# Algorytmy i struktury danych – W04

Teoria złożoności cz. 4/4

Komparator

Proste algorytmy sortowania

# Zawartość

- Złożoność  cz. 3 - analiza złożoności:
  - Rodzaje złożoności:
    - Pesymistyczna
    - Średnia
- Sortowanie – definicja
- Interfejsy porównujące obiekty - komparatory:
  - `Comparable` (porządek naturalny)
  - `Comparator`
- Komparator odwrotny
- Komparator złożony
- Interfejs `RandomAccess`
- Proste algorytmy sortowania  - O($n^2$)
  - Sortowanie przez wstawianie
  - Sortowanie przez wybór
  - Sortowanie bąbelkowe

# Notacja

- Wejściem *I* dla algorytmu są dane takie jak liczby, ciąg znaków, rekordy na który algorytm wykonuje operacje przetwarzające
- Niech $F_n$ oznacza wszystkich wejść do algorytmu o rozmiarze *n*.
- Dla $I \in F_n$ niech $\tau(I)$ oznacza liczbę elementarnych kroków, które wykonuje algorytm dla wejścia *I*
- $\tau$ – czyta się *tau*

# Złożoność pesymistyczna

- *Złożoność pesymistyczna* (najgorszego przypadku, ang. *Worst-case complexity*) algorytmu jest funkcją *W(n)* taką, że *W(n)* równa się maksymalnej wartości $\tau(I)$, gdzie *I* przyjmuje wartości spośród wszystkich możliwych wejść o rozmiarze *n*. Czyli,

$$W(n) = \max\left\{\tau(I) \middle| I \in \mathsf{F}_n\right\}$$

# Złożoność pesymistyczna - przykład

Posortuj *n* elementów poprzez wygenerowanie wszystkich permutacji elementów, sprawdzając, czy kolejne permutacja jest posortowana. Jeśli jest posortowana, zakończ generowanie permutacji i cały algorytm.

Założenia:

- następna generacja jest wykonywana w jednym kroku elementarnym
- Sprawdzanie, czy permutacja jest rozwiązaniem wykonywane jest w jednym kroku

Przy tych założeniach:

$$W(n) = 2n! = O(n!)$$

# Złożoność średnia/oczekiwana

Niech *p(I)* będzie prawdopodobieństwem, że wejście *I* będzie daną wejściową

*Średnia/oczekiwana złożoność* (ang. *average (expected) complexity) A*(*n*) algorytmu ze skończonym zbiorem wejść $F_n$ jest definiowana jako:

$$A(n) = \sum_{I \in F_n} \tau(I) p(I) = E[\tau]$$

# Formuła II

Niech $p_i$ oznacza prawdopodobieństwo, że algorytm wykona *dokładnie i* kroków elementarnych; czyli $p_i=P(\tau=i)$.
Wówczas:

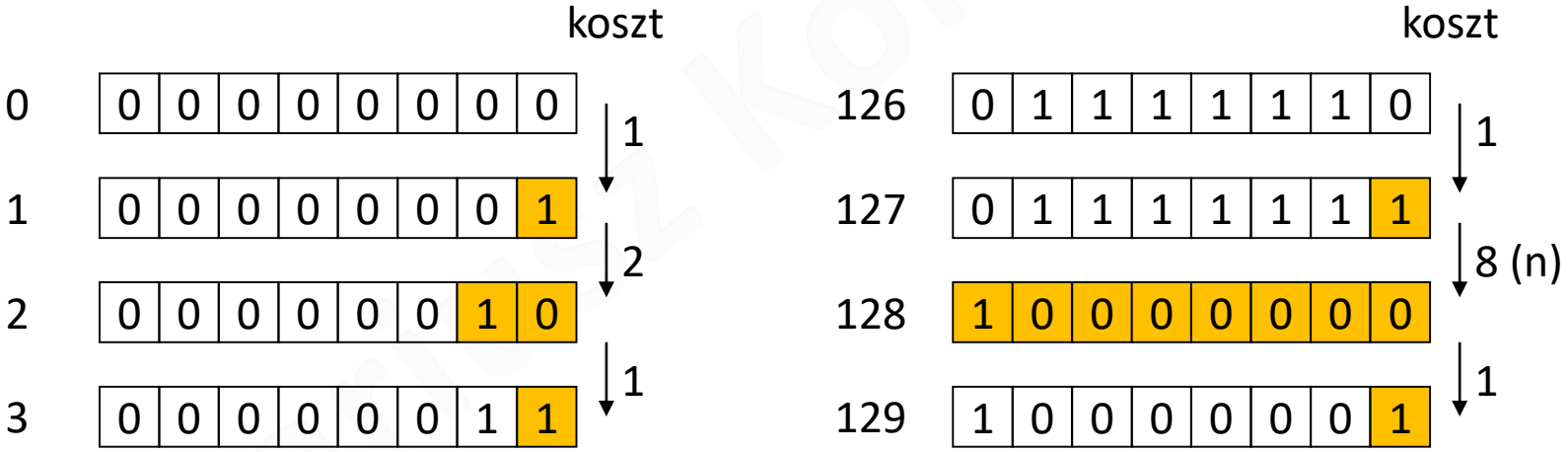
$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} i p_i$$

