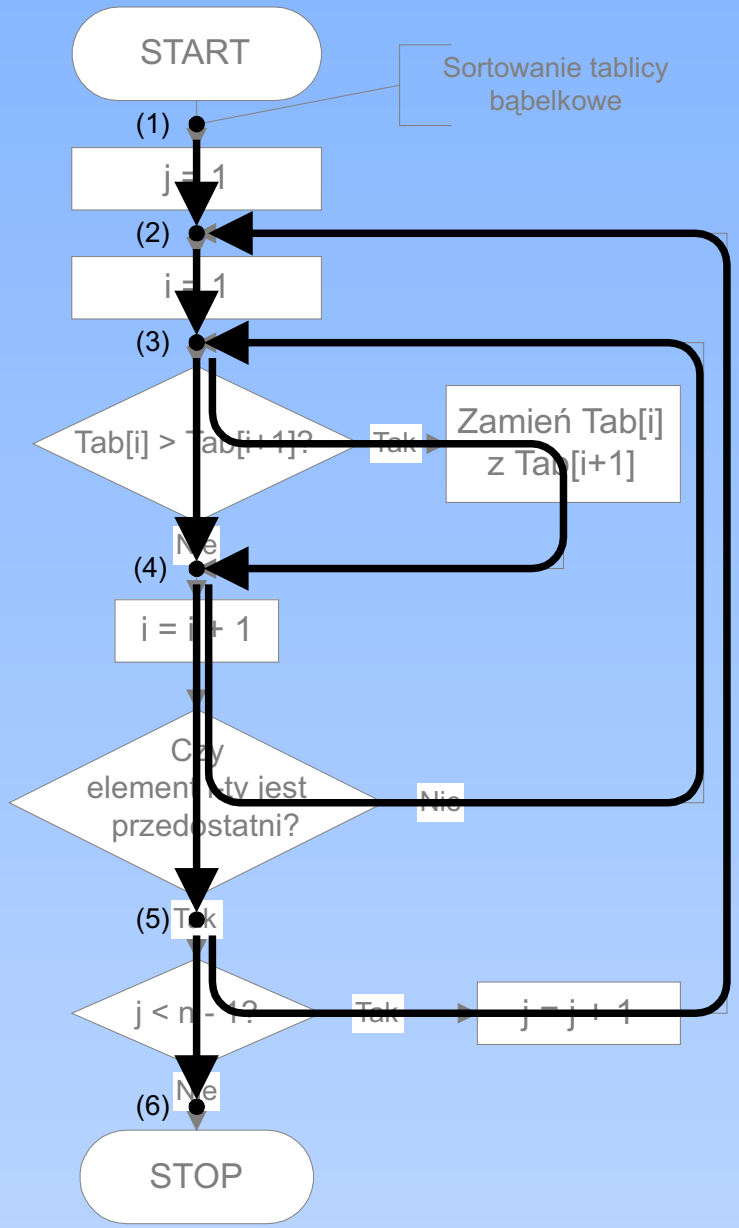
Do oszacowania złożoności obliczeniowej wystarczy policzyć liczbę porównań występujących w programie.
Dlaczego?



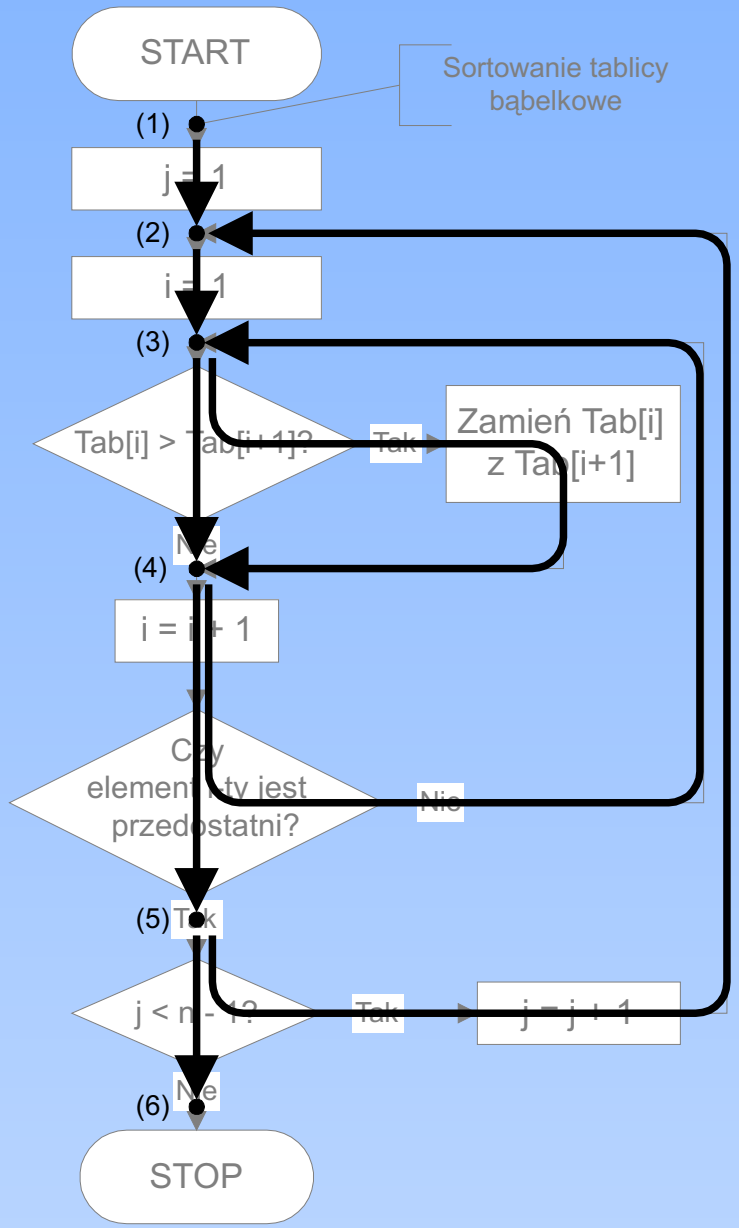
$$\text{Koszt} = n * K_{11} + K_{12}$$

n — liczba przejść pętli równa liczbie porównań
K_{ij} – koszt przejścia z punktu kontrolnego i do j (stały!)

