

Stefan Sokołowski

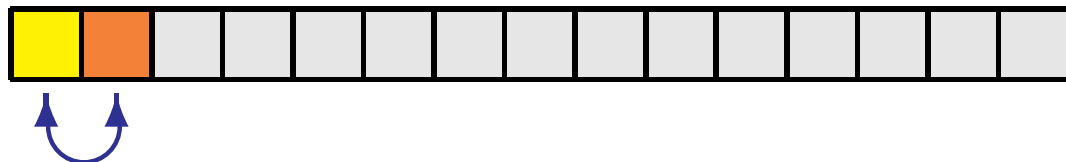
PROGRAMOWANIE OBIEKTOWE II

wykład 13

**Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025**

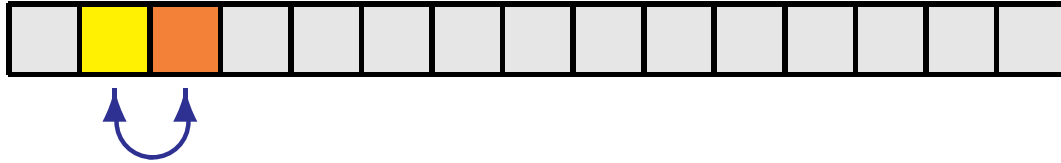
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



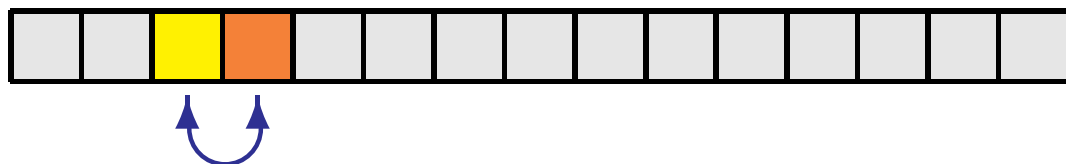
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



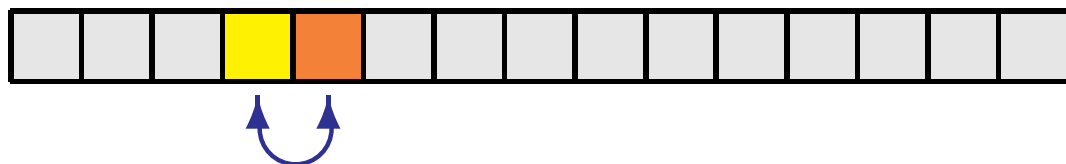
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



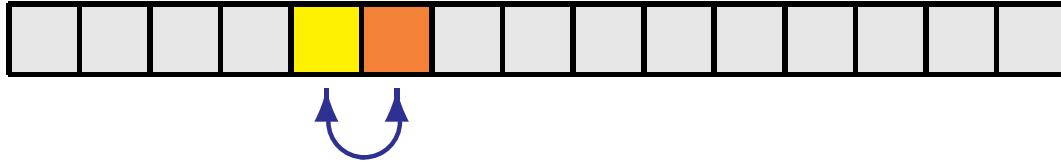
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



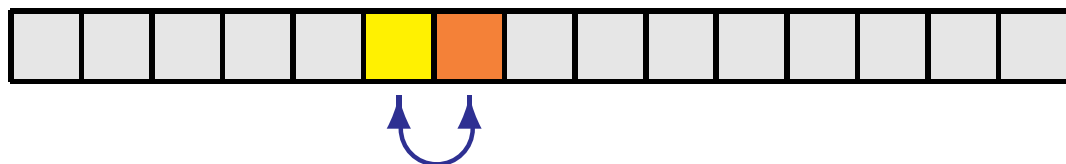
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



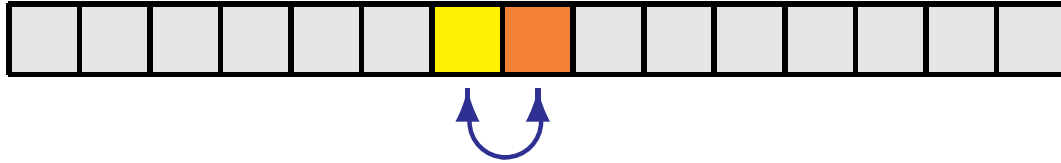
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



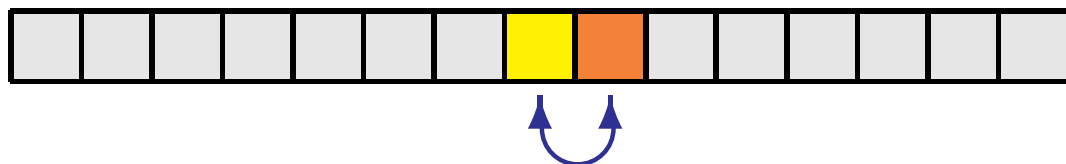
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



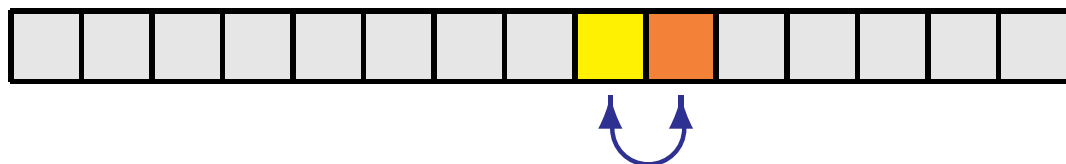
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



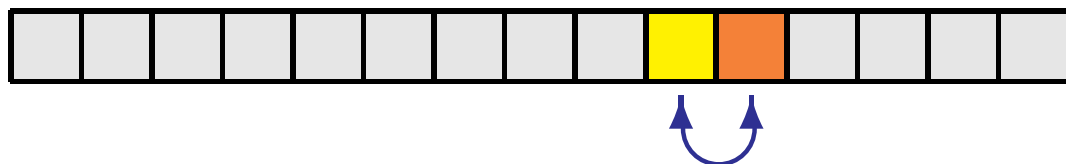
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



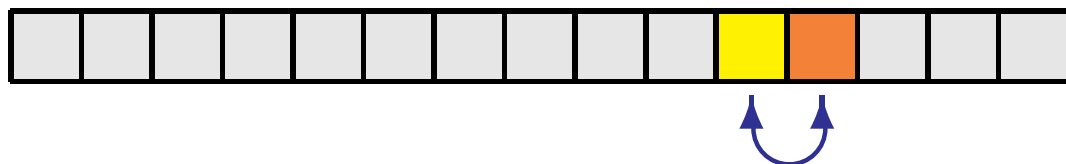
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



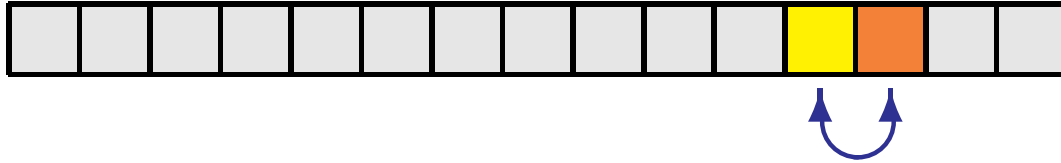
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



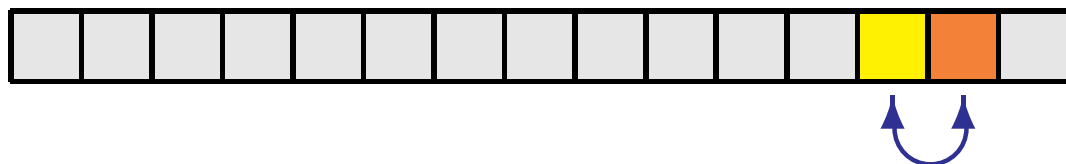
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



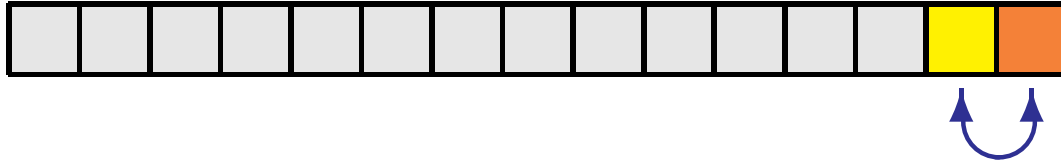
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



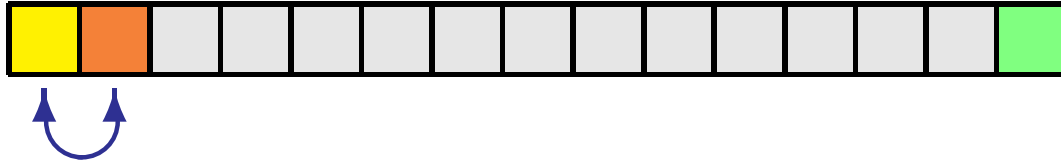
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort:**



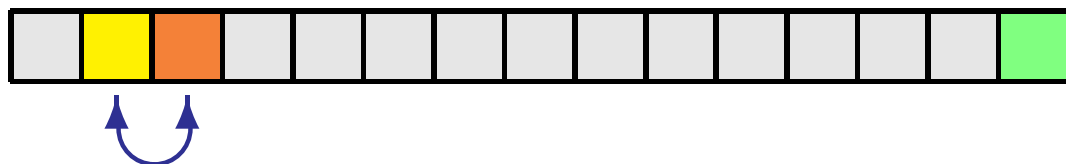
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



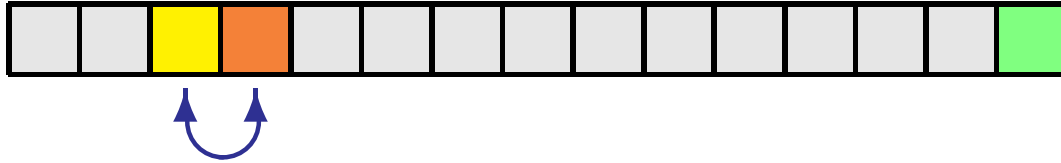
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



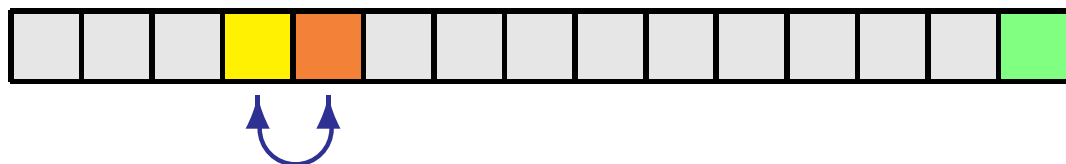
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



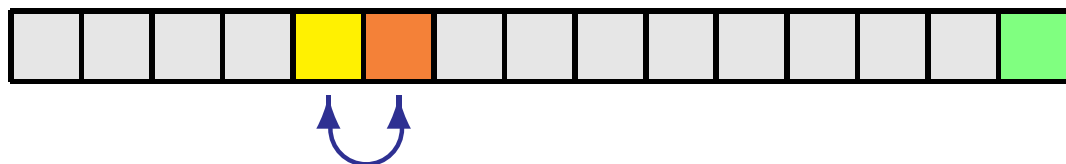
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



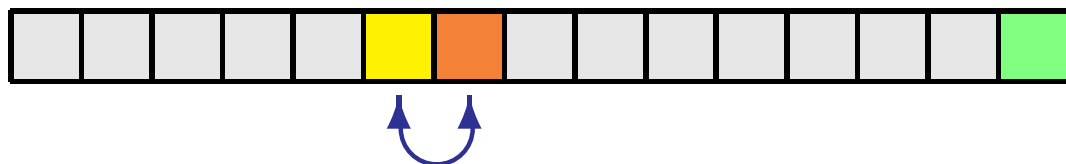
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



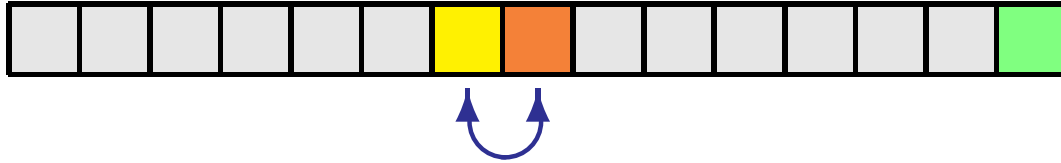
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



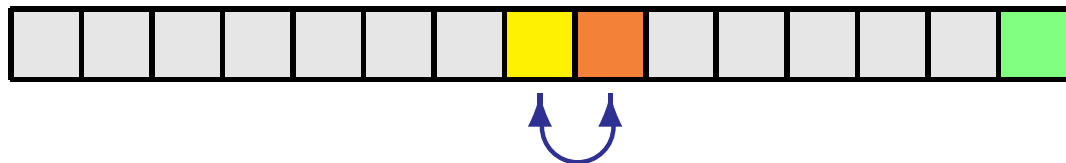
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



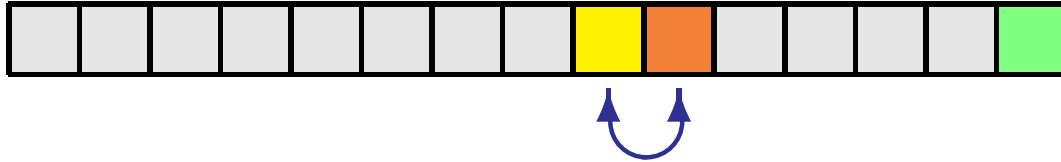
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



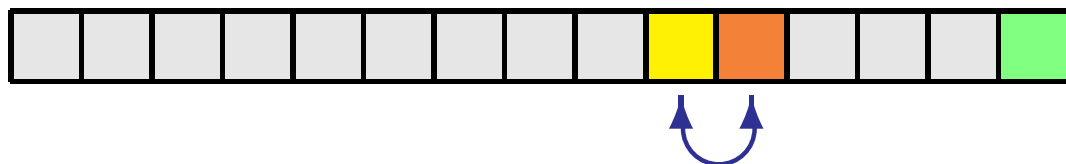
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