# Koszt zamortyzowany

W analizie kosztu zamortyzowanego pokazuje się, że dla wszystkich $n$, ciąg $n$ operacji zajmuje pesymistycznie czas całkowity $T(n)$. W pesymistyczny przypadku **średni koszt**, lub **zamortyzowany koszt**, dla jednej operacji jest zatem $T(n)/n$. Należy zauważyć, że koszt zamortyzowany odnosi się do każdej operacji, również wtedy, gdy w ciągu operacji były one różnych typów.

# Licznik *n*-bitowy

- Mamy *n*-bitowy binarny licznik, w którym wartość możemy tylko zwiększać o jeden (*n* – rozmiar danych wejściowych).
- Operacja elementarna (koszt) – zmiana **jednego** bitu (z 0 na 1 oraz z 1 na 0)



# Analiza metodą kosztu zamortyzowanego

- Złożoność pesymistyczna: *W(n)=O(n)*
- Złożoność średnia? Może *A(n)=O(n/2)*? Nie!

$$\frac{2^n}{2}\cdot 1+\frac{2^n}{4}\cdot 2+\frac{2^n}{8}\cdot 3+\ldots+\frac{2^n}{2^n}\cdot n = 1\cdot 2^{n-1}+2\cdot 2^{n-2}+3\cdot 2^{n-3}+\ldots+n\cdot 2^{n-n}$$

$$= \left(2^{n-1}+2^{n-2}+\ldots+2^0\right)+1\cdot 2^{n-2}+2\cdot 2^{n-3}+\ldots(n-1)\cdot 2^0$$

$$= \left(2^n-1\right)+\left(2^{n-2}+\ldots+2^0\right)+1\cdot 2^{n-3}+\ldots(n-2)\cdot 2^0$$