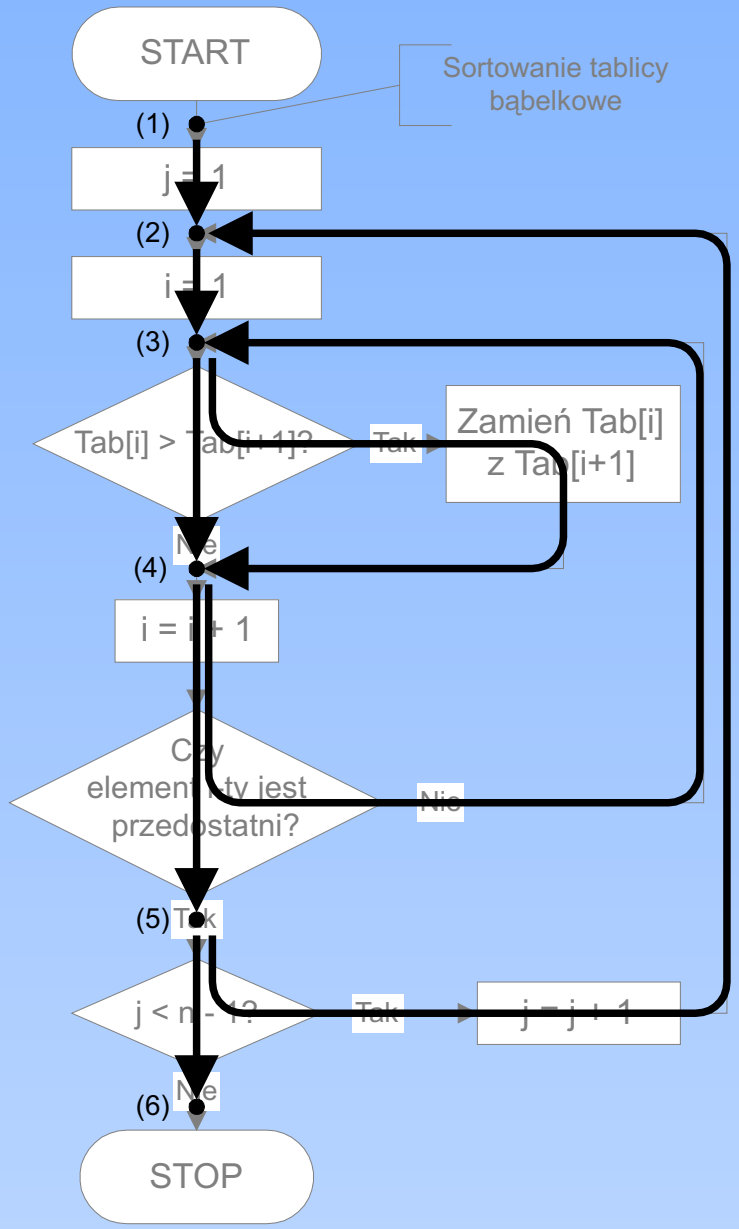

START
Sortowanie tablicy bąbelkowe
(1)
j = 1
(2)
i = 1
(3)
Tak
Zamień Tab[i] z Tab[i+1]
Nie
(4)
i = i + 1
Czy element i-ty jest przedostatni?
Nie
(5)
Tak
j < n - 1?
Tak
j = j + 1
Nie
(6)
STOP


START
Sortowanie tablicy bąbelkowe
(1)
j = 1
(2)
i = 1
(3)
Tak
Zamień Tab[i] z Tab[i+1]
(4)
i = i + 1
Nie
(5)
Tak
j = j + 1
(6)
STOP