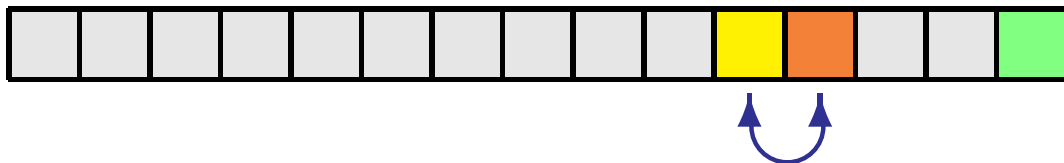
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



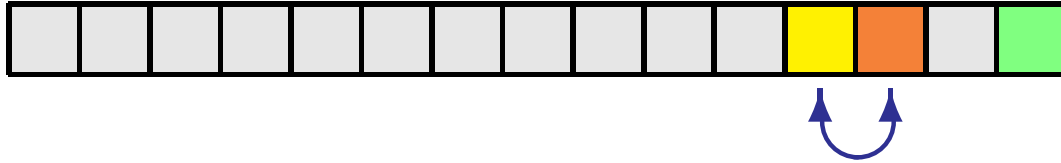
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



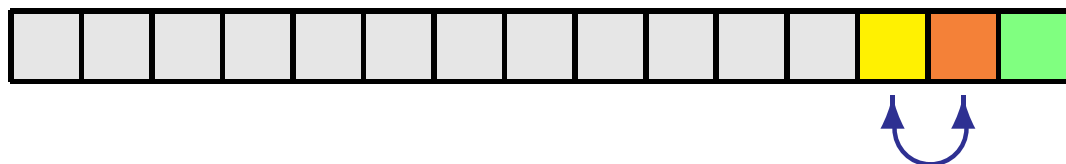
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



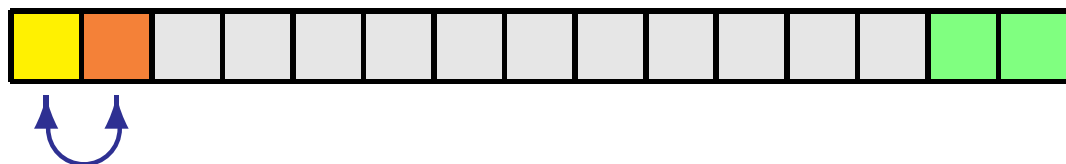
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



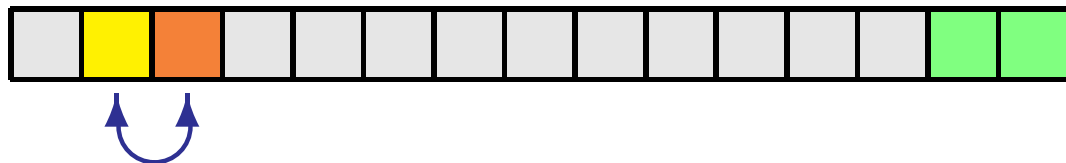
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



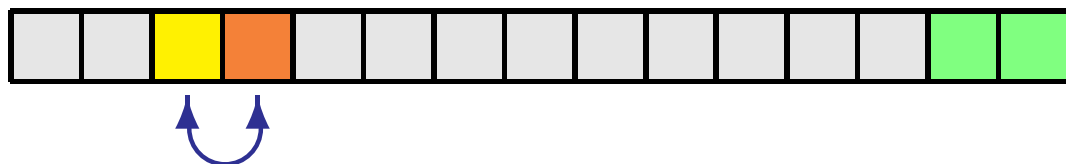
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



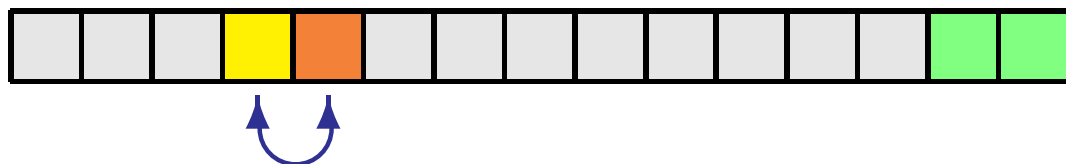
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



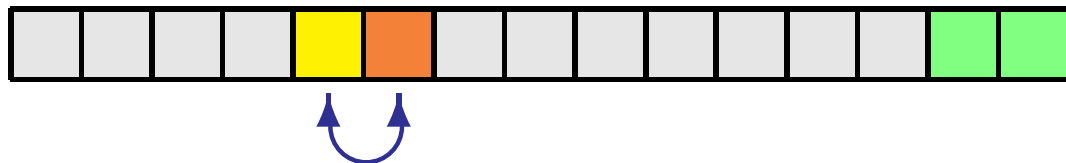
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



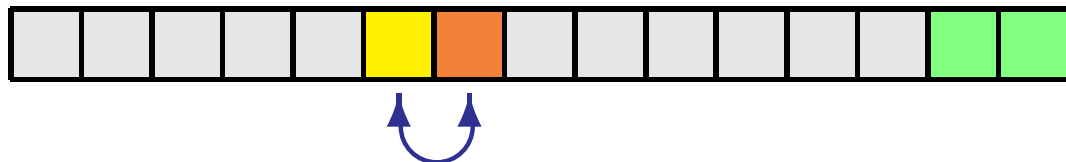
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



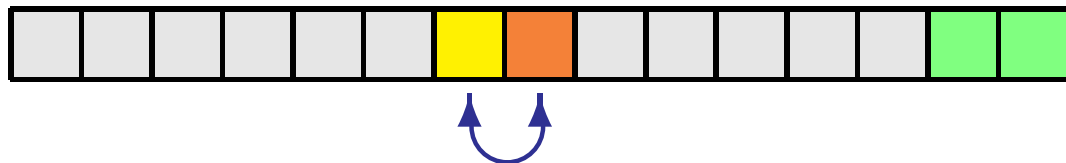
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



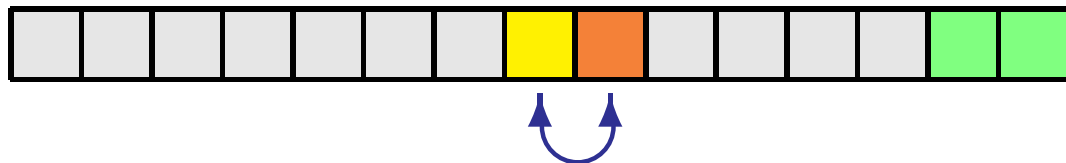
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



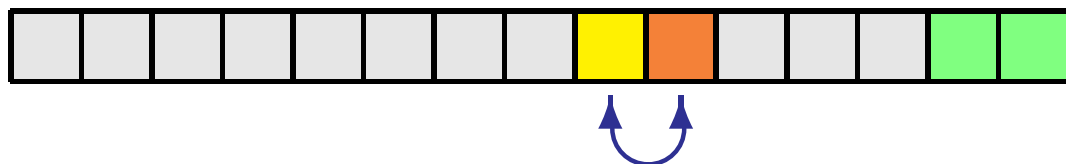
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



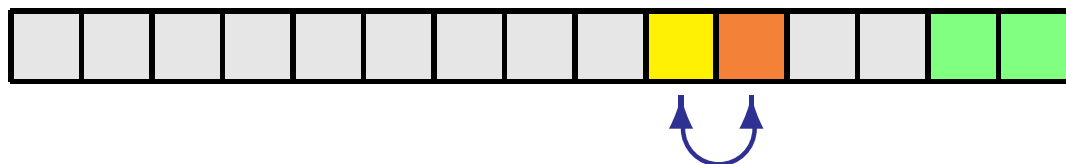
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



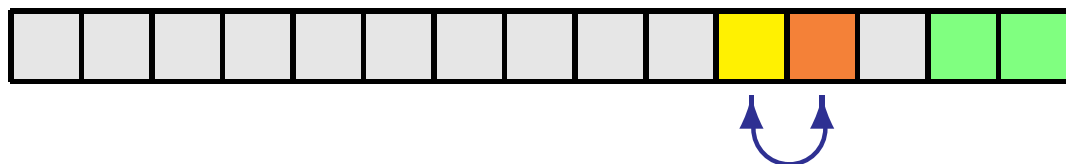
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



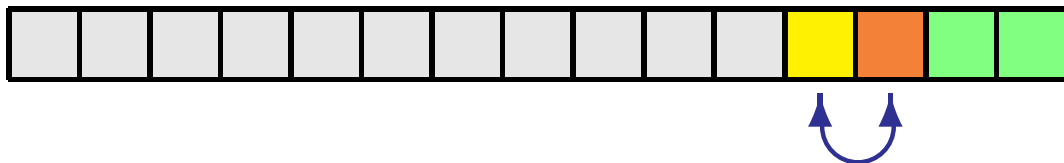
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



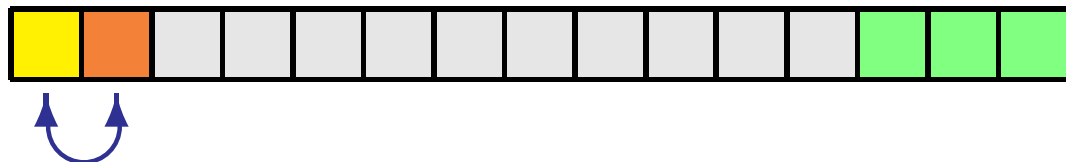
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



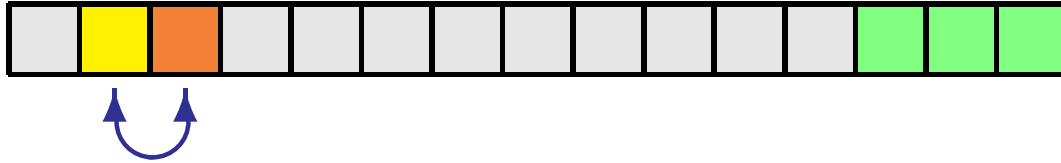
O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



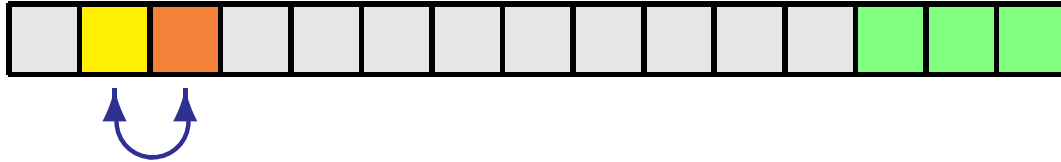
O szybsze sortowanie

Sortowanie bąbelkowe — BubbleSort:



O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort:**

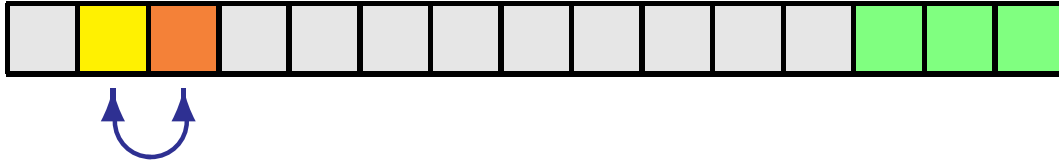


Sortowanie przez scalanie — **MergeSort:**

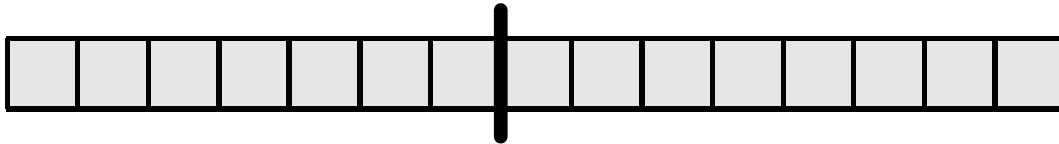


O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



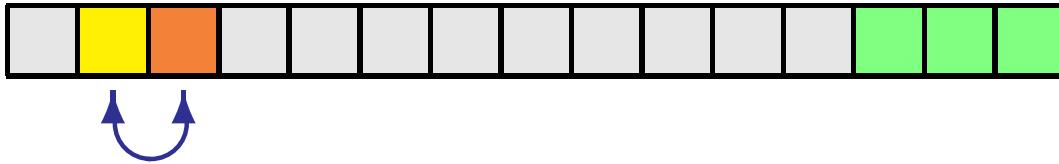
Sortowanie przez scalanie — **MergeSort**:



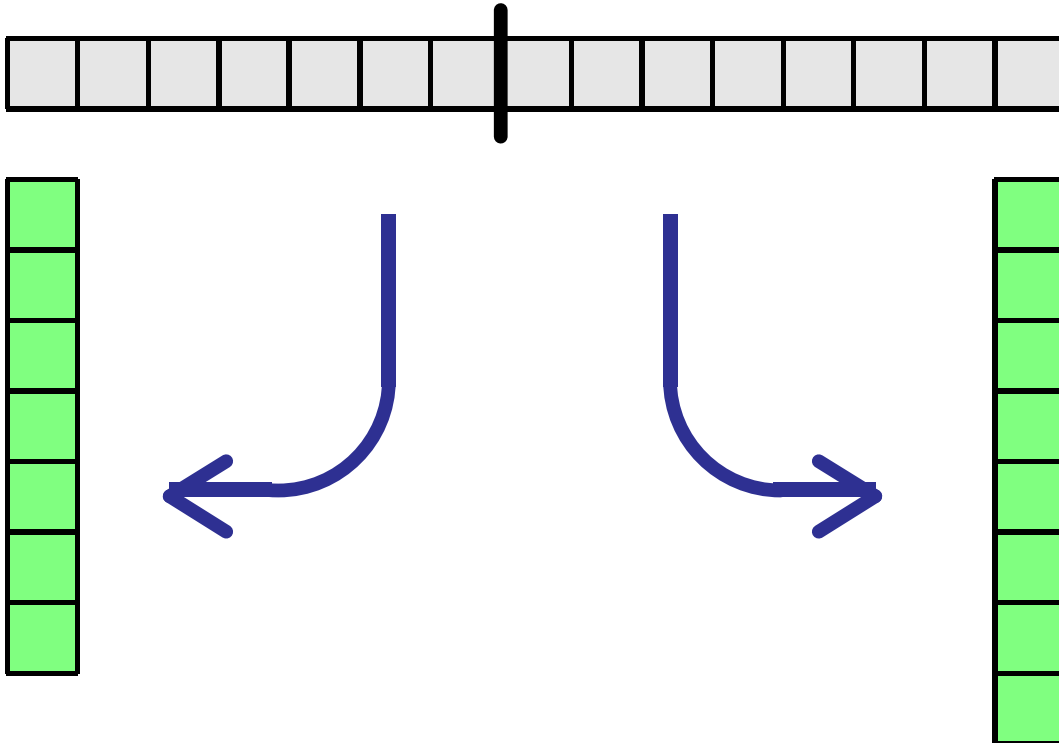
podzielić tablicę na połowy

O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



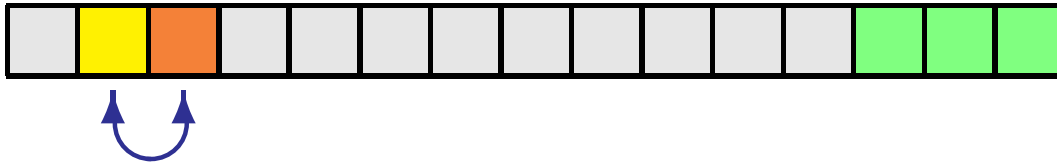
Sortowanie przez scalanie — **MergeSort**:



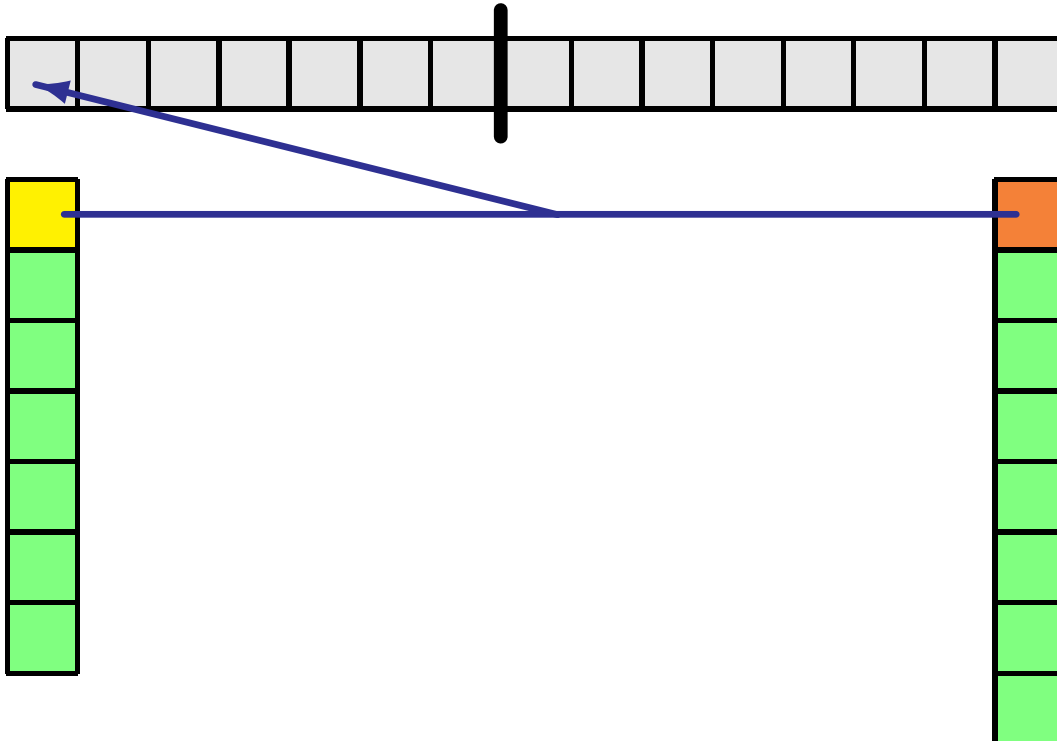
podzielić tablicę na połowy
posortować obie połowy

O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



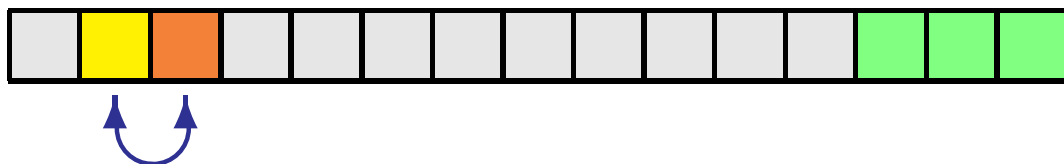
Sortowanie przez scalanie — **MergeSort**:



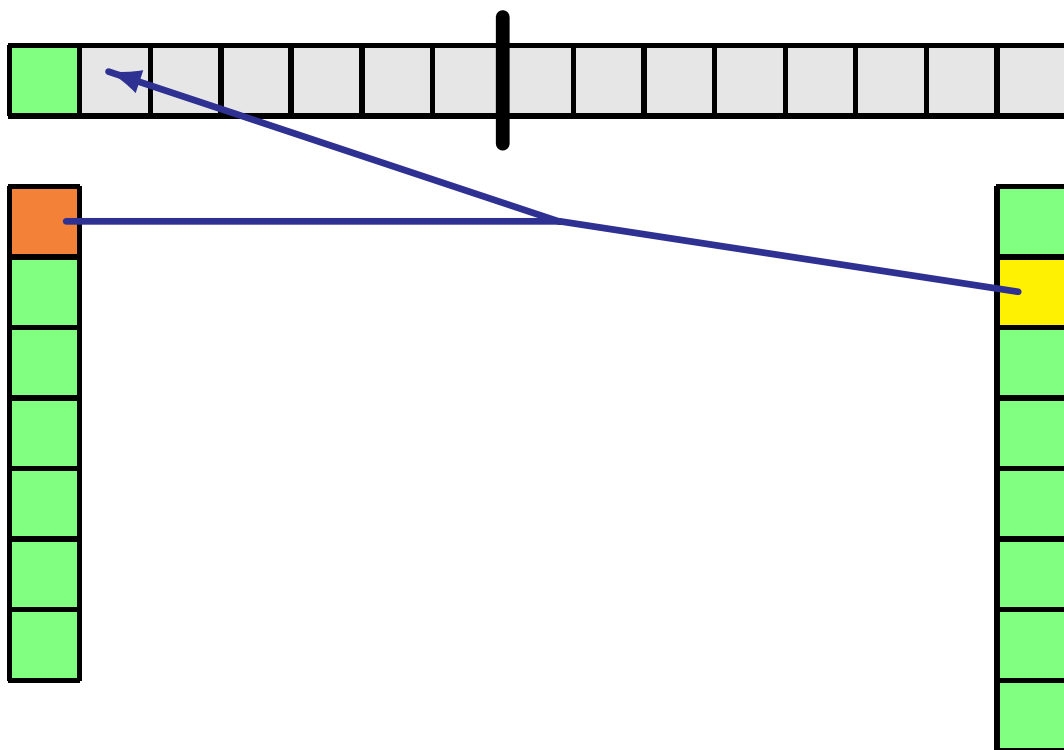
podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



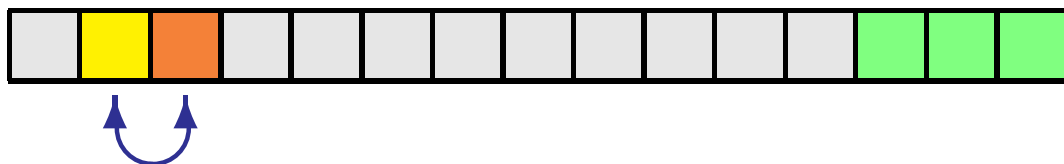
Sortowanie przez scalanie — **MergeSort**:



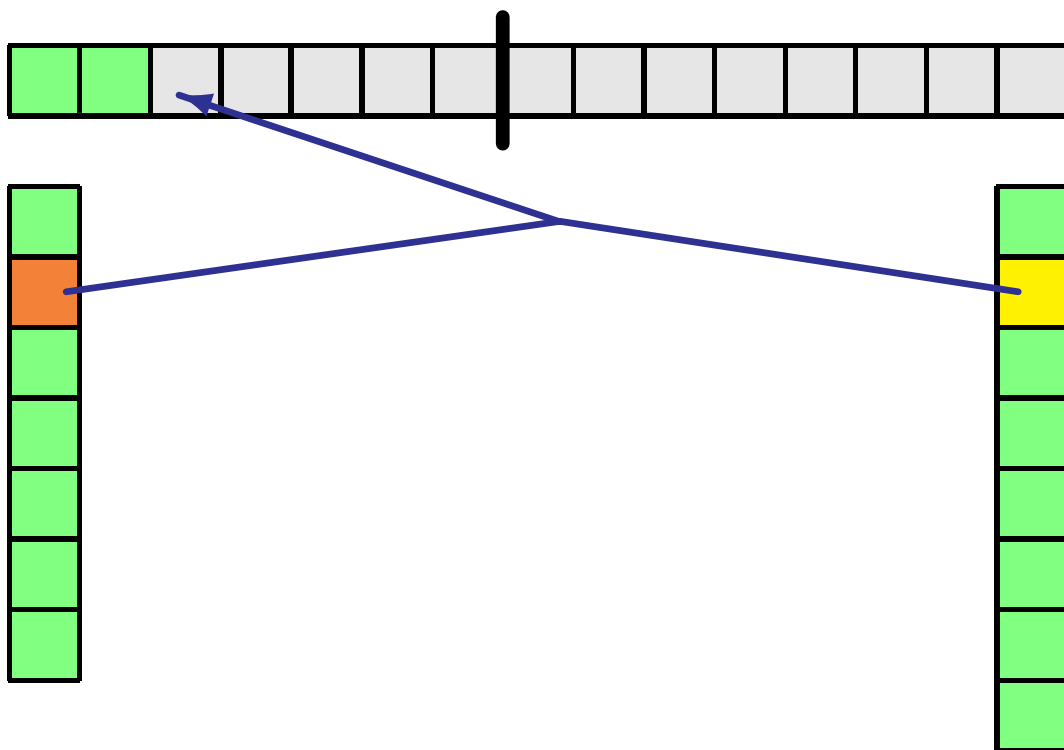
podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



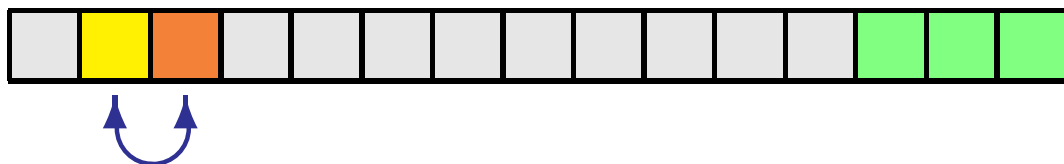
Sortowanie przez scalanie — **MergeSort**:



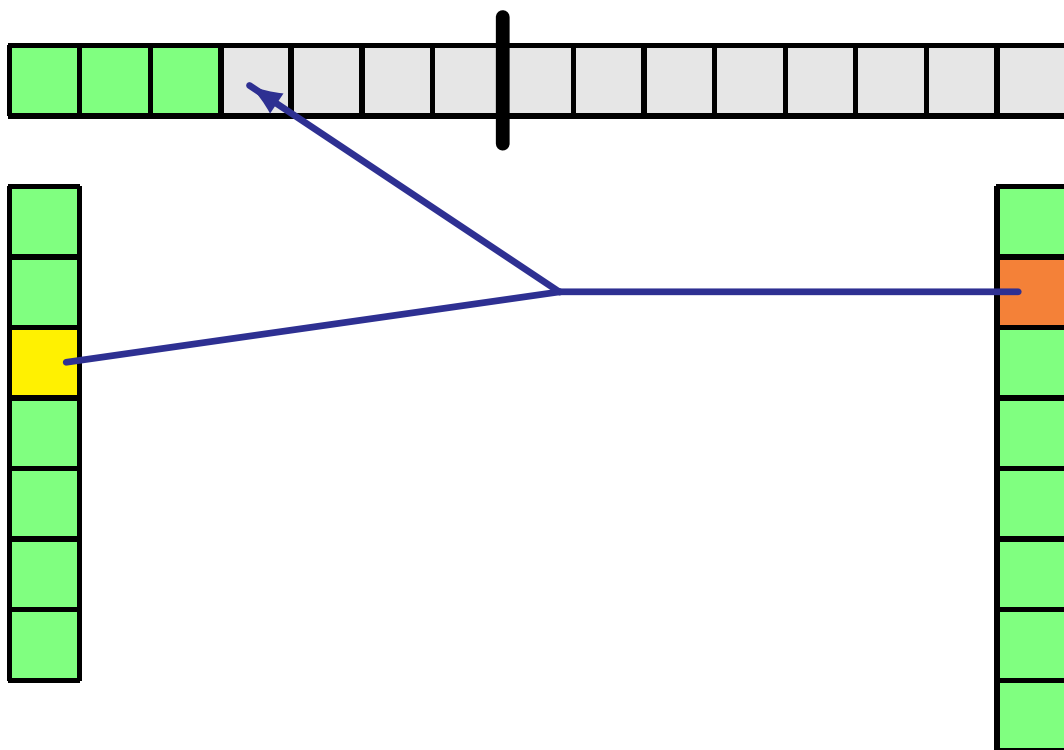
podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

O szybsze sortowanie

Sortowanie bąbelkowe — **BubbleSort**:



Sortowanie przez scalanie — **MergeSort**:



podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble						
Merge						

O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms				
Merge	1 ms	1 ms				

O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms			
Merge	1 ms	1 ms	1 ms			

O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms		
Merge	1 ms	1 ms	1 ms	3 ms		

O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms	1729 ms	
Merge	1 ms	1 ms	1 ms	3 ms	8 ms	