$$= (2^n-1)+\left(2^{n-1}-1\right)+\ldots+(2^1-1)+\left(2^0-1\right)$$

$$= 2^{n+1}-1-n$$

$$A(n) < \frac{2^{n+1}}{2^n} = 2 \qquad \Rightarrow \qquad A(n) = O(1)$$

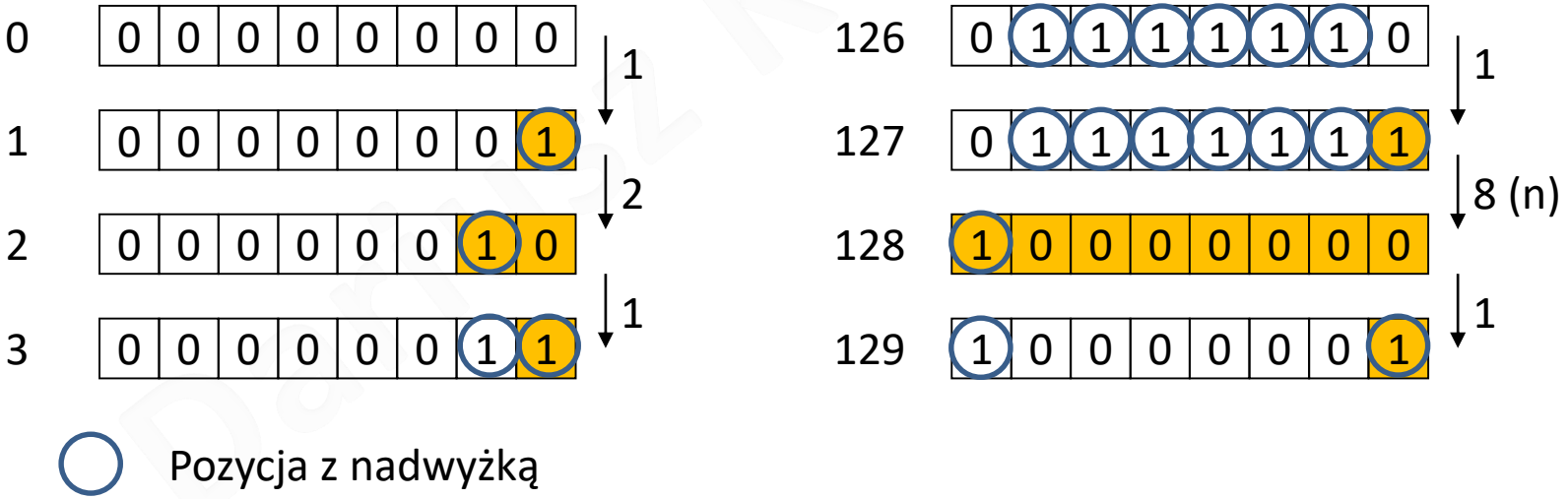
# Metoda księgowania

- W metodzie księgowania kosztu zamortyzowanego, do każdej operacji przyporządkowujemy koszt, który może być mniejszy lub większy od rzeczywistego kosztu. Jednak koszt rzeczywisty nigdy nie może być większy niż powstały zamortyzowany koszt. Nadwyżka ponad rzeczywisty koszt ciągu operacji nazywamy kredytem. Kredyt może być użyty do opłacenia tych operacji, których koszt rzeczywisty jest większy niż ustalony koszt zamortyzowany.
- Najczęściej mamy operacje, które odbywają się wcześniej i których koszt ustalamy na większy niż w rzeczywistości, natomiast operacje późniejsze obciążamy mniejszym kosztem niż koszt rzeczywisty. Natomiast możemy pokazać, że wcześniej zebrany kredyt NA PEWNO uzupełni braki.

# Licznik *n*-bitowy – met. księgowania

Używając metody księgowania dla *n*-bitowego licznika, obciążmy operację zmiany bitu na 1 opłatą równą 2. Gdy ustawiamy bit na 1, jedną z tych dwóch jednostek płacimy za rzeczywisty koszt zmiany bitu, pozostała jednostkę kładziemy  na tym bicie, aby użyć ją przy powrotnej zmianie na 0. W każdym momencie wszystkie 1 w liczniku mają na sobie to jednostkę, więc zmieniając nawet wiele jedynek na zero, mamy wystarczającą nadwyżkę na taką zamianę. Oczywiście płacąc z nadwyżki, zdejmujemy tą jednostkę z danego bitu.

- Ponieważ w każdej sytuacje tylko jedno 0 jest zmieniane na 1, zatem w każdym kroku płacimy 2 jednostki za zmianę, czyli jest to dokładnie średni zamortyzowany koszt operacji zwiększenia o jeden na tym liczniku.



# Sortowanie - definicja

- **Sortowanie**: proces, w którym kolekcję elementów ustawiamy w określonym porządku
  - Operacje porządkujące dane:
    - Porównanie
    - Zamiana lub przypisanie
  - Porządkowanie jest najczęściej wg pewnego klucza
  - Bardziej uniwersalnym podejściem jest używanie **komparatora**

**Definicja formalna:**
Dla początkowej tablicy S, składającej się z N elementów stworzyć tablicę wynikową S' taką, że:
1) $S'_i \leq S'_{i+1}$, dla $0 < i < N$ (elementy są posortowane) oraz
2) S' jest permutacją S.

# Cechy sortowania 1/2

- Metody używania elementów
  - Przez porównanie elementów – komparator
  - Bez bezpośredniego porównywania elementów
- Stabilność - czy elementy o jednakowych kluczach zachowają pierwotną kolejność :
  - **Stabilny**
  - Niestabilny
- Odległość między elementami porównywanymi:
  - **Bliskie**
  - Dalekie
- Możliwość zastosowania do częściowo posortowanej kolekcji:
  - Sortowanie  zawsze całej kolekcji
  - **„Dosortowanie” nowych elementów**

(klucz, dana dodatkowa)

(**5**,3),(**6**,4),(**5**,1),(**2**,4),(**5**,2),(**2**,5)

Stabilność sortowania:

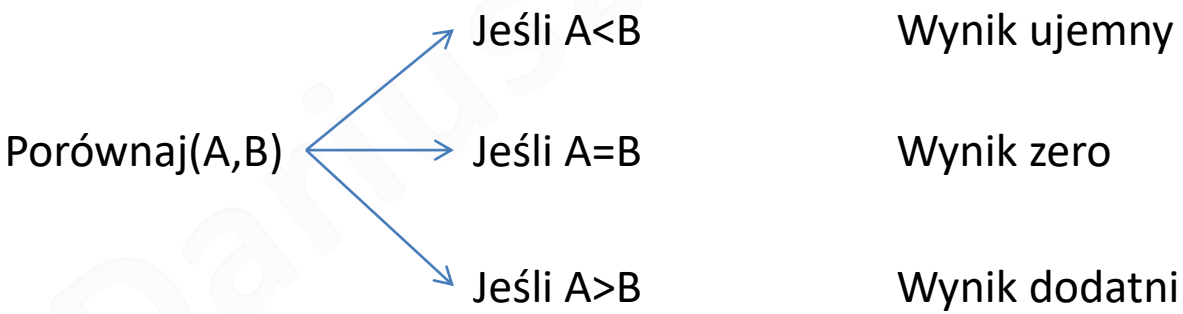
(2,4),(2,5),(5,3),(5,1),(5,2),(6,4)