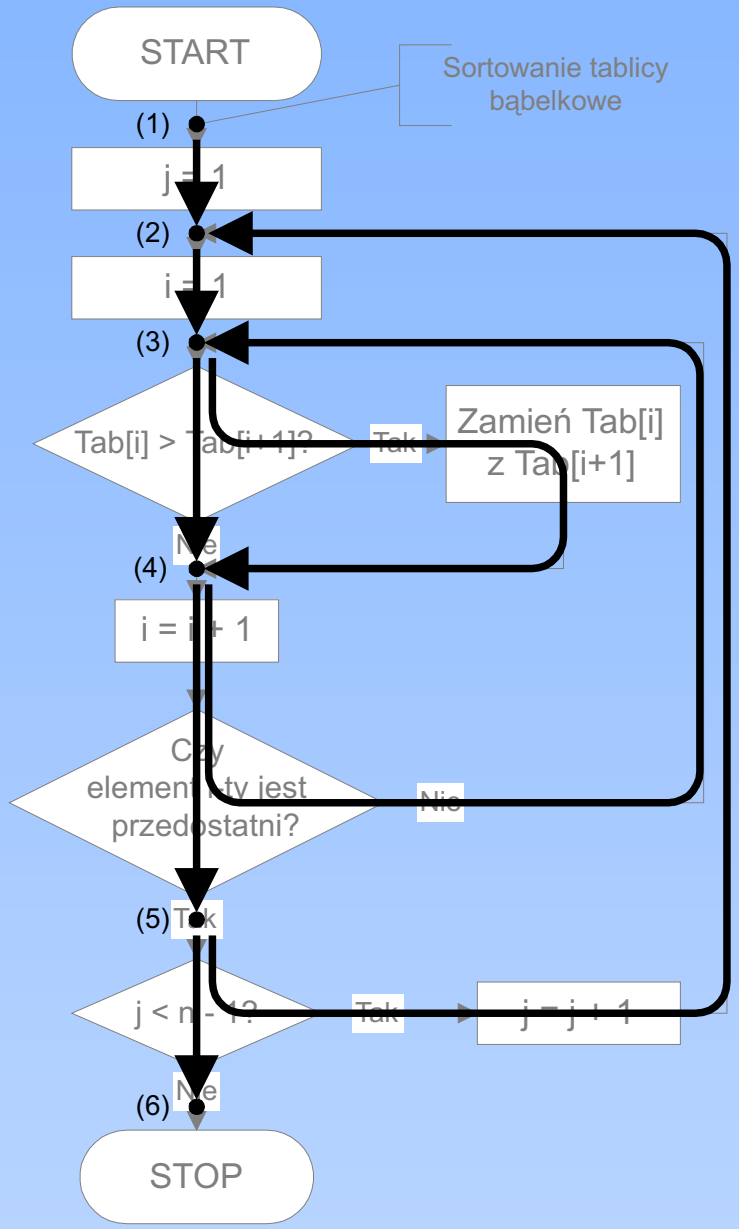
Koszt = K_{12}



$$\text{Koszt} = K_{12} + (n-1)K_{22}$$

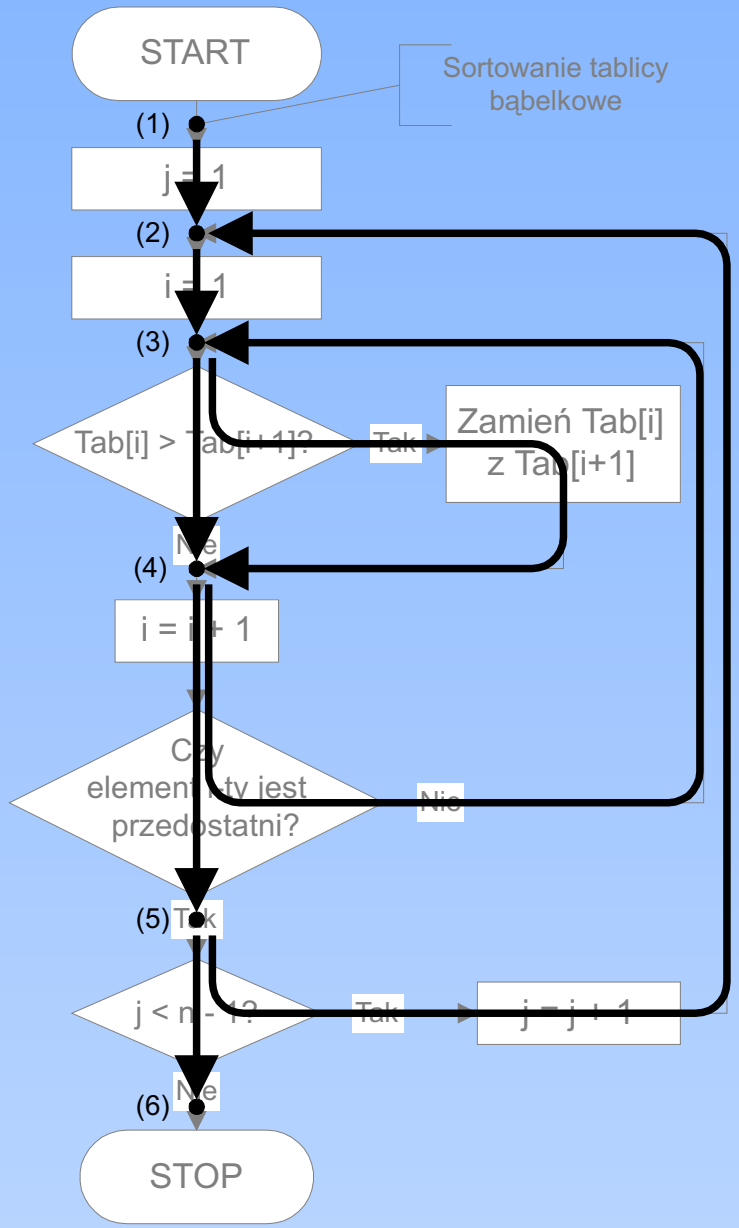


$$\text{Koszt} = K_{12} + (n-1)K_{22} + K_{56}$$



$$\text{Koszt} = K_{12} + (n-1)K_{22} + K_{56} =$$

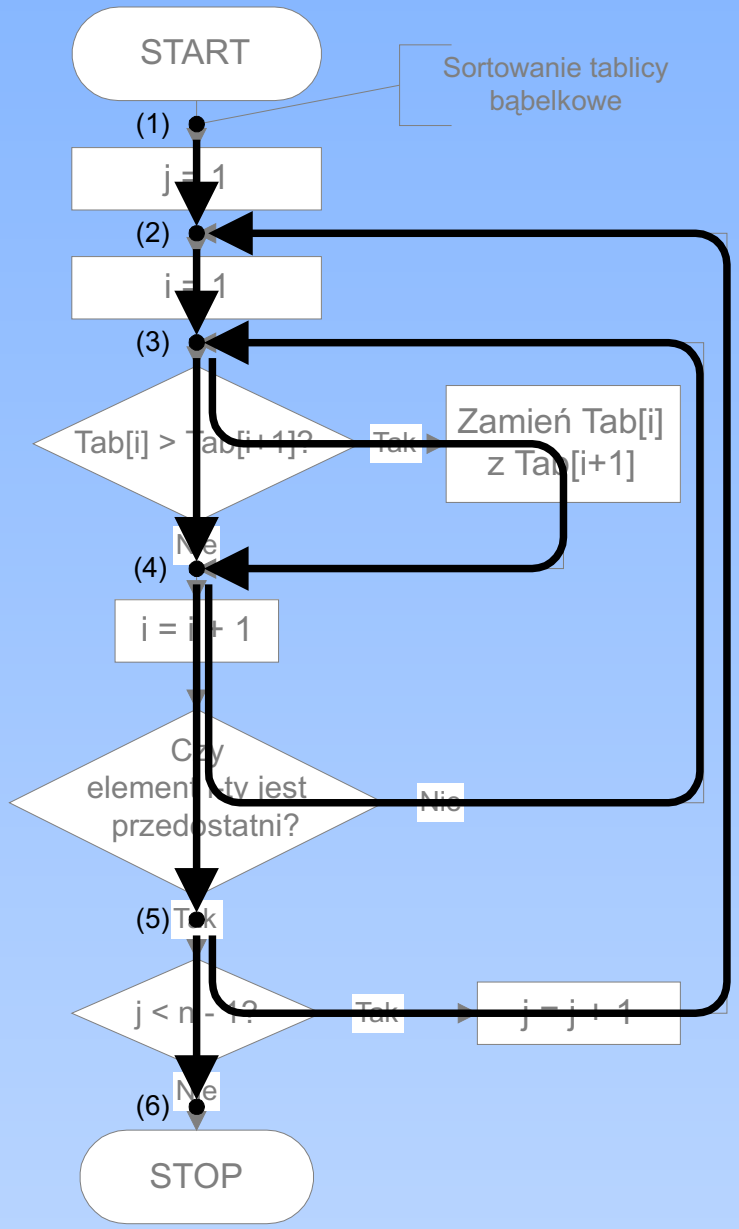
$$= K_{12} + (n-1)(K_{23} + (n-1)(K_{34} + K_{43}) + K_{45}) + K_{56}$$



$$\text{Koszt} = K_{12} + (n-1)K_{22} + K_{56} =$$

$$= K_{12}+(n-1)(K_{23}+(n-1)(K_{34}+K_{43})+K_{45})+K_{56} =$$

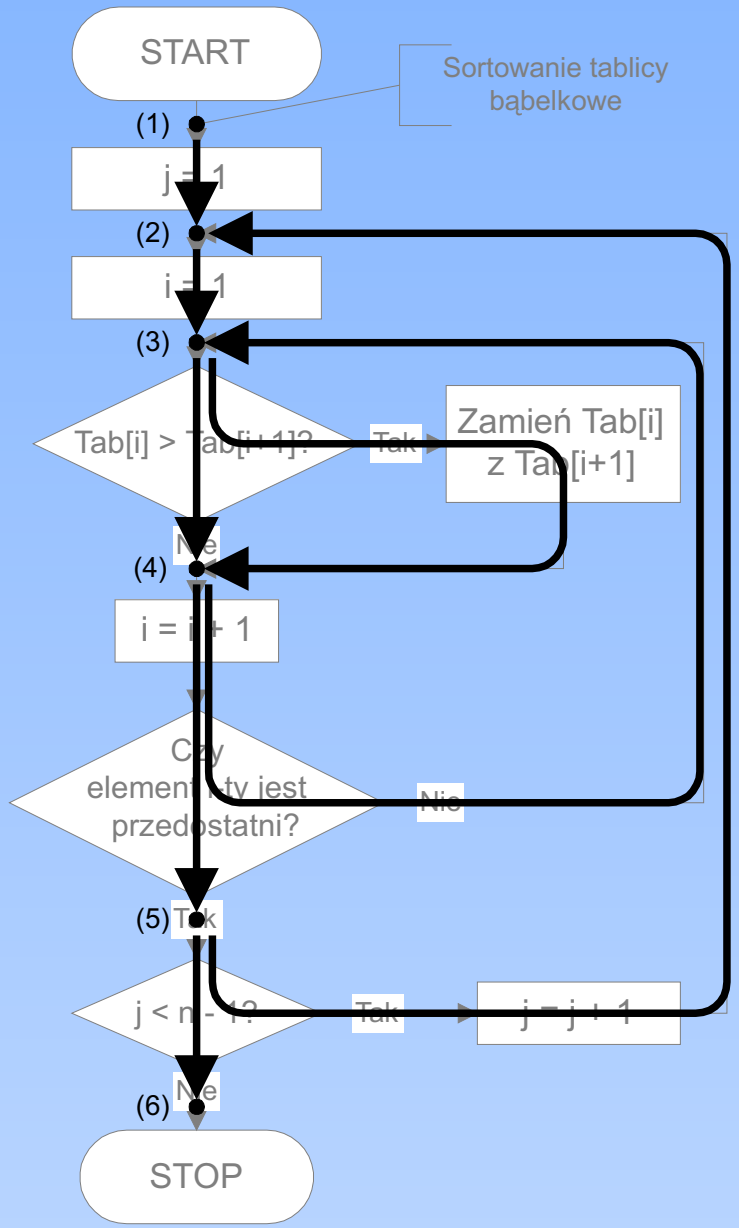
$$= (K_{34} + K_{43})n^2$$



$$\text{Koszt} = K_{12} + (n-1)K_{22} + K_{56} =$$

$$= K_{12}+(n-1)(K_{23}+(n-1)(K_{34}+K_{43})+K_{45})+K_{56} =$$

$$= (K_{34} + K_{43})n^2 +$$
$$+ (K_{23}-2K_{34}-2K_{43}+K_{45})n +$$
$$+ K_{12}-K_{23}+K_{34}+K_{43}-K_{45}+K_{56}$$



$$\text{Koszt} = K_{12} + (n-1)K_{22} + K_{56} =$$

$$= K_{12}+(n-1)(K_{23}+(n-1)(K_{34}+K_{43})+K_{45})+K_{56} =$$

$$= (K_{34} + K_{43})n^2 +$$
$$+ (K_{23}-2K_{34}-2K_{43}+K_{45})n +$$
$$+ K_{12}-K_{23}+K_{34}+K_{43}-K_{45}+K_{56}$$

Złożoność – $O(n^2)$

Teoria a praktyka

Rzeczywiste czasy wykonania programów sortowania

Same klucze	Tablica	uporządkowana		losowa		odwrotnie uporządkowana	
	n =	256	512	256	512	256	512
proste wstawianie		12	23	366	1444	704	2836
wstawianie połówkowe		56	125	373	1327	662	2490
proste wybieranie		489	1907	509	1956	695	2675
sortowanie drzewiaste		116	253	110	241	104	226
sortowanie bąbelkowe		540	2165	1026	4054	1492	5931
sortowanie szybkie		31	69	60	146	37	79

Teoria a praktyka

Rzeczywiste czasy wykonania programów sortowania

Same klucze	Tablica	uporządkowana		losowa		odwrotnie uporządkowana	
	n =	256	512	256	512	256	512
proste wstawianie		12	23	366	1444	704	2836
wstawianie połówkowe		56	125	373	1327	662	2490
proste wybieranie		489	1907	509	1956	695	2675
sortowanie drzewiaste		116	253	110	241	104	226
sortowanie bąbelkowe		540	2165	1026	4054	1492	5931
sortowanie szybkie		31	69	60	146	37	79