O szybsze sortowanie

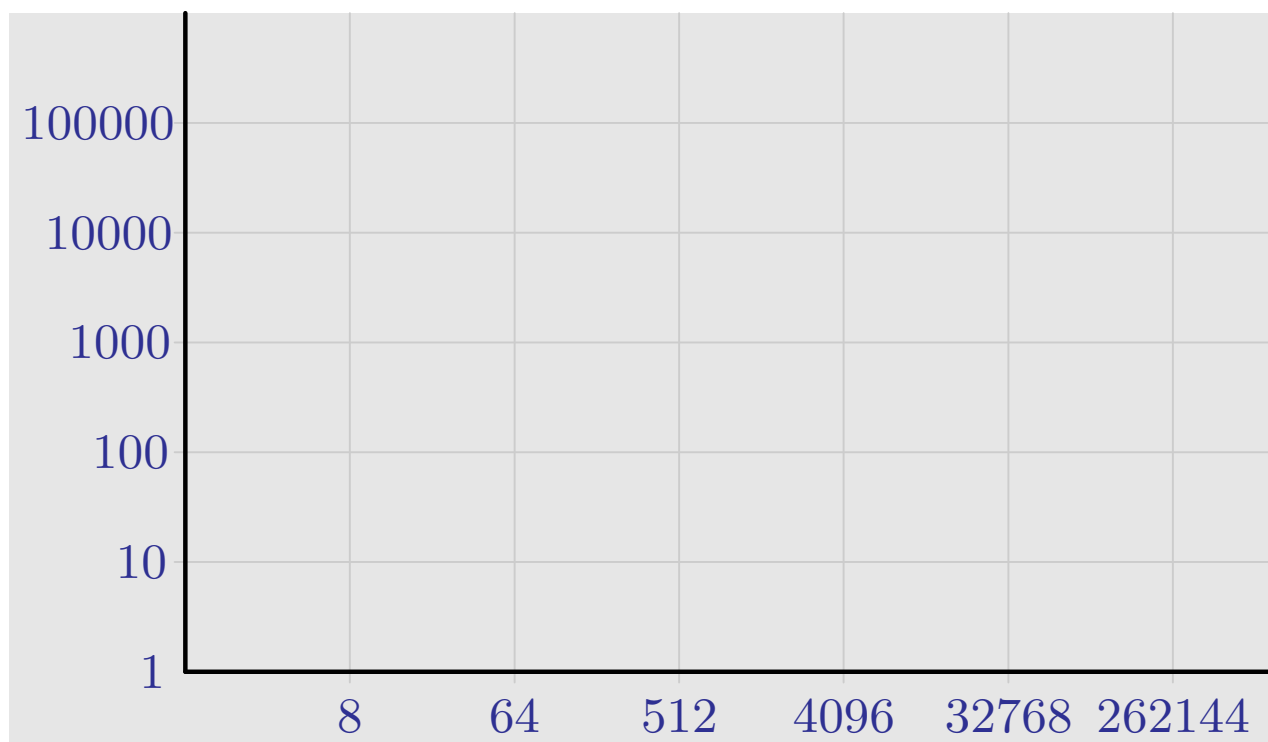
Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms	1729 ms	122 410 ms
Merge	1 ms	1 ms	1 ms	3 ms	8 ms	50 ms

O szybsze sortowanie

Porównanie czasów sortowania:

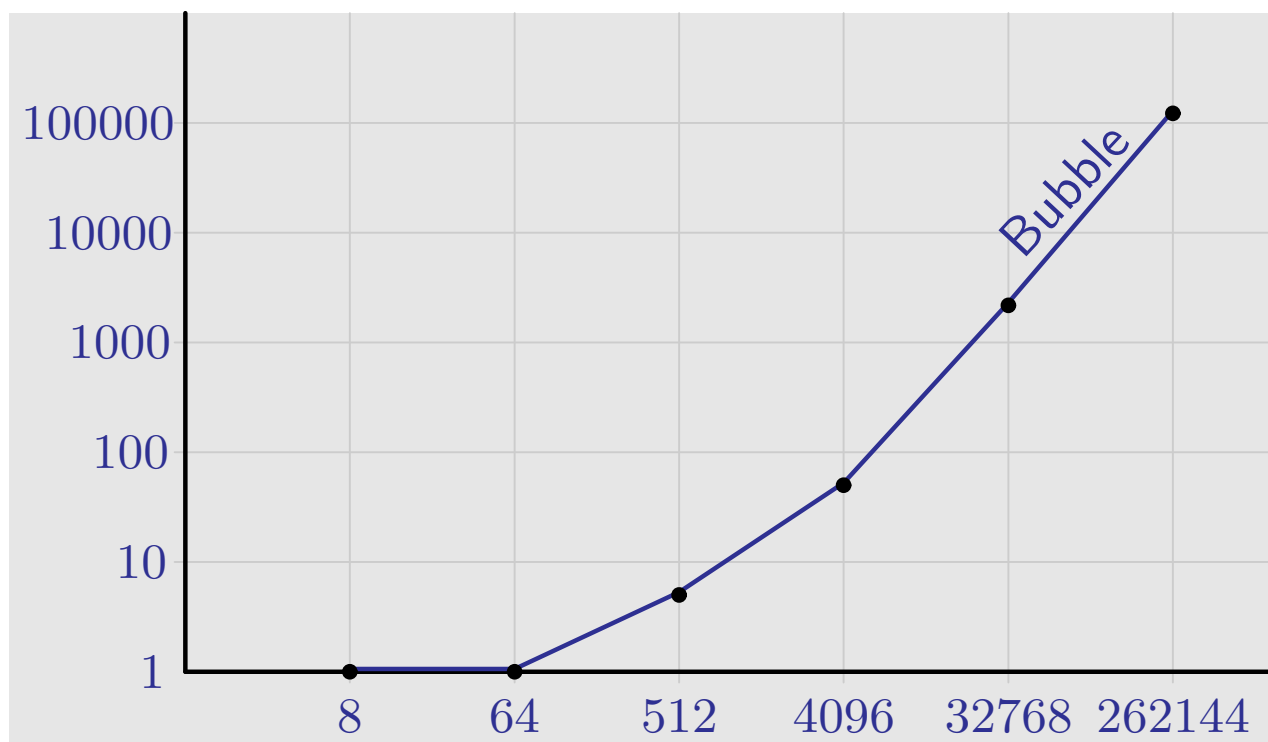
	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms	1729 ms	122 410 ms
Merge	1 ms	1 ms	1 ms	3 ms	8 ms	50 ms



O szybsze sortowanie

Porównanie czasów sortowania:

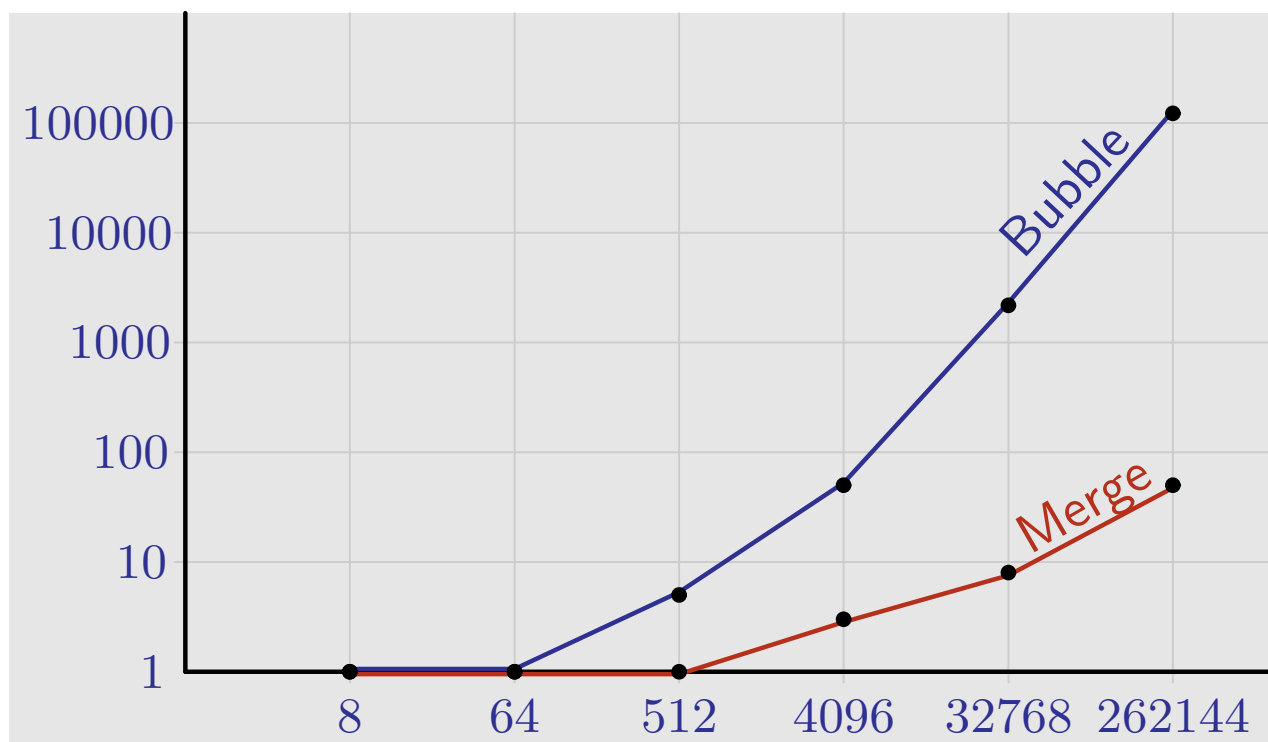
	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms	1729 ms	122 410 ms
Merge	1 ms	1 ms	1 ms	3 ms	8 ms	50 ms



O szybsze sortowanie

Porównanie czasów sortowania:

	wielkość sortowanej tablicy					
	$8^1 = 8$	$8^2 = 64$	$8^3 = 512$	$8^4 = 4096$	$8^5 = 32\,768$	$8^6 = 262\,144$
Bubble	1 ms	1 ms	5 ms	50 ms	1729 ms	122 410 ms
Merge	1 ms	1 ms	1 ms	3 ms	8 ms	50 ms



O szybsze sortowanie

MergeSort jest *znacznie* szybszy niż **BubbleSort**.
A czy może być jeszcze szybszy?

O szybsze sortowanie

MergeSort jest *znacznie* szybszy niż **BubbleSort**.
A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

O szybsze sortowanie

MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

O szybsze sortowanie

MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...

|
sort dług. n

O szybsze sortowanie

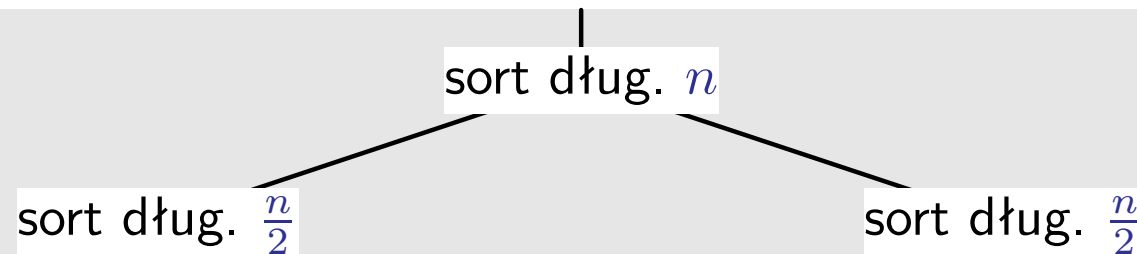
MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...



O szybsze sortowanie

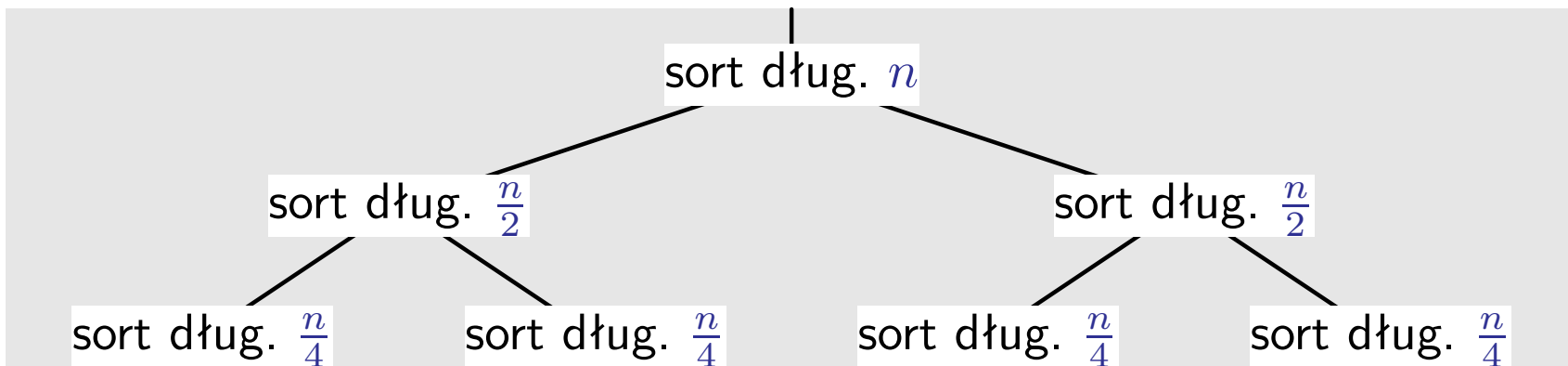
MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...