# Cechy sortowania 2/2

- Złożoność obliczeniowa:
  - **Proste: $O(n^2)$**
  - Pośrednie: $O(n^{3/2})$ i inne
  - **Szybkie: O(n log n)**
  - **Specjalnego zastosowania: O(n)**
- Potrzeba dodatkowej pamięci:
  - **w miejscu - bez dodatkowej pamięci większej niż O(1)**
  - Potrzebna pamięć większa niż O(1)
- Stopień skomplikowania implementacji
- Miejsce lub struktura danych wejściowych:
  - **Pamięć RAM**
  - Plik
  - **Tablica, lista wiązana**
  - Strumień
- Liczba porównywania lub przepisywania elementów (w C++ można przepisywać całe elementy, w Javie – raczej referencje)
- Wrażliwość na przypadki danych wejściowych:
  - dane losowe (najczęstsza sytuacja)
  - dane posortowane (w większość posortowane)
  - dane posortowane odwrotnie
  - dane wszystkie równe (lub wiele równych)
  - inne

# Komparator - idea

- Komparator składa się w zasadzie z jednej funkcji z dwoma parametrami A i B typu X, która zwraca informację, czy element A powinien być w ustalonym porządku przed elementem B, czy powinien być za elementem B, czy też są to te same elementy (ze względu na ustalony porządek, co nie znaczy, że są to elementy identyczne).
- Funkcja ta powinna być określona dla wszystkich możliwych par danego typu X oraz wyznaczać liniowy porządek.
- Liniowy porządek (<=) ma właściwości (dla dowolnych A,B,C typu X):
  - **Zwrotność**: (A<=A)
  - **Spójność**: (A<=B)v(B<=A)
  - **Antysymetryczność:** ((A<=B) oraz (B<=A)) => (A=B)
  - **Przechodniość**: ((A<=B) oraz (B<=C)) => (A<=C)
- Komparator musi zwracać w sumie 3 rodzaje wyniki:



# Interfejs `Comparable`

- Posiada tylko jedną funkcję
- Wyznacza w kolekcji tzw. **porządek naturalny** elementów.
- Wynik powinien być zgodny z poprzednim slajdem.
- Porównuje bieżący obiekt (`this`) z argumentem `other`: `porównaj(this,other)`.
- Wada: w ten sposób obiekt może udostępniać tylko jeden porządek.

```
interface Comparable<T>{
  int compareTo(T other);
}
```

```
public class Student implements Comparable<Student>{
  …
  @Override
  public int compareTo(Student other){
  }
}
```

- Ciekawostka: dla klasy `String` wartość zwracana przez metodę `compareTo()` oznacza różnicę kodów znaków na pierwszym miejscu, gdzie ciągi się różnią lub różnicę długości, gdy jeden jest prefiksem drugiego.

# Klasa `Student` i naturalny porządek

```
public class Student implements Comparable<Student>{
  String nazwisko;
  int nrIndeksu;
  double stypendium;
  public Student(String nazw, int nr, double kwota){
    nazwisko=nazw;
    nrIndeksu=nr;
    stypendium=kwota; }
  public void zwiekszStypendium(double kwota){
    stypendium+=kwota; }
  @Override
  public String toString(){
    return String.format("%6d %8.2f\n",nrIndeksu,stypendium); }
  public boolean equals(Student stud) {
    return nazwisko.equals(stud.nazwisko);}
  @Override
  public boolean equals(Object obj) {
    if(obj==null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    return equals((Student) obj);}
  @Override
  public int compareTo(Student o) {
    return nazwisko.compareTo(o.nazwisko);}
}
```

# Interfejs `Comparator`

- W Javie istnieje inny interfejs: `Comparator` używany w wielu kolekcjach i metodach.
- Posiada tylko jedną funkcję
- Można tworzyć wiele takich interfejsów, bo obydwa argumenty otrzymuje jako parametry.
- Każda implementacja takiego interfejsu to nowa klasa.
- Aby nie tworzyć nazw dla klas z takim komparatorem można tworzyć klasy anonimowe.
- Zamiast w metodach/strukturach korzystających z komparator tworzyć dwie implementacje dla dwóch rodzajów interfejsów, interfejs `Comparable` przekształca się do interfejsu `Comparator`

```
interface Comparator<T>{
  int compare(T left, T right);
}
```

```
class XYZ<T> {
  Comparator<T> comparator;

  public XYZ(Comparator<T> comp){
    comparator=comp;  }
  public XYZ(){
    comparator=new Comparator(){   // klasa anonimowa
      public int compare(T first, T other){
        return first.compareTo(other);
      }
    }
  }
```