- Usprawnienie wstawiania połówkowego nie ma praktycznie żadnego znaczenia

Teoria a praktyka

Rzeczywiste czasy wykonania programów sortowania

Same klucze	Tablica	uporządkowana		losowa		odwrotnie uporządkowana	
	n =	256	512	256	512	256	512
proste wstawianie		12	23	366	1444	704	2836
wstawianie połówkowe		56	125	373	1327	662	2490
proste wybieranie		489	1907	509	1956	695	2675
sortowanie drzewiaste		116	253	110	241	104	226
sortowanie bąbelkowe		540	2165	1026	4054	1492	5931
sortowanie szybkie		31	69	60	146	37	79

- Usprawnienie wstawiania połówkowego nie ma praktycznie żadnego znaczenia
- Sortowanie bąbelkowe jest zdecydowanie najgorszą ze wszystkich metod

Teoria a praktyka

Rzeczywiste czasy wykonania programów sortowania

Same klucze	Tablica	uporządkowana		losowa		odwrotnie uporządkowana	
	n =	256	512	256	512	256	512
proste wstawianie		12	23	366	1444	704	2836
wstawianie połówkowe		56	125	373	1327	662	2490
proste wybieranie		489	1907	509	1956	695	2675
sortowanie drzewiaste		116	253	110	241	104	226
sortowanie bąbelkowe		540	2165	1026	4054	1492	5931
sortowanie szybkie		31	69	60	146	37	79

- Usprawnienie wstawiania połówkowego nie ma praktycznie żadnego znaczenia
- Sortowanie bąbelkowe jest zdecydowanie najgorszą ze wszystkich metod
- Sortowanie szybkie jest rzeczywiście szybkie

Klucze + dane	Tablica	uporządkowana		losowa		odwrotnie uporząd.	
	+ dane	NIE	TAK	NIE	TAK	NIE	TAK
proste wstawianie		12	46	366	1129	704	2150
wstawianie połówkowe		56	76	373	1105	662	2070
proste wybieranie		489	547	509	607	695	1430
sortowanie drzewiaste		116	264	110	246	104	227
sortowanie bąbelkowe		540	610	1026	3212	1492	5599
sortowanie szybkie		31	55	60	137	37	75

Klucze + dane	Tablica	uporządkowana		losowa		odwrotnie uporząd.	
	+ dane	NIE	TAK	NIE	TAK	NIE	TAK
proste wstawianie		12	46	366	1129	704	2150
wstawianie połówkowe		56	76	373	1105	662	2070
proste wybieranie		489	547	509	607	695	1430
sortowanie drzewiaste		116	264	110	246	104	227
sortowanie bąbelkowe		540	610	1026	3212	1492	5599
sortowanie szybkie		31	55	60	137	37	75

- Metoda prostego wyboru znacznie zyskała wśród metod prostych

Klucze + dane	Tablica	uporządkowana		losowa		odwrotnie uporząd.	
	+ dane	NIE	TAK	NIE	TAK	NIE	TAK
proste wstawianie		12	46	366	1129	704	2150
wstawianie połówkowe		56	76	373	1105	662	2070
proste wybieranie		489	547	509	607	695	1430
sortowanie drzewiaste		116	264	110	246	104	227
sortowanie bąbelkowe		540	610	1026	3212	1492	5599
sortowanie szybkie		31	55	60	137	37	75

- Metoda prostego wyboru znacznie zyskała wśród metod prostych
- Sortowanie bąbelkowe jest dalej zdecydowanie najgorsze (straciło znaczenie)

Klucze + dane	Tablica	uporządkowana		losowa		odwrotnie uporząd.	
	+ dane	NIE	TAK	NIE	TAK	NIE	TAK
proste wstawianie		12	46	366	1129	704	2150
wstawianie połówkowe		56	76	373	1105	662	2070
proste wybieranie		489	547	509	607	695	1430
sortowanie drzewiaste		116	264	110	246	104	227
sortowanie bąbelkowe		540	610	1026	3212	1492	5599
sortowanie szybkie		31	55	60	137	37	75

- Metoda prostego wyboru znacznie zyskała wśród metod prostych
- Sortowanie bąbelkowe jest dalej zdecydowanie najgorsze (straciło znaczenie)
- Sortowanie szybkie umocniło nawet swoją pozycję

Klucze + dane	Tablica	uporządkowana		losowa		odwrotnie uporząd.	
	+ dane	NIE	TAK	NIE	TAK	NIE	TAK
proste wstawianie		12	46	366	1129	704	2150
wstawianie połówkowe		56	76	373	1105	662	2070
proste wybieranie		489	547	509	607	695	1430
sortowanie drzewiaste		116	264	110	246	104	227
sortowanie bąbelkowe		540	610	1026	3212	1492	5599
sortowanie szybkie		31	55	60	137	37	75

- Metoda prostego wyboru znacznie zyskała wśród metod prostych
- Sortowanie bąbelkowe jest dalej zdecydowanie najgorsze (straciło znaczenie)
- Sortowanie szybkie umocniło nawet swoją pozycję

Generalnie wyróżniamy metody sortowania prymitywne (złożoność $O(n^2)$) oraz nowoczesne — „logarytmiczne” (złożoność $O(n \log n)$)

Efektywność algorytmów w praktyce

- Dla małych zbiorów danych można pominąć zagadnienia efektywności algorytmów

Efektywność algorytmów w praktyce

- Dla małych zbiorów danych można pominąć zagadnienia efektywności algorytmów
- Dla bardzo dużych zbiorów danych najważniejsza jest klasa złożoności obliczeniowej algorytmu (wpływ czynników stałych pominiętych w notacji duże-O może okazać się dominujący przy małych i średnich wielkościach zbiorów)

Efektywność algorytmów w praktyce

- Dla małych zbiorów danych można pominąć zagadnienia efektywności algorytmów
- Dla bardzo dużych zbiorów danych najważniejsza jest klasa złożoności obliczeniowej algorytmu (wpływ czynników stałych pominiętych w notacji duże-O może okazać się dominujący przy małych i średnich wielkościach zbiorów)
- Często ważniejsze od wyboru algorytmu o dobrej klasie złożoności jest tzw. „lokalna optymalizacja” — usprawnienie fragmentów programu stanowiących tzw. „wąskie gardła” (zasada 80–20, lub nawet 90–10)

Efektywność algorytmów w praktyce

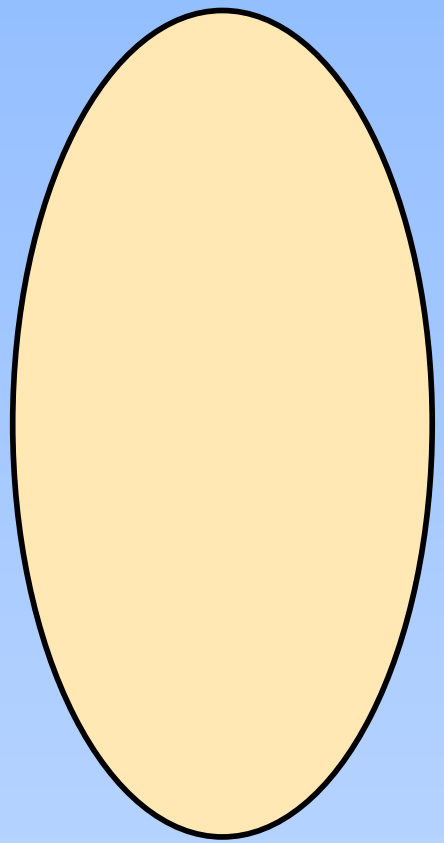
- Dla małych zbiorów danych można pominąć zagadnienia efektywności algorytmów
- Dla bardzo dużych zbiorów danych najważniejsza jest klasa złożoności obliczeniowej algorytmu (wpływ czynników stałych pominiętych w notacji duże-O może okazać się dominujący przy małych i średnich wielkościach zbiorów)
- Często ważniejsze od wyboru algorytmu o dobrej klasie złożoności jest tzw. „lokalna optymalizacja” — usprawnienie fragmentów programu stanowiących tzw. „wąskie gardła” (zasada 80–20, lub nawet 90–10)
- Malejące koszty sprzętu komputerowego i równocześnie rosnące koszty tworzenia oprogramowania prowadzą do zaniedbywania analizy efektywności oprogramowania — wówczas najważniejsze są zasady stylu programowania