O szybsze sortowanie

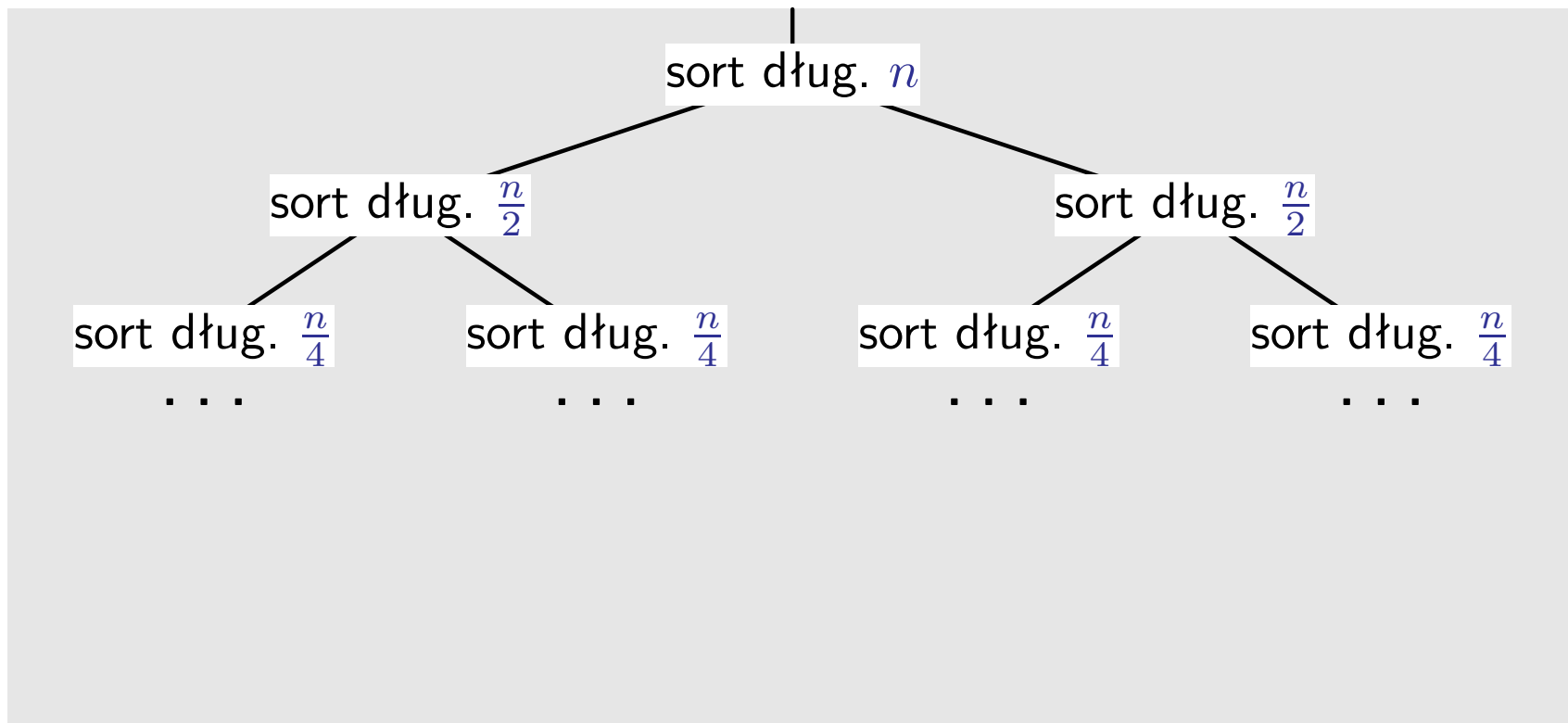
MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...



O szybsze sortowanie

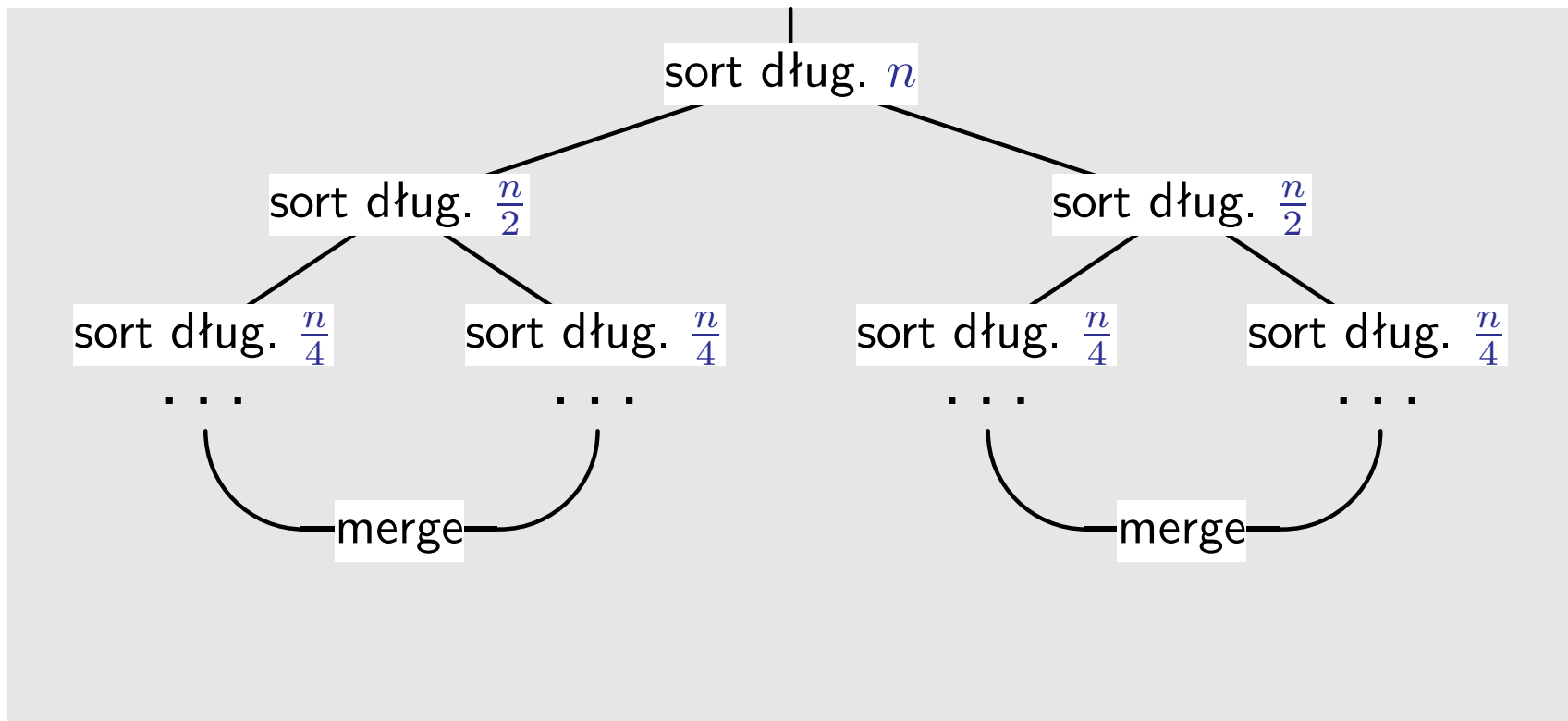
MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...



O szybsze sortowanie

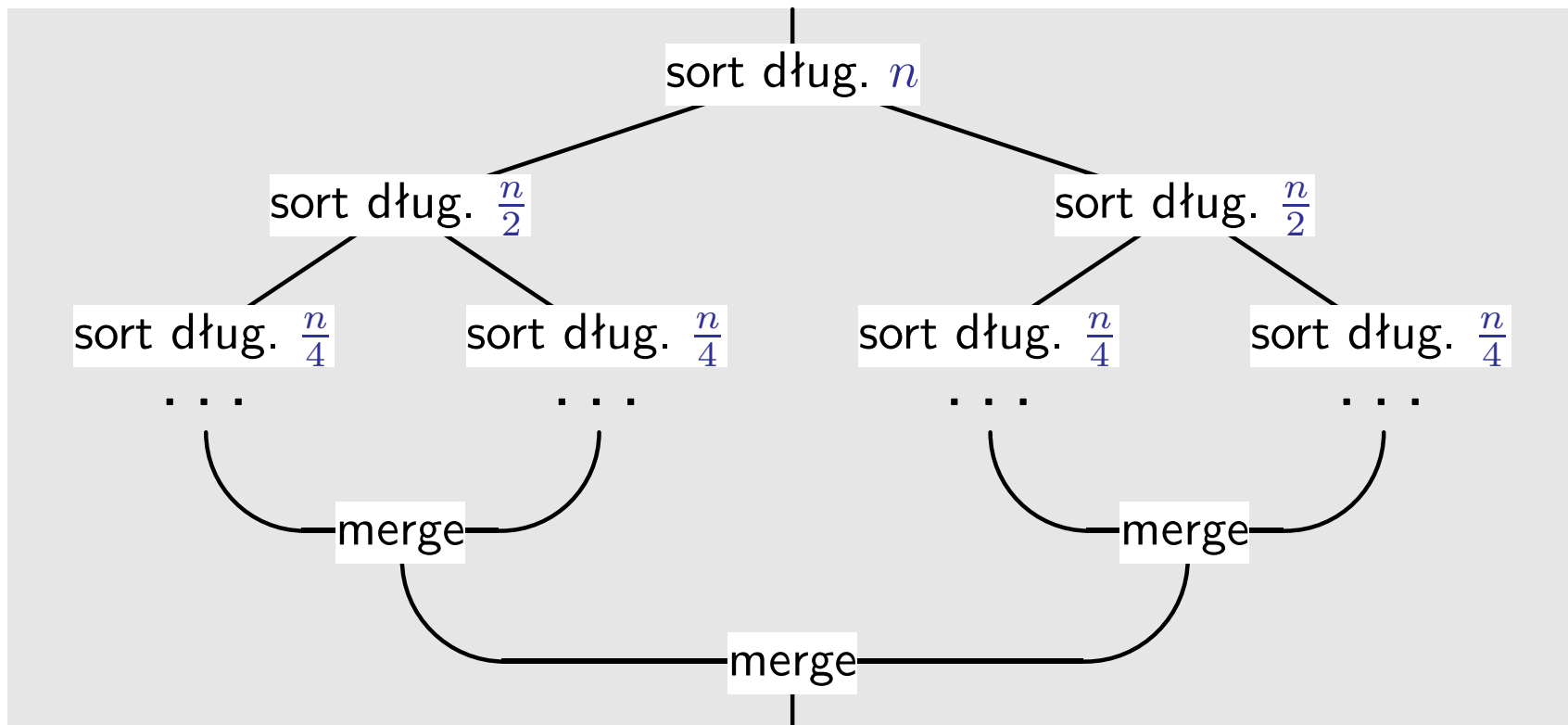
MergeSort jest *znacznie* szybszy niż **BubbleSort**.

A czy może być jeszcze szybszy?

podzielić tablicę na połowy
posortować obie połowy
połączyć przez wybieranie
mniejszego

nie ma zależności między połowami
można sortować **równocześnie**

Rozbić na współbieżne wątki...



O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```

0 szybsze sortowanie

```
void sort(tablicę) {
    podziel tablicę na lewą i prawą;
    sort(podtablicę lewą);
    sort(podtablicę prawą);
    merge(lewą i prawą);
}
```



```
void run( ) {
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
  
}
```

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
  
}
```

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start( ); t2.start( );  
    t1.join( ); t2.join( );  
}
```

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start( ); t2.start( );  
    t1.join( ); t2.join( );  
    merge(lewą i prawą);  
}
```

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start( ); t2.start( );  
    t1.join( ); t2.join( );  
    merge(lewą i prawą);  
}
```

Gdyby komputer miał bardzo dużo procesorów (rdzeni), to takie sortowanie powinno być *znacznie* szybsze niż **MergeSort**.

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start( ); t2.start( );  
    t1.join( ); t2.join( );  
    merge(lewą i prawą);  
}
```

Gdyby komputer miał bardzo dużo procesorów (rdzeni), to takie sortowanie powinno być *znacznie* szybsze niż **MergeSort**. Ale nie ma i czasy wychodzą gorsze.

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run( ) {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start( ); t2.start( );  
    t1.join( ); t2.join( );  
    merge(lewą i prawą);  
}
```

Gdyby komputer miał bardzo dużo procesorów (rdzeni), to takie sortowanie powinno być *znacznie* szybsze niż **MergeSort**. Ale nie ma i czasy wychodzą gorsze.

„Symulowany czas” — z poprawką na tę współbieżność — wychodzi lepszy.

O szybsze sortowanie

```
void sort(tablicę) {  
    podziel tablicę na lewą i prawą;  
    sort(podtablicę lewą);  
    sort(podtablicę prawą);  
    merge(lewą i prawą);  
}
```



```
void run() {  
    podziel tablicę na lewą i prawą;  
    Thread t1 = new Thread(new Sort(lewą));  
    Thread t2 = new Thread(new Sort(prawą));  
    t1.start(); t2.start();  
    t1.join(); t2.join();  
    merge(lewą i prawą);  
}
```

Gdyby komputer miał bardzo dużo procesorów (rdzeni), to takie sortowanie powinno być *znacznie* szybsze niż **MergeSort**. Ale nie ma i czasy wychodzą gorsze.

„Symulowany czas” — z poprawką na tą współbieżność — wychodzi lepszy.

ŚPIEW PRZYSZŁOŚCI...

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.



Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.



zysk na czasie
wynikający ze
współbieżności

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.



zysk na czasie
wynikający ze
współbieżności

przeciwko

strata czasu na
organizację wątków
dla współbieżności

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.



zysk na czasie
wynikający ze
współbieżności

przeciwko

strata czasu na
organizację wątków
dla współbieżności

Na razie na moim komputerze wygrywa strata czasu.

Na razie...

Najbardziej oszczędne sortowanie przez porównania n elementów wymaga ok. $n \cdot \log(n)$ porównań; więc przy przetwarzaniu **jednobieżnym** czas sortowania **nie może być niższy** niż $O(n \cdot \log(n))$.

Ale jeśli dysponujemy nieograniczonym przetwarzaniem **współbieżnym** i możemy dokonywać wielu porównań na raz, to wystarczy czas $O(\log(n))$, czyli znacznie mniej.



zysk na czasie
wynikający ze
współbieżności

przeciwko

strata czasu na
organizację wątków
dla współbieżności

Na razie na moim komputerze wygrywa strata czasu.
Czekam na bardziej współbieżne komputery...