To jest szkielet kodu, niekompilowalny

# Komparator porządku naturalnego

- Można stworzyć klasę generyczną to tworzenia komparatorów naturalnych porównujących wg porządku naturalnego podanej w parametrze klasy

```
public class NaturalComparator<T extends Comparable<? super T>> implements Comparator<T> {
    @Override
    public int compare(T o1, T o2) {
        return o1.compareTo(o2);
    }
}
```

# Komparator odwrotny

- Można stworzyć klasę, która umożliwi, z zadanego komparatora wejściowego, stworzenie komparatora odwrotnego

```
public class ReverseComparator<T extends Comparable<? super T>> implements
          Comparator<T> {
  // podstawowy komparator
  private final Comparator<T> _comparator;
  public ReverseComparator(Comparator<T> comparator)
  {
    _comparator = comparator;
  }
  @Override
  public int compare(T left, T right)
  {
    return _comparator.compare(right, left);
  }
}
```

- Poprawny zapis **obiektowy** w języku Java dla takich klas wymagałby dodatkowego wytłumaczenia, stąd w dalszych materiałach zostanie on porzucony, celem skupienia się na algorytmice, a nie specyfikacji klas generycznych języka Java.
- Zamiast tego używane będzie wyłączanie ostrzeżeń o możliwych kłopotach z niepoprawnym rzutowaniem.

# Komparator złożony

- Zamiast tworzyć za każdym razem nowe, skomplikowane komparatory można tworzyć komparator poprzez złożenie prostszych komparatorów.
- W takim przypadku porównanie elementów nastąpi najpierw poprzez użycie pierwszego komparatora. Gdy on uzna, że elementy są równe, uruchamia się drugi komparator itd.
- Komparatory przechowywane będą w liście/tablicy.

```
public class CompoundComparator<T> implements Comparator<T> {
  //tablica komparatorów ; od najważniejszego
  private final IList<Object> _comparators =new ArrayList<Object>();
  public void addComparator(Comparator<T> comparator)
  {
    _comparators.add(comparator);
  }
  @SuppressWarnings("unchecked")
  public int compare(T left, T right) throws ClassCastException {
    int result = 0;
    for (Object obj:_comparators){
      Comparator<T> comp=(Comparator<T>)obj;
      result=comp.compare(left, right);
      if(result!=0) break;
    }
    return result;
  }
}
```

# Testowanie komparatorów

```
public static void main(String[] args) {
  Student stud1=new Student("Nowak",1234, 1200.0);
  Student stud2=new Student("Nowak",1230, 1000.0);
  Student stud3=new Student("Kowalski",1111, 2000.0);
  System.out.println("porzadek naturalny:");
  System.out.println(stud1.compareTo(stud2));
  System.out.println(stud1.compareTo(stud3));
  System.out.println(stud2.compareTo(stud3));
  System.out.println(stud3.compareTo(stud2));
  Comparator<Student> revComp=new ReverseComparator<Student>(
    new NaturalComparator<Student>());
  System.out.println("komparator odwrocony:");
  System.out.println(revComp.compare(stud2,stud3));
  CompoundComparator<Student> complexComp=new CompoundComparator<Student>();
  complexComp.addComparator(new Comparator<Student>(){
    public int compare(Student o1, Student o2) {
      return o1.nazwisko.compareTo(o2.nazwisko);}
  });
  complexComp.addComparator(new Comparator<Student>(){
    public int compare(Student o1, Student o2) {
  return o1.nrIndeksu-o2.nrIndeksu;}
  });
  complexComp.addComparator(new Comparator<Student>(){
    public int compare(Student o1, Student o2) {
      if (o1.stypendium<o2.stypendium)       return -1000;
      else if(o1.stypendium>o2.stypendium)   return 1000;
      else return 0;}
  });
  System.out.println("komparator złożony:");
  System.out.println(complexComp.compare(stud1,stud2));
}
```

porzadek naturalny:
0
3
3
-3
komparator odwrocony:
-3
komparator złożony:
4

# Interfejs `RandomAccess`

- Jeden z kilku interfejsów **bez metod**!
- Służy jako **oznaczenie** kolekcji jako takiej, w której metody `get(int index)` i `set(int index, T element)` są stałoczasowe,czyli O(1).
- Np. metody z klasy `Collections` sprawdzają, czy kolekcja implementuje ten interfejs czy nie i uruchamia odpowiednie wewnętrzne działanie.
- Klasa `ArrayList` **zapewnia** interfejs `RandomAccess`.
- Klasa `LinkedList` **nie zapewnia** interfejsu `RandomAccess`.

```
interface RandomAccess{
}
```