Efektywność algorytmów w praktyce

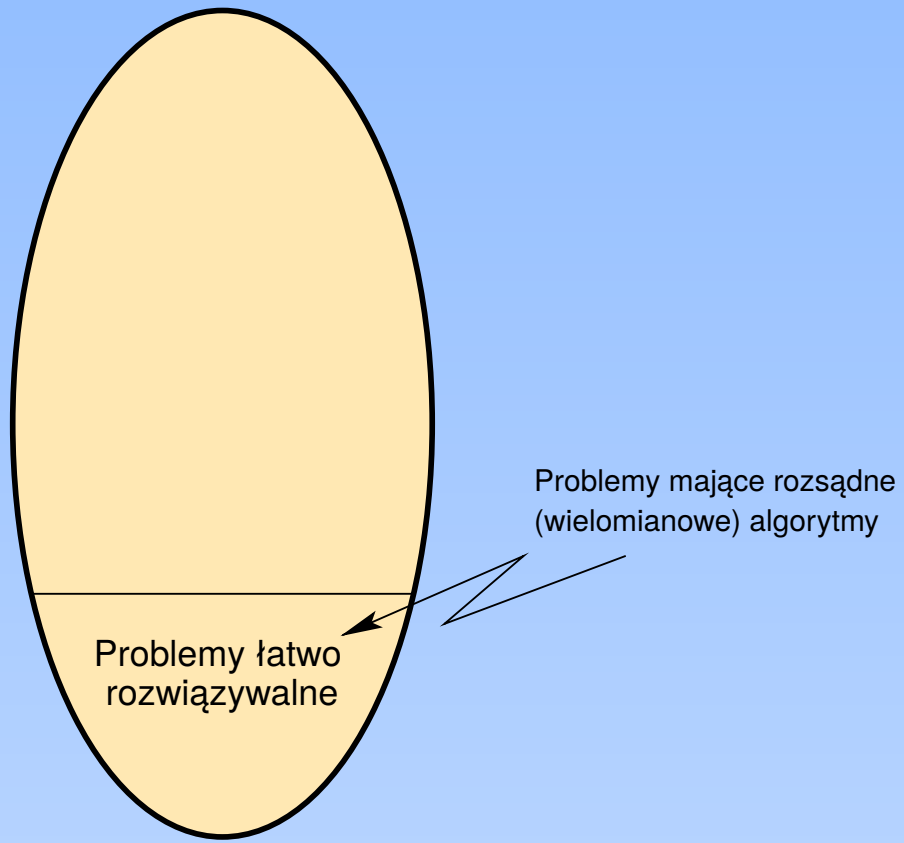
- Dla małych zbiorów danych można pominąć zagadnienia efektywności algorytmów
- Dla bardzo dużych zbiorów danych najważniejsza jest klasa złożoności obliczeniowej algorytmu (wpływ czynników stałych pominiętych w notacji duże-O może okazać się dominujący przy małych i średnich wielkościach zbiorów)
- Często ważniejsze od wyboru algorytmu o dobrej klasie złożoności jest tzw. „lokalna optymalizacja” — usprawnienie fragmentów programu stanowiących tzw. „wąskie gardła” (zasada 80–20, lub nawet 90–10)
- Malejące koszty sprzętu komputerowego i równocześnie rosnące koszty tworzenia oprogramowania prowadzą do zaniedbywania analizy efektywności oprogramowania — wówczas najważniejsze są zasady stylu programowania
- Często algorytmy o mniejszej złożoności obliczeniowej charakteryzują się większą złożonością pamięciową