Wyrywanie sobie drukarki z rąk

iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog13/W1_Druk.java

Wyrywanie sobie drukarki z rąk

iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog13/W1_Druk.java

Wątek 0 drukuje:

Ileż scen i obrazow z samej gry oblokow!
Bo kazda chmura inna: na przyklad jesienna

Wątek 1 drukuje:

Chmury nad nami rozpal w lune,
uderz nam w serca zlotym dzwonem,
otworz nam Polske, jak piorunem

Wyrywanie sobie drukarki z rąk

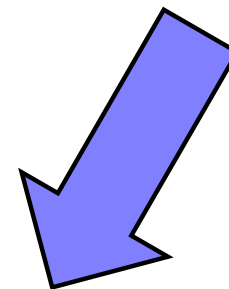
iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biIII/Prog13/W1_Druk.java

Wątek 0 drukuje:

Ileż scen i obrazow z samej gry oblokow!
Bo kazda chmura inna: na przykład jesienna

Wątek 1 drukuje:

Chmury nad nami rozpal w lune,
uderz nam w serca złotym dzwonem,
otworz nam Polske, jak piorunem



Cały wydruk:

ChmuryIleż scen i obrazow z nad nami samej gry oblo rozpal w lkow!
Bo kazda chmura inna: na przykład june,
uderz nam w seresieca złotym dzwonem,
otworz nam Polske, jak pnnaioru
nem

Wyrywanie sobie drukarki z rąk

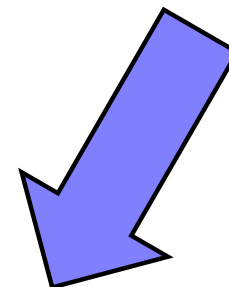
iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biIII/Prog13/W1_Druk.java

Wątek 0 drukuje:

Ileż scen i obrazow z samej gry oblokow!
Bo kazda chmura inna: na przyklad jesienna

Wątek 1 drukuje:

Chmury nad nami rozpal w lune,
uderz nam w serca zlotym dzwonem,
otworz nam Polske, jak piorunem



Cały wydruk:

ChmuryIleż scen i obrazow z nad nami samej gry oblo rozpal w lkow!
Bo kazda chmura inna: na przyklad june,
uderz nam w seresieca zlotym dzwonem,
otworz nam Polske, jak pnaioru
nem

Wyrywanie sobie drukarki z rąk

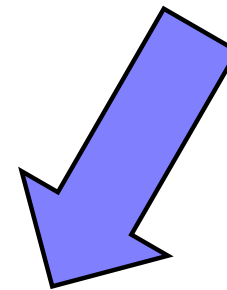
iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biIII/Prog13/W1_Druk.java

Wątek 0 drukuje:

Ileż scen i obrazow z samej gry oblokow!
Bo kazda chmura inna: na przyklad jesienna

Wątek 1 drukuje:

Chmury nad nami rozpal w lune,
uderz nam w serca zlotym dzwonem,
otworz nam Polske, jak piorunem



Cały wydruk:

ChmuryIleż scen i obrazow z nad nami samej gry oblo rozpal w lkow!
Bo kazda chmura inna: na przyklad june,
uderz nam w seresieca zlotym dzwonem,
otworz nam Polske, jak pnaioru
nem

Urządzenie zewnętrzne (np. drukarka) wymaga **ochrony dostępu**.

Jak synchronizować dostęp do drukarki?

```
class Zamek {}

class Drukowanie implements Runnable {
    private String doWydruku;

    public Drukowanie(String doWydruku) {
        this.doWydruku = doWydruku;
    }

    public void run() {
        synchronized(Main.zz) {
            for (int i=0; i<doWydruku.length(); i++)
                System.out.print(doWydruku.charAt(i));
            System.out.println();
        }
    }
}

class Main {
    public static Zamek zz = new Zamek();
    . . . . .
}
```

Jak synchronizować dostęp do drukarki?

```
class Zamek {} ← klasa „pusta”
```

```
class Drukowanie implements Runnable {
    private String doWydruku;

    public Drukowanie(String doWydruku) {
        this.doWydruku = doWydruku;
    }

    public void run() {
        synchronized(Main.zz) {
            for (int i=0; i<doWydruku.length(); i++)
                System.out.print(doWydruku.charAt(i));
            System.out.println();
        }
    }
}

class Main {
    public static Zamek zz = new Zamek();
    . . . . .
}
```

Jak synchronizować dostęp do drukarki?

```
class Zamek {} ← klasa „pusta”
```

```
class Drukowanie implements Runnable {  
    private String doWydruku;  
  
    public Drukowanie(String doWydruku) {  
        this.doWydruku = doWydruku;  
    }  
  
    public void run() {  
        synchronized(Main.zz) {  
            for (int i=0; i<doWydruku.length(); i++)  
                System.out.print(doWydruku.charAt(i));  
            System.out.println();  
        }  
    }  
}
```

```
class Main {  
    public static Zamek zz = new Zamek(); ← obiekt zamykający dostęp  
    . . . . .  
}
```

Jak synchronizować dostęp do drukarki?

```
class Zamek {}
```

← klasa „pusta”

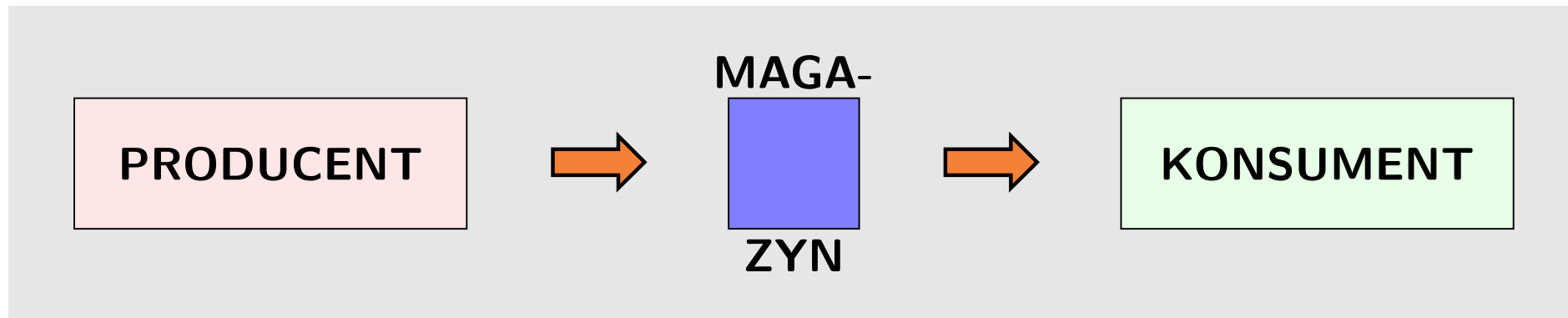
```
class Drukowanie implements Runnable {  
    private String doWydruku;  
  
    public Drukowanie(String doWydruku) {  
        this.doWydruku = doWydruku;  
    }  
  
    public void run() {  
        synchronized(Main.zz) {  
            for (int i=0; i<doWydruku.length(); i++)  
                System.out.print(doWydruku.charAt(i));  
            System.out.println();  
        }  
    }  
}
```

wykonać niepodzielnie

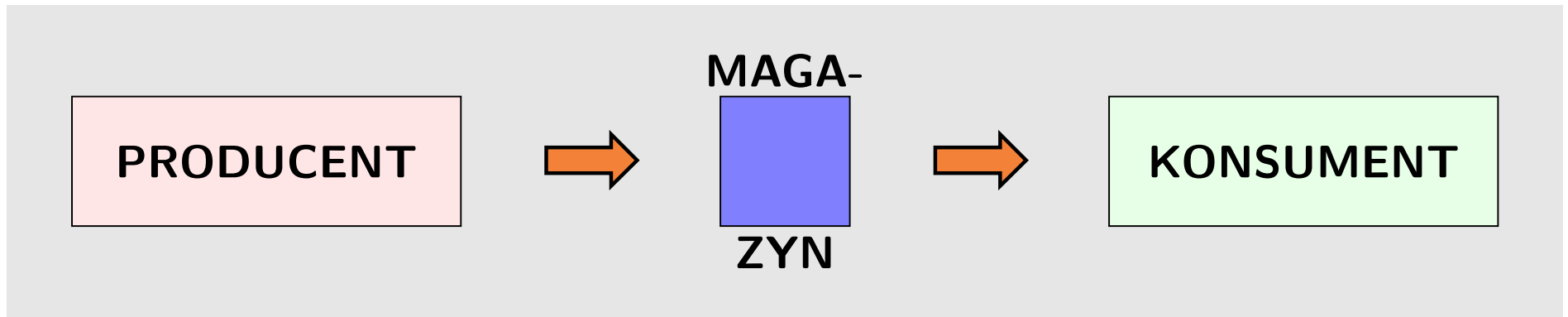
```
class Main {  
    public static Zamek zz = new Zamek();  
    . . . . .  
}
```

← obiekt zamykający dostęp

Producent i konsument

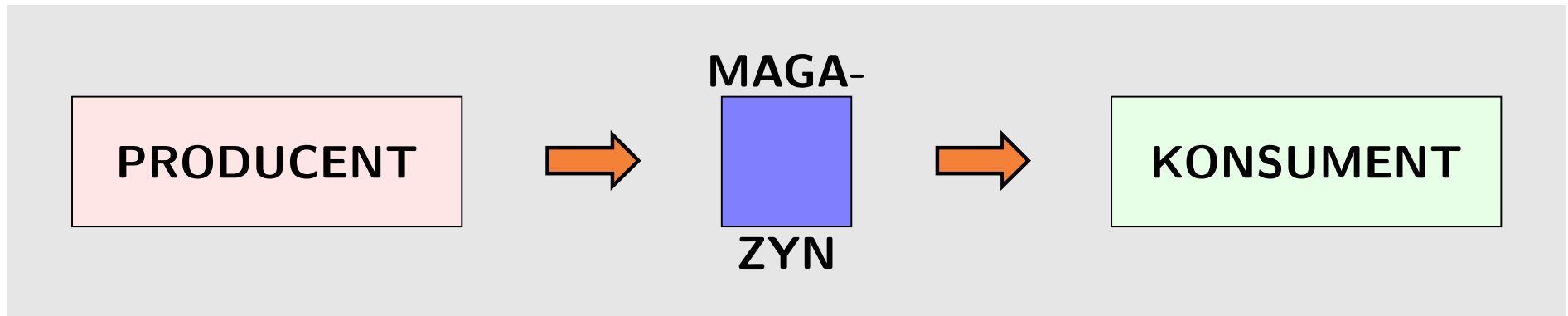


Producent i konsument



PRODUCENT produkuje jakiś produkt, odkłada go do magazynu i natychmiast przystępuje do produkcji następnego produktu.

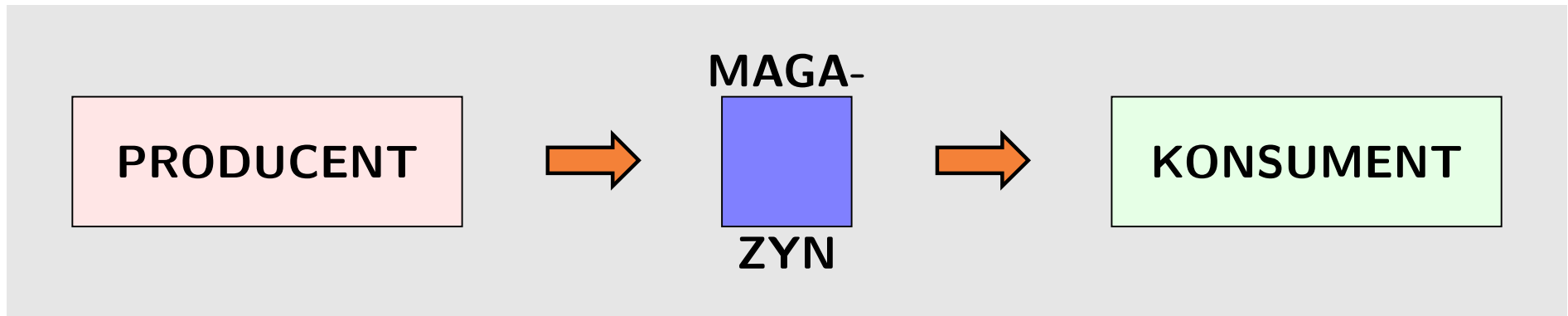
Producent i konsument



PRODUCENT produkuje jakiś produkt, odkłada go do magazynu i natychmiast przystępuje do produkcji następnego produktu.

Współbieżnie działający **KONSUMENT** zabiera produkt z magazynu, konsumuje go i sięga po następny.

Producent i konsument

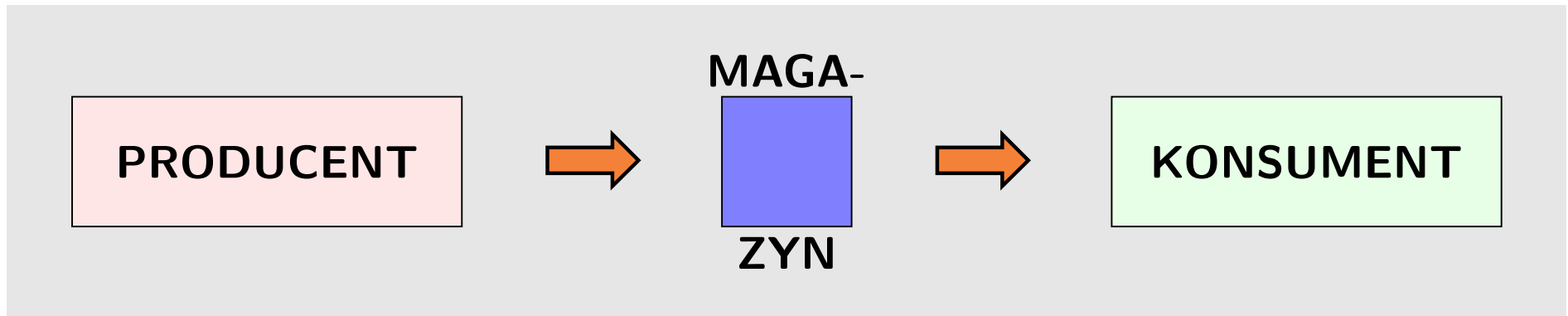


PRODUCENT produkuje jakiś produkt, odkłada go do magazynu i natychmiast przystępuje do produkcji następnego produktu.

Współbieżnie działający **KONSUMENT** zabiera produkt z magazynu, konsumuje go i sięga po następny.

Jeśli producent jest szybszy od konsumenta, to liczba przedmiotów w magazynie rośnie.