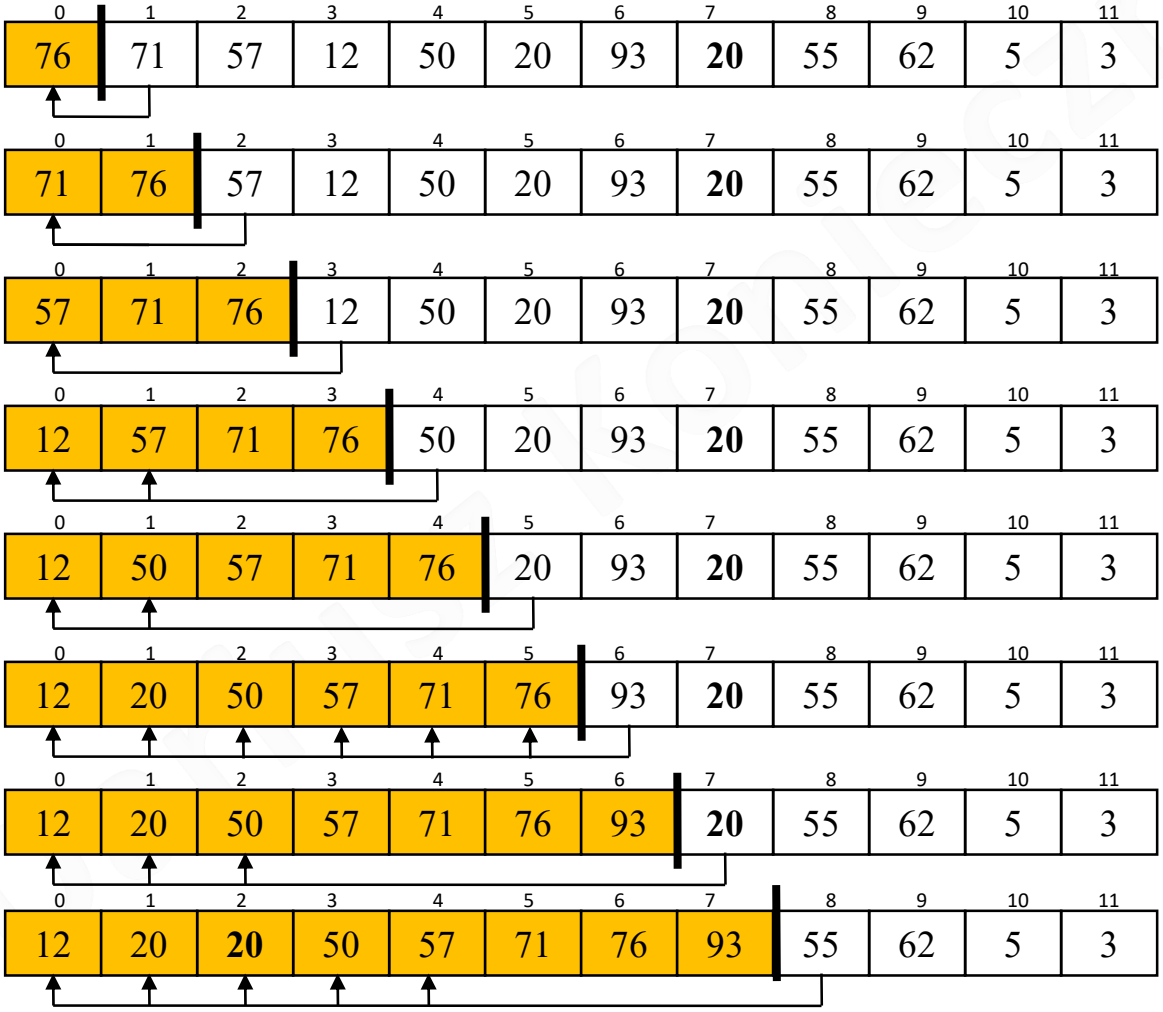
# Interfejs `ListSorter`

- Dla celów dydaktycznych stwórzmy interfejs dla klas sortujących `ListSorter<T>`.
- Każda klasa z algorytmem sortowania będzie musiała implementować ten interfejs.
- W większości metod prezentowanych na wykładzie będzie założenie, że lista wejściowa dostarczają interfejsu `RandomAccess`.
- Implementacja prezentowanych algorytmów w wersji dla list wiązanych jest dobrym ćwiczeniem praktycznym.

```
public interface ListSorter<T> {
  public IList<T> sort(IList<T> list);
}
```

# Sortowanie przez wstawianie

- Jak układanie kart do gry po rozdaniu graczom
- Idea: do części już posortowanej dokładany kolejny element, szukając dla niego poprawnej pozycji.