Sfera problemów algorytmicznych

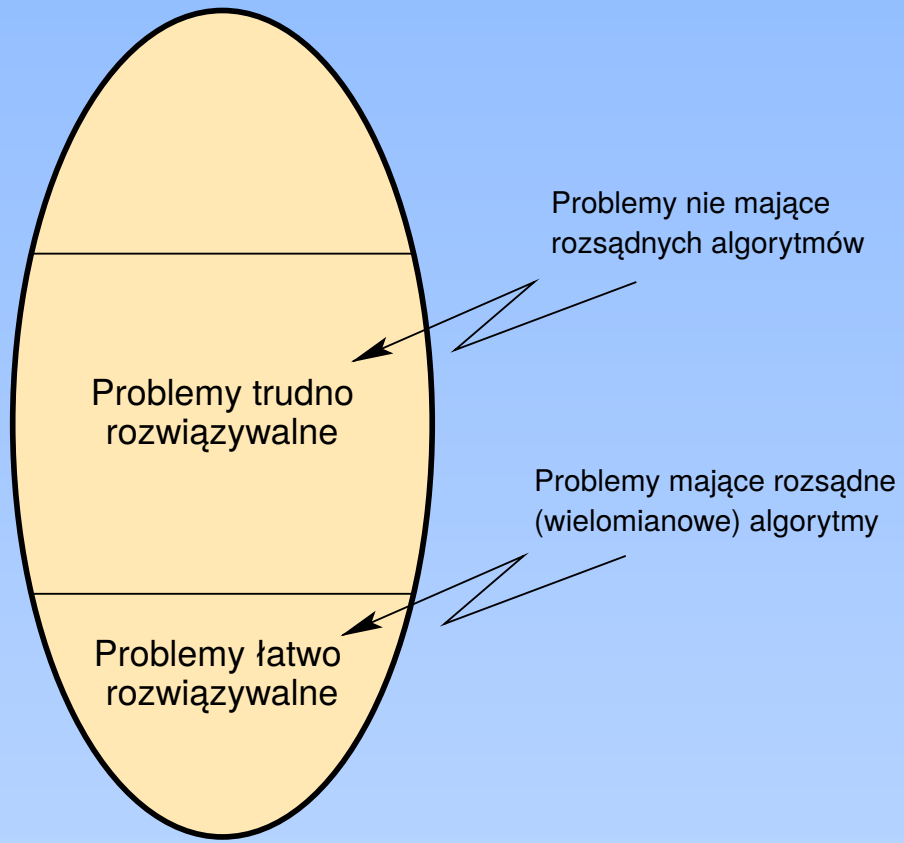
Sfera problemów algorytmicznych



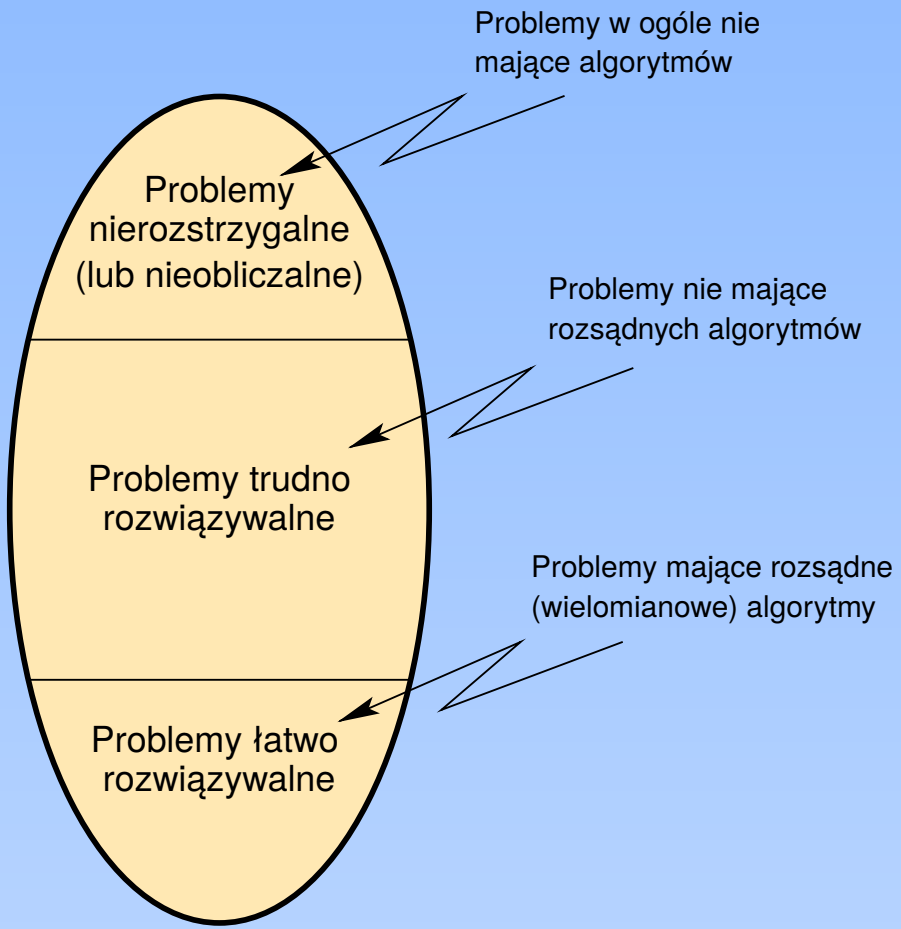
Sfera problemów algorytmicznych



Sfera problemów algorytmicznych

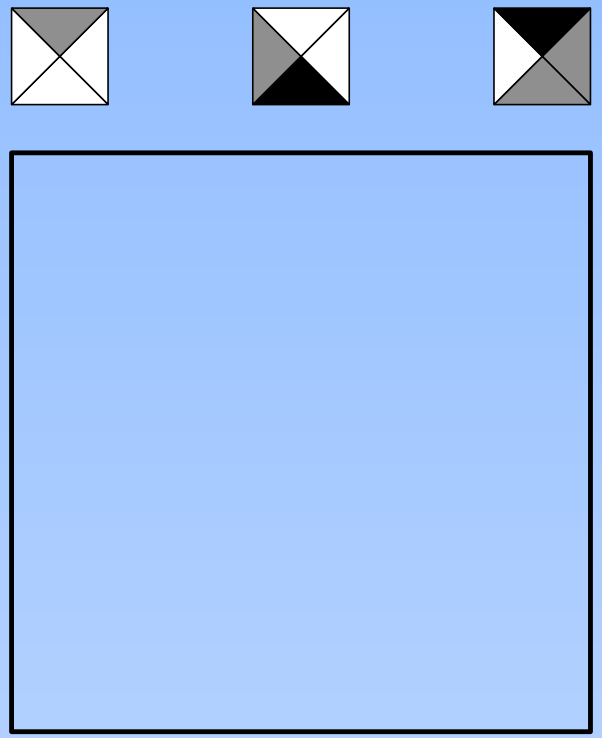


Sfera problemów algorytmicznych

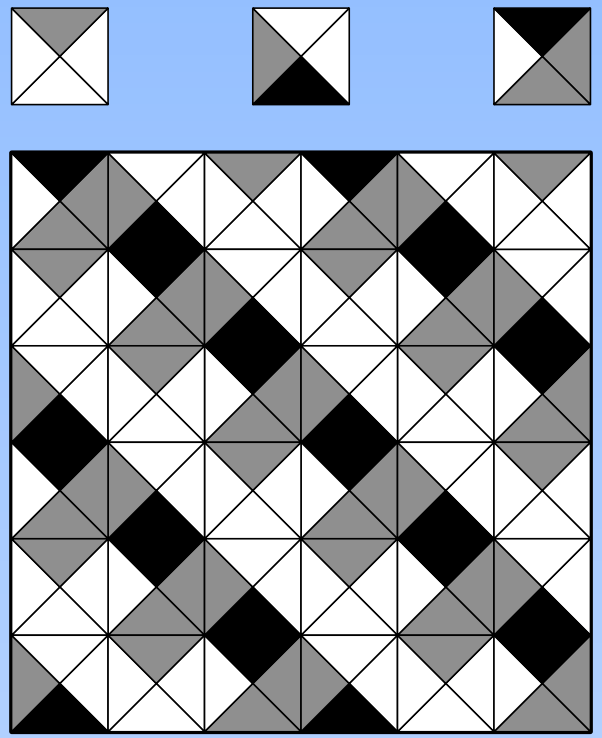


Przykład problemu nieobliczalnego — problem domina

Przykład problemu nieobliczalnego — problem domina

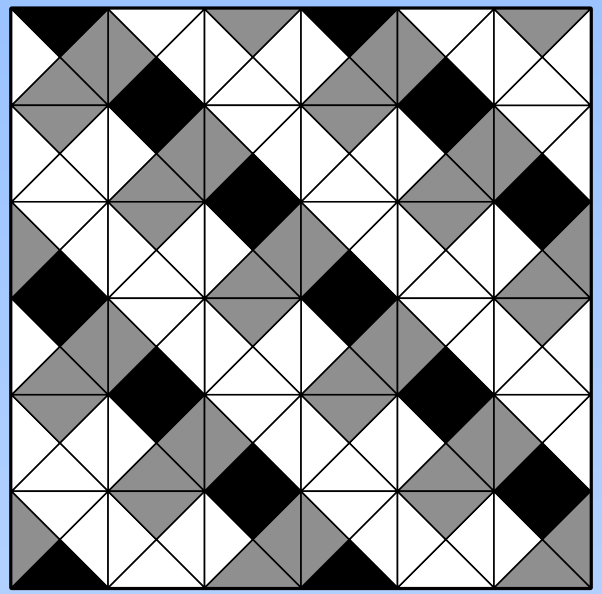


Przykład problemu nieobliczalnego — problem domina



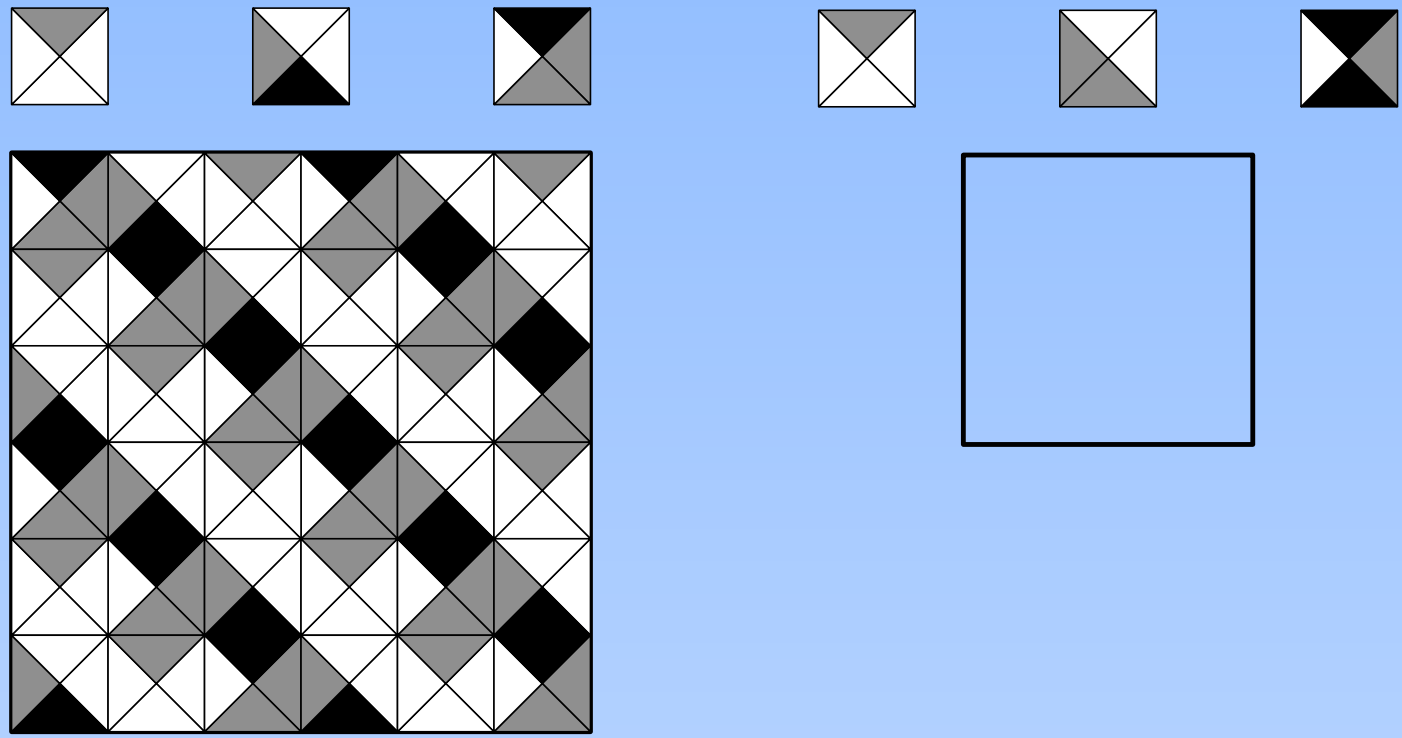
Da się!!!