Producent i konsument



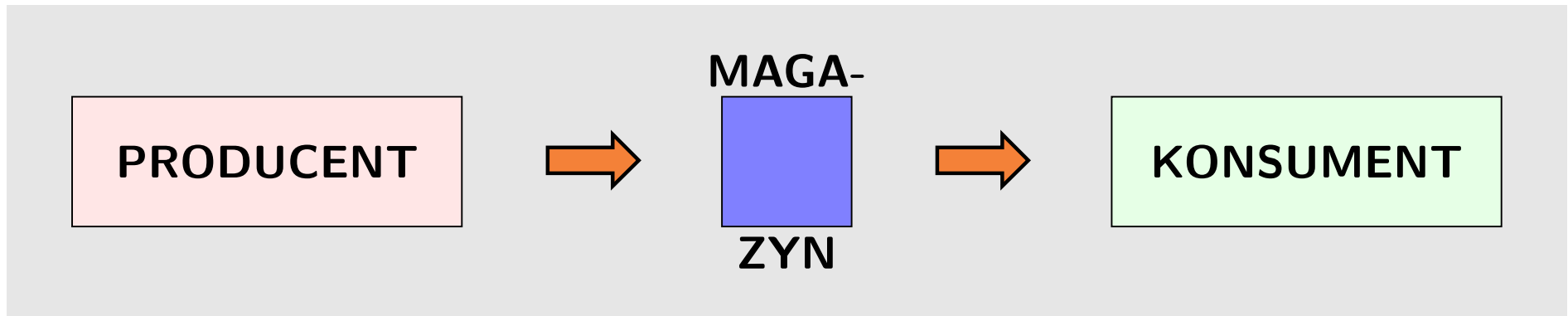
PRODUCENT produkuje jakiś produkt, odkłada go do magazynu i natychmiast przystępuje do produkcji następnego produktu.

Współbieżnie działający **KONSUMENT** zabiera produkt z magazynu, konsumuje go i sięga po następny.

Jeśli producent jest szybszy od konsumenta, to liczba przedmiotów w magazynie rośnie.

Jeśli konsument jest szybszy od producenta, to musi czekać na pojawienie się w magazynie kolejnego przedmiotu.

Producent i konsument



PRODUCENT produkuje jakiś produkt, odkłada go do magazynu i natychmiast przystępuje do produkcji następnego produktu.

Współbieżnie działający **KONSUMENT** zabiera produkt z magazynu, konsumuje go i sięga po następny.

Jeśli producent jest szybszy od konsumenta, to liczba przedmiotów w magazynie rośnie.

Jeśli konsument jest szybszy od producenta, to musi czekać na pojawienie się w magazynie kolejnego przedmiotu.

Jak zorganizować czekanie?

Producent i konsument

Producent

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włóż prod do magazynu;  
}  
ogłosić fajrant;
```

Producent i konsument

Producent

koniec dnia
roboczego



```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włóż prod do magazynu;  
}  
ogłosić fajrant;
```

Producent i konsument

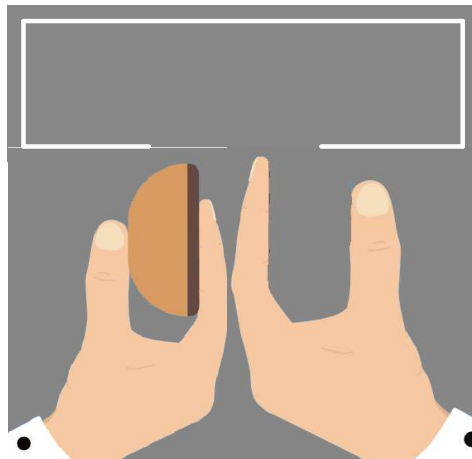
Producent

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włoż prod do magazynu;  
}  
ogłosić fajrant;
```

koniec dnia
roboczego

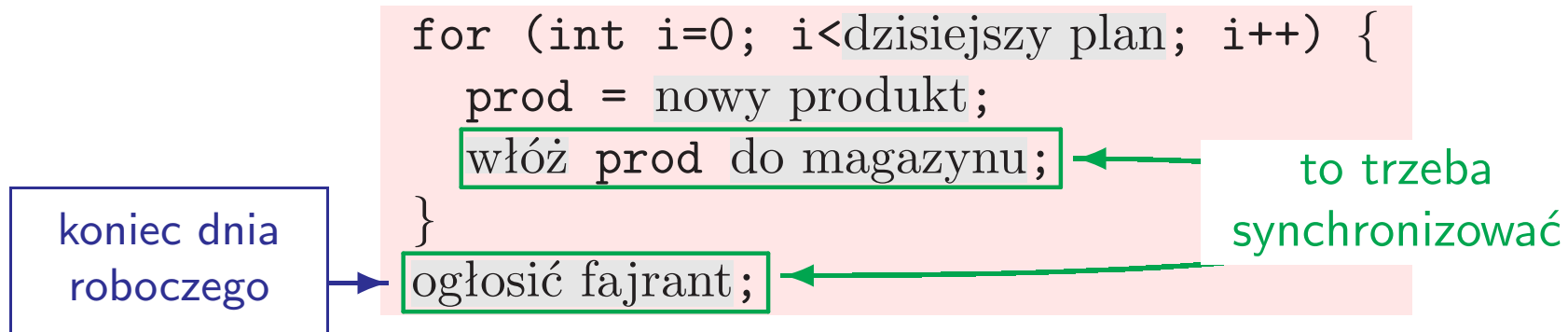
to trzeba
synchronizować

Wkładanie do magazynu musi być **synchronized**, bo w tej samej chwili konsument może chcieć wyjąć z magazynu.

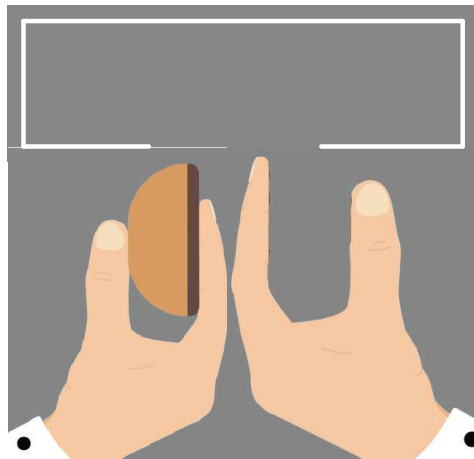


Producent i konsument

Producent



Wkładanie do magazynu musi być **synchronized**, bo w tej samej chwili konsument może chcieć wyjąć z magazynu.



Ogłoszenie fajrantu (końca dnia pracy) również musi być **synchronized**.

Producent i konsument

Konsument — czekanie aktywne (*Busy waiting*)

Producent i konsument

Konsument — czekanie aktywne (*Busy waiting*)

```
do {  
    if (magazyn jest pusty) {  
        nic nie rób;  jestNowy = false;  
    } else {  
        zabierz przedmiot z magazynu;  jestNowy = true;  
    }  
    if (jestNowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

Producent i konsument

Konsument — czekanie aktywne (*Busy waiting*)

```
do {  
    if (magazyn jest pusty) {  
        nic nie rób;  jestNowy = false;  
    } else {  
        zabierz przedmiot z magazynu;  jestNowy = true;  
    }  
    if (jestNowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

to trzeba
synchronizować

Producent i konsument

Konsument — czekanie aktywne (*Busy waiting*)

```
do {  
    if (magazyn jest pusty) {  
        nic nie rób;  jestNowy = false;  
    } else {  
        zabierz przedmiot z magazynu;  jestNowy = true;  
    }  
    if (jestNowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

to trzeba
synchronizować

Dopóki magazyn jest pusty, ale nie ma fajrantu — pętla kręci się jałowo:

```
do {  
    if (prawda) {  
        nic nie rób;  jestNowy = false;  
    }  
} while (true);
```

Producent i konsument

Konsument — czekanie aktywne (*Busy waiting*)

```
do {  
    if (magazyn jest pusty) {  
        nic nie rób;  jestNowy = false;  
    } else {  
        zabierz przedmiot z magazynu;  jestNowy = true;  
    }  
    if (jestNowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

to trzeba
synchronizować

Dopóki magazyn jest pusty, ale nie ma fajrantu — pętla kręci się jałowo:

```
do {  
    if (prawda) {  
        nic nie rób;  jestNowy = false;  
    }  
} while (true);
```

Procesor jest zajęty bezużytecznie, a konsument przeszkadza producentowi, synchronizując dostęp do magazynu.

Producent i konsument

Aktywne czekanie,

czyli pętla z ciągłym badaniem dostępności przedmiotu,
to nienajlepsze rozwiązanie.

Producent i konsument

Aktywne czekanie,

czyli pętla z ciągłym badaniem dostępności przedmiotu,
to nienajlepsze rozwiązanie.

Na szczęście istnieją **inne** narzędzia synchronizacji wątków.

Producent i konsument

Aktywne czekanie,

czyli pętla z ciągłym badaniem dostępności przedmiotu, to nienajlepsze rozwiązanie.

Na szczęście istnieją **inne** narzędzia synchronizacji wątków.

Konsument powinien **biernie czekać** (nie zajmując mocy obliczeniowych) aż zostanie **powiadomiony** o dostępności przedmiotu.

Wątkowe przepisy drogowe

`Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund

Wątkowe przepisy drogowe



`Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund

Wątkowe przepisy drogowe



`Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund

`t.join();` — *ten* wątek czeka, aż `t` zakończy działanie

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż `t` zakończy działanie

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż `t` zakończy działanie
- `synchronized` — użyte na raz tylko przez jeden wątek

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż `t` zakończy działanie
- ✓ `synchronized` — użyte na raz tylko przez jeden wątek

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż `t` zakończy działanie
- ✓ `synchronized` — użyte na raz tylko przez jeden wątek
- `wait();` — *ten* wątek czeka na obudzenie

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka `ms` milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż `t` zakończy działanie
- ✓ `synchronized` — użyte na raz tylko przez jeden wątek
- `wait();` — *ten* wątek czeka na obudzenie
- `notify();` — budzi **jeden** (losowy) ze śpiących wątków

Wątkowe przepisy drogowe

- ✓ `Thread.sleep(ms);` — *ten* wątek czeka **ms** milisekund
- ✓ `t.join();` — *ten* wątek czeka, aż **t** zakończy działanie
- ✓ `synchronized` — użyte na raz tylko przez jeden wątek
- `wait();` — *ten* wątek czeka na obudzenie
- `notify();` — budzi **jeden** (losowy) ze śpiących wątków
- `notifyAll();` — budzi **wszystkie** śpiące wątki

Producent i konsument

Czekanie bierne (*wait-notify*)

Producent i konsument

Czekanie bierne (*wait-notify*)

PRODUCENT:

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włóż prod do magazynu;  
    obiekt.notify();  
}
```

Producent i konsument

Czekanie bierne (*wait-notify*)

PRODUCENT:

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włóż prod do magazynu;  
    obiekt.notify();  
}
```

KONSUMENT:

```
do {  
    if (magazyn jest pusty) {  
        obiekt.wait(); nowy = false;  
    } else {  
        zabierz przedmiot z magazynu; nowy = true;  
    }  
    if (nowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

Producent i konsument

Czekanie bierne (*wait-notify*)

PRODUCENT:

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włóż prod do magazynu;  
    obiekt.notify();  
}
```

obudzić śpiący wątek



KONSUMENT:

```
do {  
    if (magazyn jest pusty) {  
        obiekt.wait(); nowy = false;  
    } else {  
        zabierz przedmiot z magazynu; nowy = true;  
    }  
    if (nowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

zasnąć



Producent i konsument

Czekanie bierne (*wait-notify*)

PRODUCENT:

```
for (int i=0; i<dzisiejszy plan; i++) {  
    prod = nowy produkt;  
    włoż prod do magazynu;  
    obiekt.notify();  
}
```

trzeba synchronizować

obudzić śpiący wątek

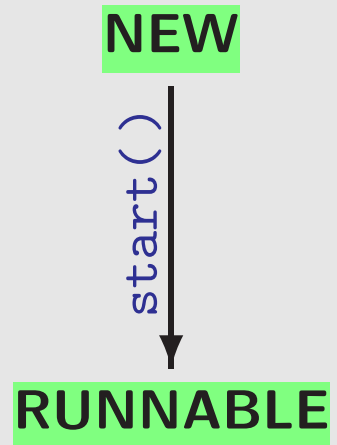
KONSUMENT:

```
do {  
    if (magazyn jest pusty) {  
        obiekt.wait(); nowy = false;  
    } else {  
        zabierz przedmiot z magazynu; nowy = true;  
    }  
    if (nowy) skonsumuj;  
} while (! (fajrant && magazyn jest pusty))
```

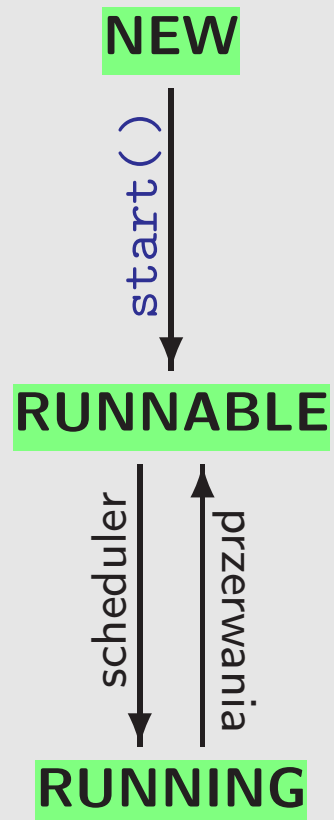
zasnąć

trzeba
synchronizować

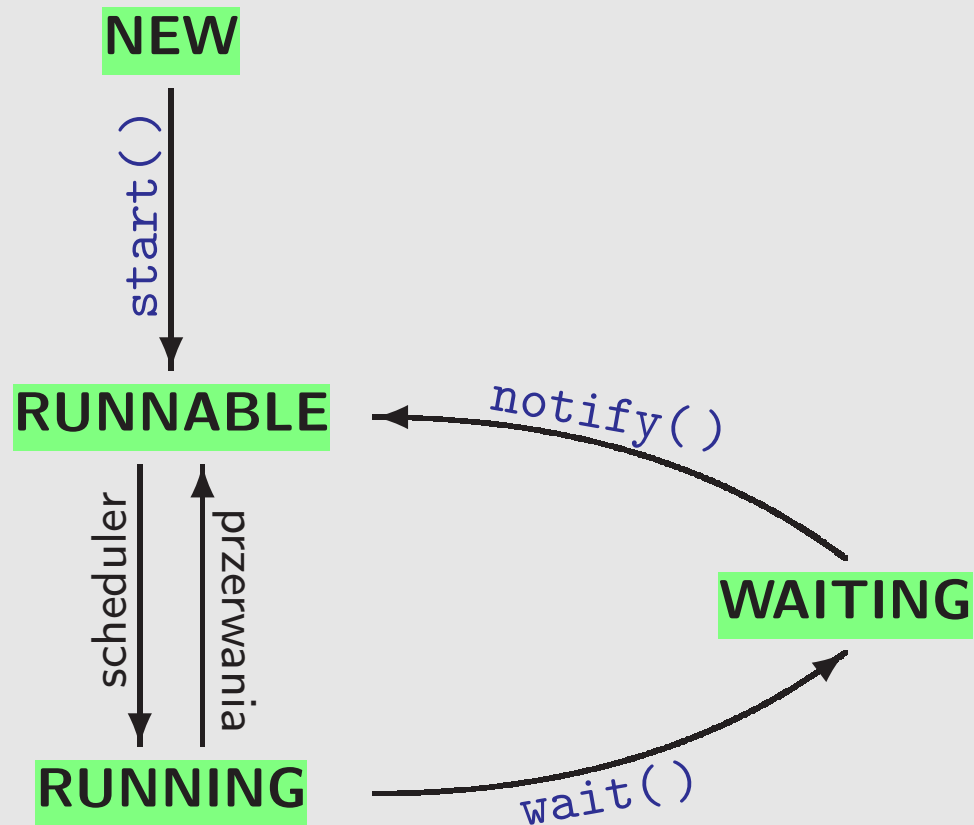
Stany wątków



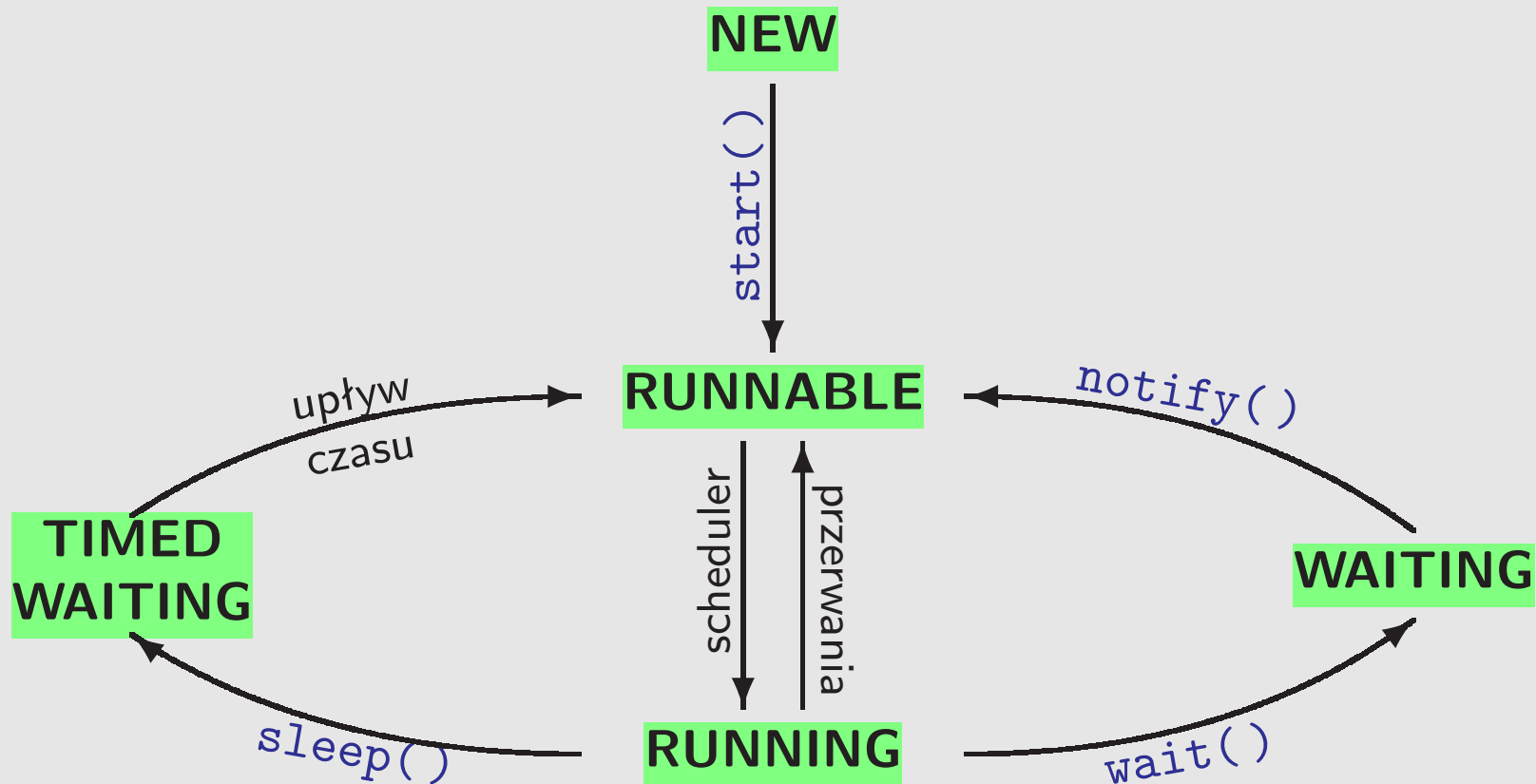
Stany wątków



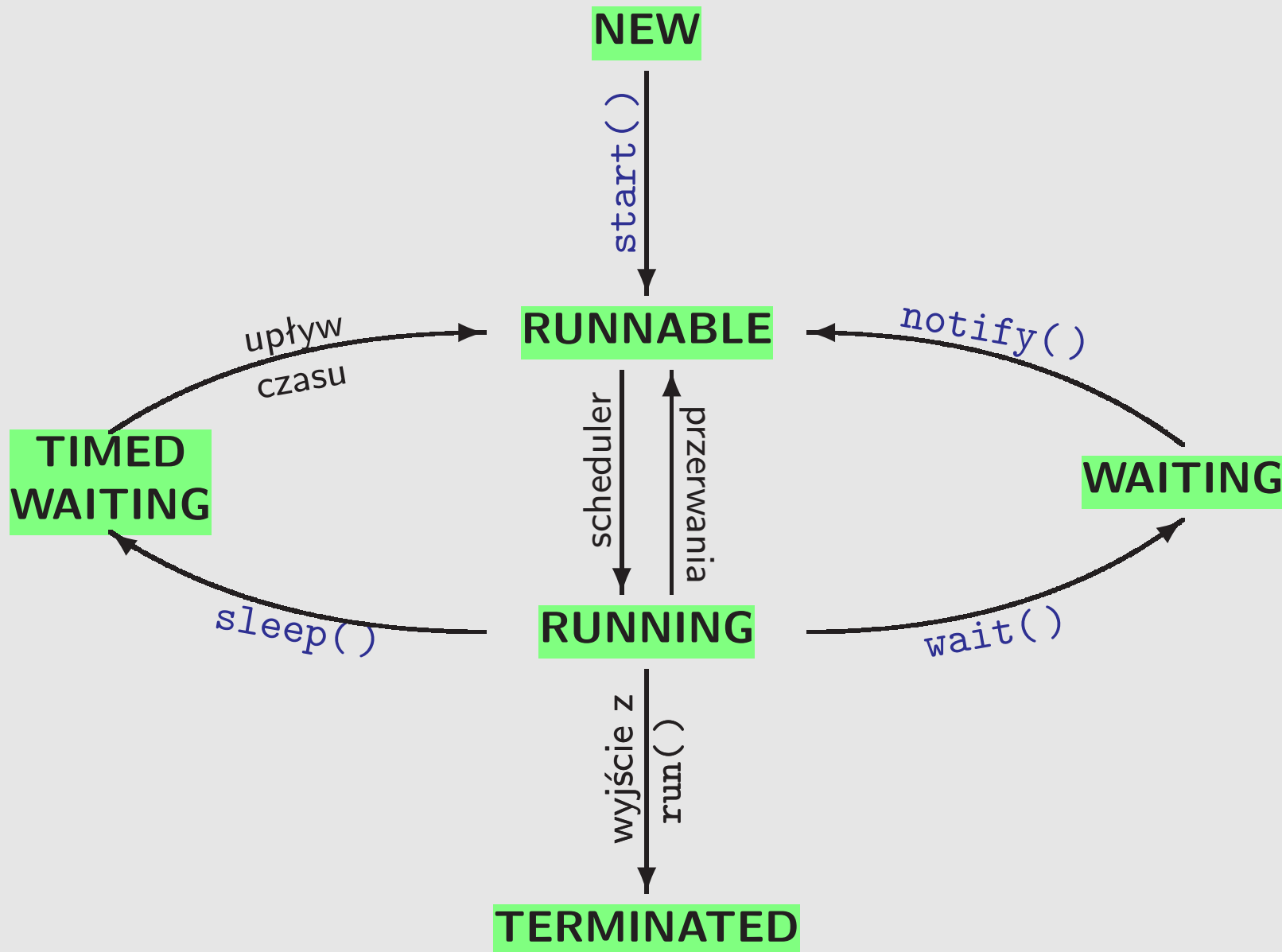
Stany wątków



Stany wątków



Stany wątków



Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.

Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.
- `t.start();`
— uruchamia wątek, czyli zmienia jego stan na **RUNNABLE**.

Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.
- `t.start();`
— uruchamia wątek, czyli zmienia jego stan na **RUNNABLE**. *Scheduler* (dyspozytor?) w systemie operacyjnym zmienia jego stan na **RUNNING**, kiedy wątek naprawdę działa.

Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.
- `t.start();`
— uruchamia wątek, czyli zmienia jego stan na **RUNNABLE**. *Scheduler* (dyspozytor?) w systemie operacyjnym zmienia jego stan na **RUNNING**, kiedy wątek naprawdę działa. Co jakiś czas *scheduler* przerywa jego działanie (znowu stan **RUNNABLE**), żeby dać szansę innym wątkom.

Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.
- `t.start();`
— uruchamia wątek, czyli zmienia jego stan na **RUNNABLE**. *Scheduler* (dyspozytor?) w systemie operacyjnym zmienia jego stan na **RUNNING**, kiedy wątek naprawdę działa. Co jakiś czas *scheduler* przerywa jego działanie (znowu stan **RUNNABLE**), żeby dać szansę innym wątkom.
- `obiekt.wait();`
— zawiesza wątek, czyli zmienia jego stan na **WAITING**.

Stany wątków

- `Thread t = new Thread(new Wątek());`
— tworzy nowy wątek, jego stan jest **NEW**.
- `t.start();`
— uruchamia wątek, czyli zmienia jego stan na **RUNNABLE**. *Scheduler* (dyspozytor?) w systemie operacyjnym zmienia jego stan na **RUNNING**, kiedy wątek naprawdę działa. Co jakiś czas *scheduler* przerywa jego działanie (znowu stan **RUNNABLE**), żeby dać szansę innym wątkom.
- `obiekt.wait();`
— zawiesza wątek, czyli zmienia jego stan na **WAITING**. Wątek zwalnia procesor i jest uśpiony — biernie czeka na obudzenie związane z tym samym obiektem.

Stany wątków

- `obiekt.notify();`
— budzenie wątku.

`obiekt.notifyAll();`

Stany wątków

- `obiekt.notify()`; `obiekt.notifyAll()`;
— budzenie wątku. `obiekt.notify()` budzi **losowo wybrany** wątek czekający na tym obiekcie; `obiekt.notifyAll()` budzi **wszystkie** wątki czekające na tym obiekcie.

Stany wątków

- `obiekt.notify()`; `obiekt.notifyAll()`;
— budzenie wątku. `obiekt.notify()` budzi **losowo wybrany** wątek czekający na tym obiekcie; `obiekt.notifyAll()` budzi **wszystkie** wątki czekające na tym obiekcie. Obudzony wątek staje się **RUNNABLE**.

Stany wątków

- `obiekt.notify()`; `obiekt.notifyAll()`;
— budzenie wątku. `obiekt.notify()` budzi **losowo wybrany** wątek czekający na tym obiekcie; `obiekt.notifyAll()` budzi **wszystkie** wątki czekające na tym obiekcie. Obudzony wątek staje się **RUNNABLE**.
- `Thread.sleep(milis)`;
— usypia wątek na określony czas (w milisekundach).

Stany wątków

- `obiekt.notify()`; `obiekt.notifyAll()`;
— budzenie wątku. `obiekt.notify()` budzi **losowo wybrany** wątek czekający na tym obiekcie; `obiekt.notifyAll()` budzi **wszystkie** wątki czekające na tym obiekcie. Obudzony wątek staje się **RUNNABLE**.
- `Thread.sleep(milis)`;
— usypia wątek na określony czas (w milisekundach). Wątek przechodzi w stan **TIMED WAITING**. Po upływie tego czasu staje się **RUNNABLE**.

Stany wątków

- `obiekt.notify()`; `obiekt.notifyAll()`;
— budzenie wątku. `obiekt.notify()` budzi **losowo wybrany** wątek czekający na tym obiekcie; `obiekt.notifyAll()` budzi **wszystkie** wątki czekające na tym obiekcie. Obudzony wątek staje się **RUNNABLE**.
- `Thread.sleep(milis)`;
— usypia wątek na określony czas (w milisekundach). Wątek przechodzi w stan **TIMED WAITING**. Po upływie tego czasu staje się **RUNNABLE**.
- koniec metody `run` wątku
(przez dojście do końca ciała, albo przez wykonanie `return;`)
— zakończenie wątku: stan **TERMINATED**.

wait-notify

Komendy

- uśpienia `....wait();`
- obudzenia `....notify();` i `....notifyAll();`

można wpisać poprawnie **tylko** do wnętrza bloku `synchronized`.

wait-notify

Komendy

- uśpienia `....wait();`
- obudzenia `....notify();` i `....notifyAll();`

można wpisać poprawnie **tylko** do wnętrza bloku `synchronized`.

Nie można tego zrobić tam, gdzie jakiś inny wątek mógłby jednocześnie coś zmieniać. Wygląda to np. tak:

```
synchronized(this) {  
    try { this.wait(); }  
    catch(InterruptedException e) {}  
}
```

wait-notify

Komendy

- uśpienia `....wait();`
- obudzenia `....notify();` i `....notifyAll();`

można wpisać poprawnie **tylko** do wnętrza bloku `synchronized`.

Nie można tego zrobić tam, gdzie jakiś inny wątek mógłby jednocześnie coś zmieniać. Wygląda to np. tak:

```
synchronized(this) {  
    try { this.wait(); }  
    catch(InterruptedException e) {}  
}
```

Sygnal błędu — w czasie działania, wyjątek:

Exception in thread "...": java.lang.`IllegalMonitorStateException`