# S. przez wstawianie - kod

- Wersja z poszukiwaniem miejsca do wstawienia od prawej strony
- Podczas poszukiwania miejsca – przesuwanie elementów.

```
public class InsertSort<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public InsertSort(Comparator<T> comparator)
  {
    _comparator = comparator;
  }
  public IList<T> sort(IList<T> list) {
    for (int i = 1; i < list.size(); ++i) {
      T value = list.get(i),temp;
      int j; // będzie wykorzystywane poza pętlą
      for (j = i; j > 0&& _comparator.compare(value, temp=list.get(j - 1))< 0; --j)
        list.set(j,temp);
      list.set(j, value);
    }
    return list;
  }
}
```

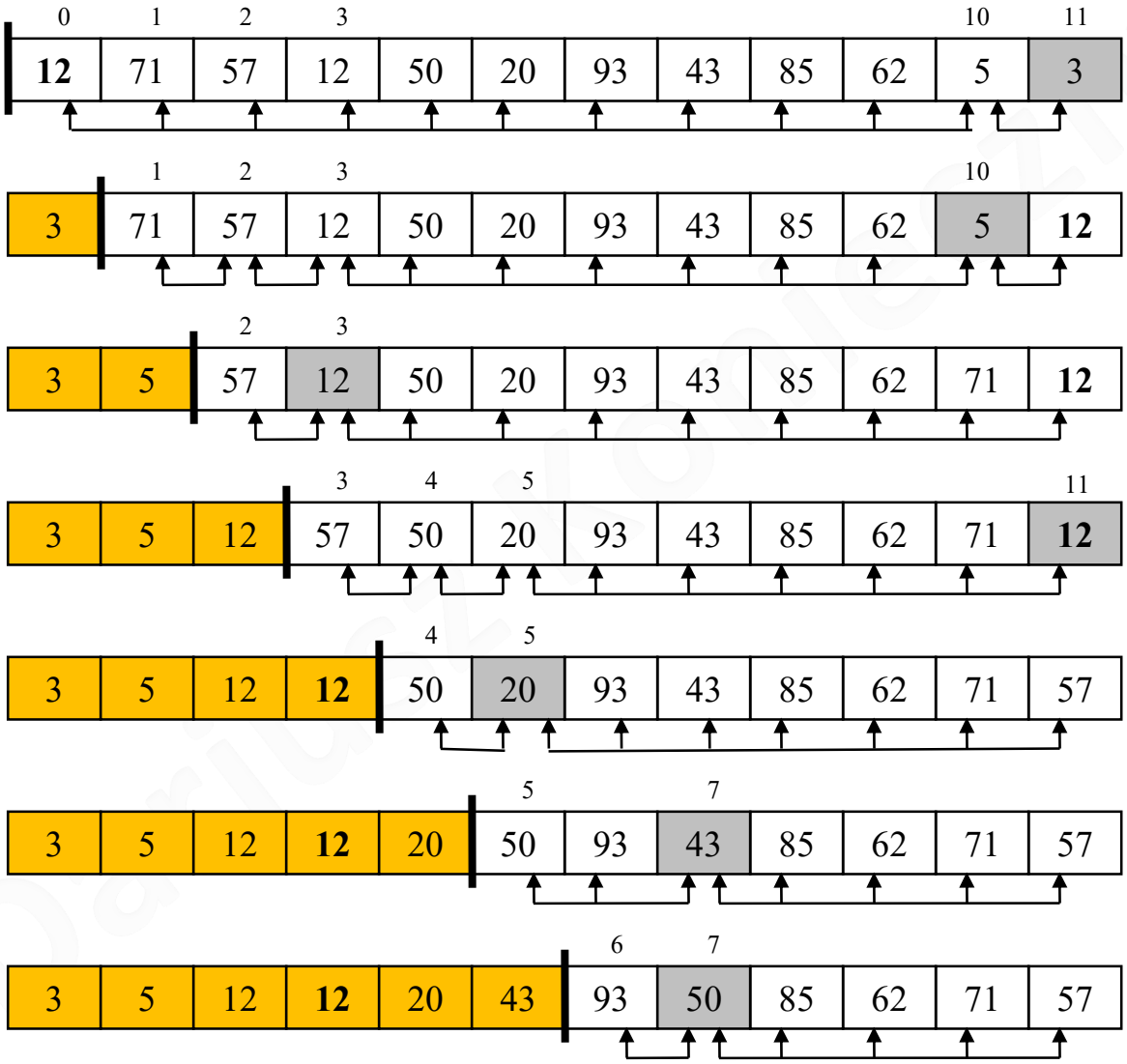
# S. przez wstawianie - analiza

- Złożoność obliczeniowa (pesymistyczna, średnia) – $O(n^2)$
- Sortowanie w miejscu
- Sortowanie stabilne
- Działa szybko dla małych list – algorytm progowy dla algorytmu szybkiego
- Idea sortowania przez wstawianie używana w listach wiązanych posortowanych
- Warianty:
  - Część posortowana narasta od lewej/prawej strony
  - Szukamy miejsca do wstawienia od lewej/prawej strony

- Możliwość usprawnień:
  - Szukać miejsca wstawienie używając poszukiwania binarnego – czas $O(\log n)$
  - Szukać miejsca od prawej strony – szczególnie gdy kolekcja jest w dużych fragmentach już posortowana: 1,3,5,6,2,7,8,10,11,9,13

# Sortowanie przez wybór

- Idea: szukamy kolejnej wartości minimalnej (maksymalnej) oraz dostawienie jej do wcześniej znalezionych wartości.