Przykład problemu nieobliczalnego — problem domina



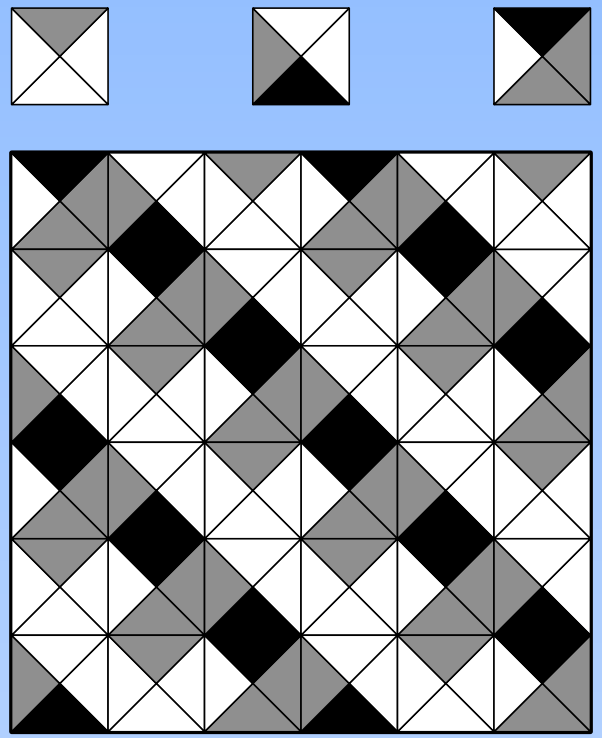
Da się!!!

Przykład problemu nieobliczalnego — problem domina

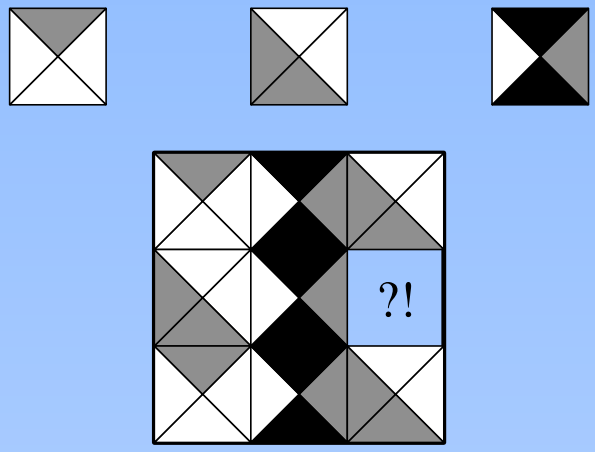


Da się!!!

Przykład problemu nieobliczalnego — problem domina



Da się!!!



Nie da się:(

Problem domina — twierdzenie

Twierdzenie 1

Dla każdego algorytmu (zapisanego w dającym się efektywnie wykonać języku programowania), który byłby przeznaczony do rozstrzygnięcia problemu domina, istnieje nieskończenie wiele dopuszczalnych zestawów danych wejściowych, dla których algorytm ten będzie działał w nieskończoność lub poda błędną odpowiedź.

Problem domina — twierdzenie

Twierdzenie 1

Dla każdego algorytmu (zapisanego w dającym się efektywnie wykonać języku programowania), który byłby przeznaczony do rozstrzygnięcia problemu domina, istnieje nieskończenie wiele dopuszczalnych zestawów danych wejściowych, dla których algorytm ten będzie działał w nieskończoność lub poda błędną odpowiedź.

Wniosek 1

Problem domina jest problemem **nierozstrzygalnym**.

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj:
   1.1. dla X parzystego $X \leftarrow X/2$

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj:
   1.1. dla X parzystego $X \leftarrow X/2$
   1.2. dla X nieparzystego $X \leftarrow 3 * X + 1$

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj:
   1.1. dla X parzystego $X \leftarrow X/2$
   1.2. dla X nieparzystego $X \leftarrow 3 * X + 1$
2. zatrzymaj obliczenia

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj:
   1.1. dla X parzystego $X \leftarrow X/2$
   1.2. dla X nieparzystego $X \leftarrow 3 * X + 1$
2. zatrzymaj obliczenia

- dla wszystkich sprawdzonych liczb algorytm zatrzymał się

Problem stopu

Mając jako dane wejściowe tekst poprawnego programu zapisanego w pewnym języku, zbudować algorytm, który by sprawdzał, czy program zatrzyma się dla pewnych dopuszczalnych dla niego danych.

$X \in \mathbb{N}$

1. dopóki $X \neq 1$ wykonuj
   $X \leftarrow X - 2$
2. zatrzymaj obliczenia

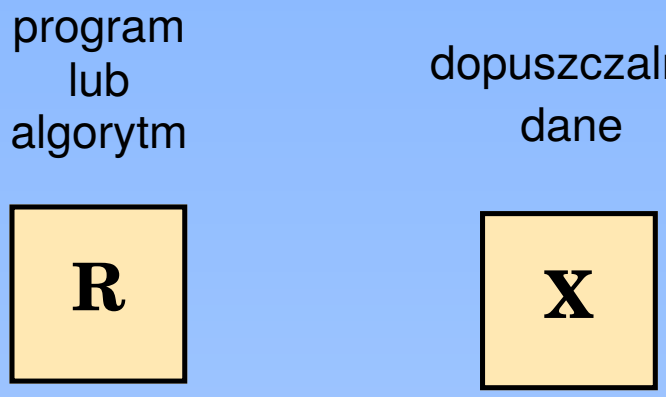
- algorytm zatrzymuje się dla X nieparzystych
- nie zatrzymuje się dla X parzystych

$X \in \mathbb{N}$

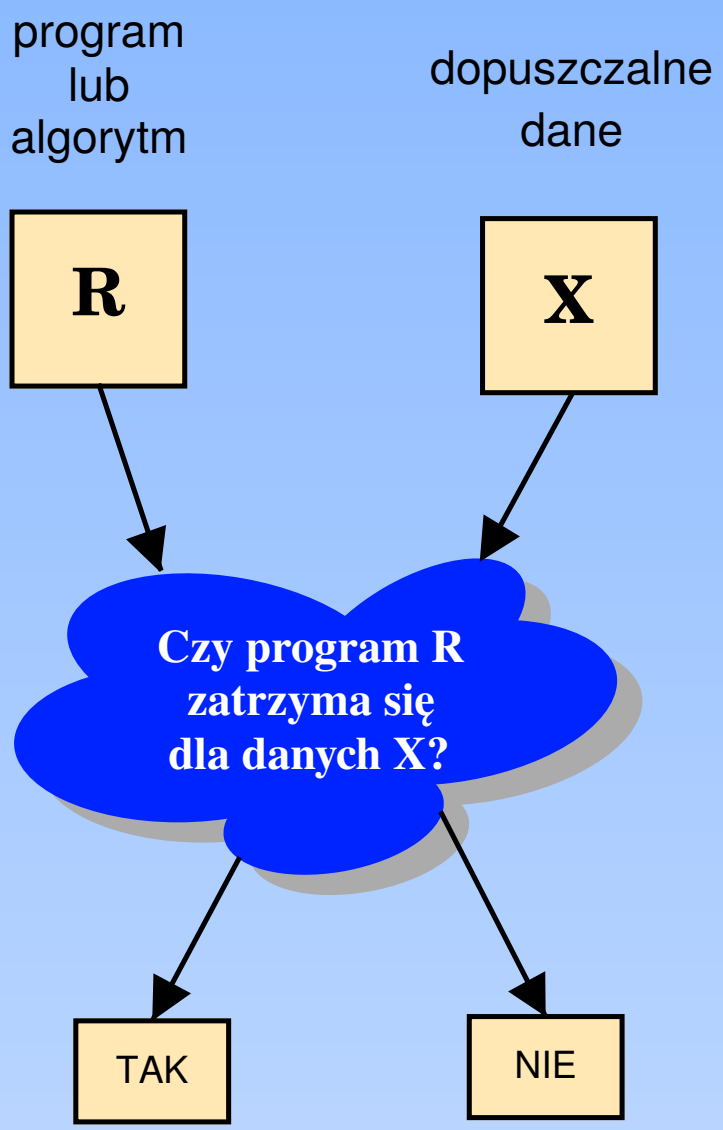
1. dopóki $X \neq 1$ wykonuj:
   1.1. dla X parzystego $X \leftarrow X/2$
   1.2. dla X nieparzystego $X \leftarrow 3 * X + 1$
2. zatrzymaj obliczenia

- dla wszystkich sprawdzonych liczb algorytm zatrzymał się
- nie udowodniono, że zatrzymuje się dla dowolnej liczby naturalnej

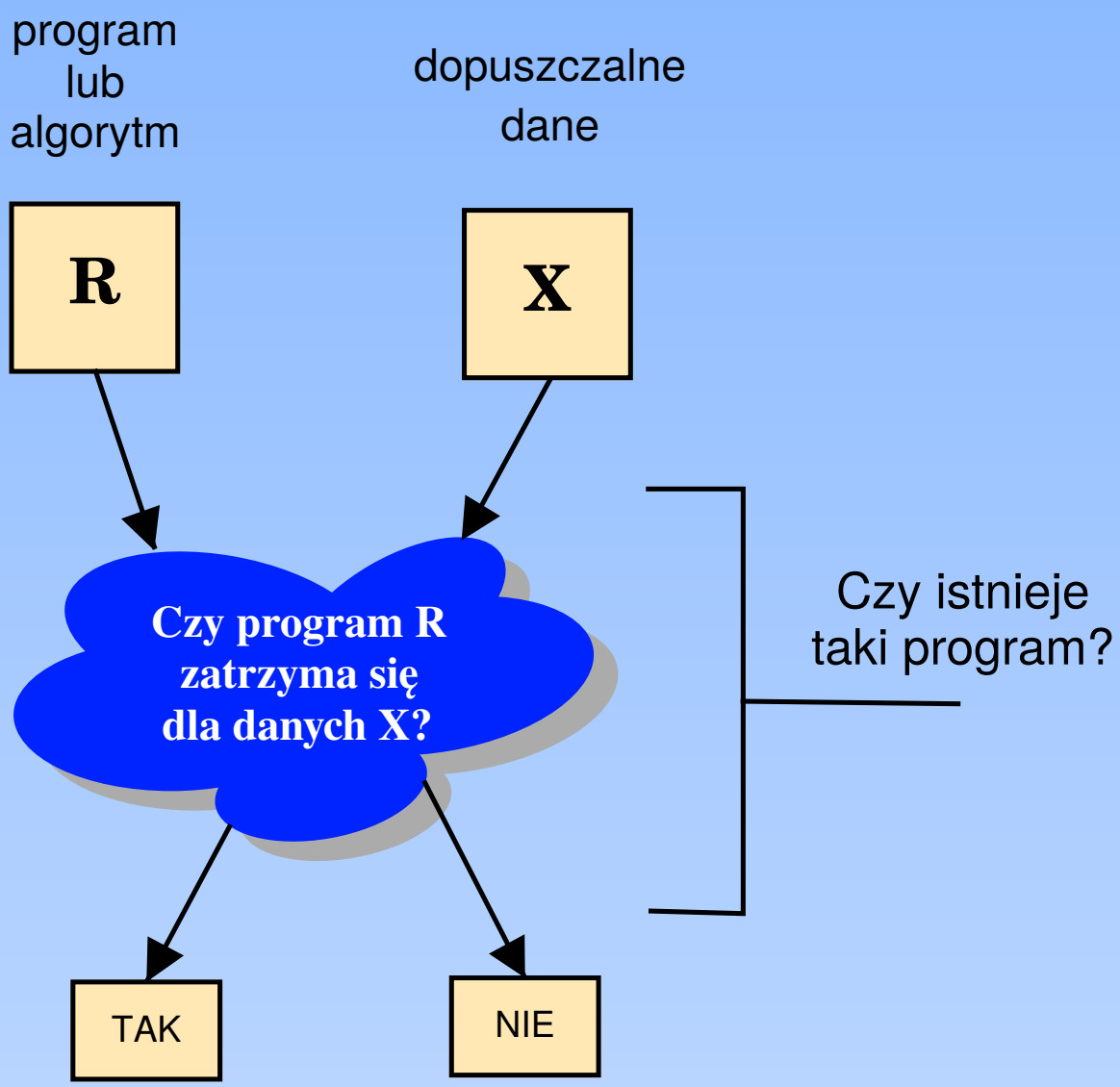
Problem stopu — rozwiązanie



Problem stopu — rozwiązanie



Problem stopu — rozwiązanie



Więcej w...

- Michael Sipser, „*Wprowadzenie do teorii obliczeń*”, WNT 2016
- Christos H. Papadimitriou, „*Złożoność obliczeniowa*”, Helion 2012
- Paul Beynon-Davies, „*Inżynieria systemów informacyjnych*”, WNT 2004

Podsumowanie

- ## Zagadnienia podstawowe

1. Wymień najważniejsze cechy dobrego oprogramowania.
2. Jakie podstawowe czynniki należy uwzględnić przy ocenie efektywności algorytmów.
3. Uporządkuj według rosnącej złożoności obliczeniowej $\mathrm{O}(n)$, $\mathrm{O}(\log n)$, $\mathrm{O}(n^2)$, $\mathrm{O}(2^n)$, $\mathrm{O}(n!)$, $\mathrm{O}(n^n)$, $\mathrm{O}(n \log n)$, $\mathrm{O}(n^{n^n})$
4. W jaki sposób można sprawdzić złożoność obliczeniową?
5. Dlaczego do oszacowania złożoności obliczeniowej algorytmu wystarczy policzyć liczbę wyliczanych w trakcie jego wykonania porównań?
6. Jaki jest sens poszukiwania nowych algorytmów, jeśli są już dostępne rozwiązania danego problemu?
7. Czym się różnią algorytmy o rozsądnym i nierozsądnym czasie działania?
8. Co to znaczy, że problem jest nierozstrzygalny?
9. Dlaczego problem domina jest przykładem problemu nieobliczalnego?
10. Jaka jest definicja problemu stopu?

- ## Zagadnienia rozszerzające

1. Podaj przykłady, kiedy algorytm (np. sortowania) o mniejszej złożoności obliczeniowej będzie działał wolniej od algorytmu o większej złożoności.

2. Zapoznaj się z klasyfikacją problemów algorytmicznych. Czym są problemy P, NP i NP-zupełne?
3. Czym są, do czego służą i czym się różnią notacje “O” (duże O), “Ω” (duże omega) i “Θ” (teta)? A notacje “o” (małe o) i “ω” (małe omega)?
4. Jakie są inne niż podane na wykładzie problemy nierozstrzygalne?
5. Jakie są sposoby rozwiązywania problemów nieobliczanych?
6. Czy przegląd zupełny może posłużyć do określenia rozstrzygalności instancji (przykładu) danego problemu?
7. Czy dla każdego problemu jesteśmy w stanie zaproponować algorytm o wykładniczej klasie złożoności obliczeniowej?

• Zadania

1. Oszacuj złożoność obliczeniową programu na przecięcia zera.
2. Oszacuj złożoność obliczeniową i pamięciową funkcji napisanego przez Ciebie programu na przetwarzanie obrazów.
3. Przejrzyj napisany przez Ciebie program na przetwarzanie obrazów i popraw jego efektywność tam, gdzie jest to możliwe.
4. Wyznacz złożoność obliczeniową w najlepszym i najgorszym przypadku algorytmu sortowania szybkiego.
5. Znajdź algorytmy na wyliczanie liczb pierwszych i dowiedz się, jaka jest ich złożoność obliczeniowa.

Indeks