# S. przez wybór - kod

- Wybór minimum
- Wydzielona operacja `swap`

```
public class SelectSort<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public SelectSort(Comparator<T> comparator) {
    _comparator = comparator;
  }
  public IList<T> sort(IList<T> list) {
    int size = list.size();
    for (int slot = 0; slot < size - 1; ++slot) {
      int smallest = slot;  // pozycja wartości minimalnej
      for (int check = slot + 1; check < size; ++check)
        if (_comparator.compare(list.get(check), list.get(smallest)) < 0)
          smallest = check;
      swap(list, smallest, slot);
    }
    return list;
  }
  private void swap(IList<T> list, int left, int right) {
    if (left != right) {
      T temp = list.get(left);
      list.set(left, list.get(right));
      list.set(right, temp);
    }
  }
}
```

# S. przez wstawianie - analiza

- Złożoność obliczeniowa pesymistyczna i średnia – $O(n^2)$
- Sortowanie w miejscu
- Sortowanie niestabilne! – dodanie w komparatorze porównywania indeksu sprzed sortowania wprowadza stabilność
- Łatwe do implementacji, szczególnie, gdy już jest zaimplementowanie poszukiwanie pozycji minimalnego elementu
- Dużo porównań $O(n^2/2)$, ale tylko $O(n)$ przepisań/zamian
- Usprawnienia:
  - Szukać jednocześnie wartości minimalnej i maksymalnej – mniej porównań i podstawień elementów.

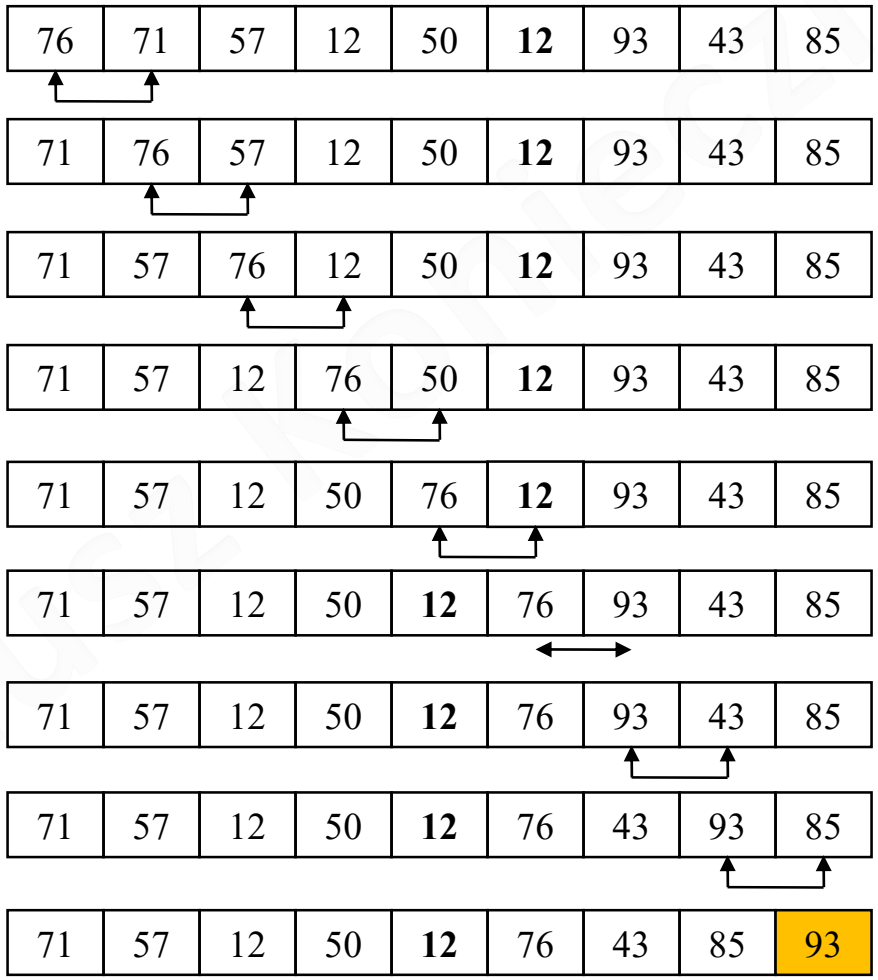
# Sortowanie bąbelkowe

- Idea: lokalne zamiany między sąsiednimi elementami, jeśli nie są we właściwej kolejności. Zamiany robi się od lewej do prawej od zerowego elementu, potem od pierwsze go itd.

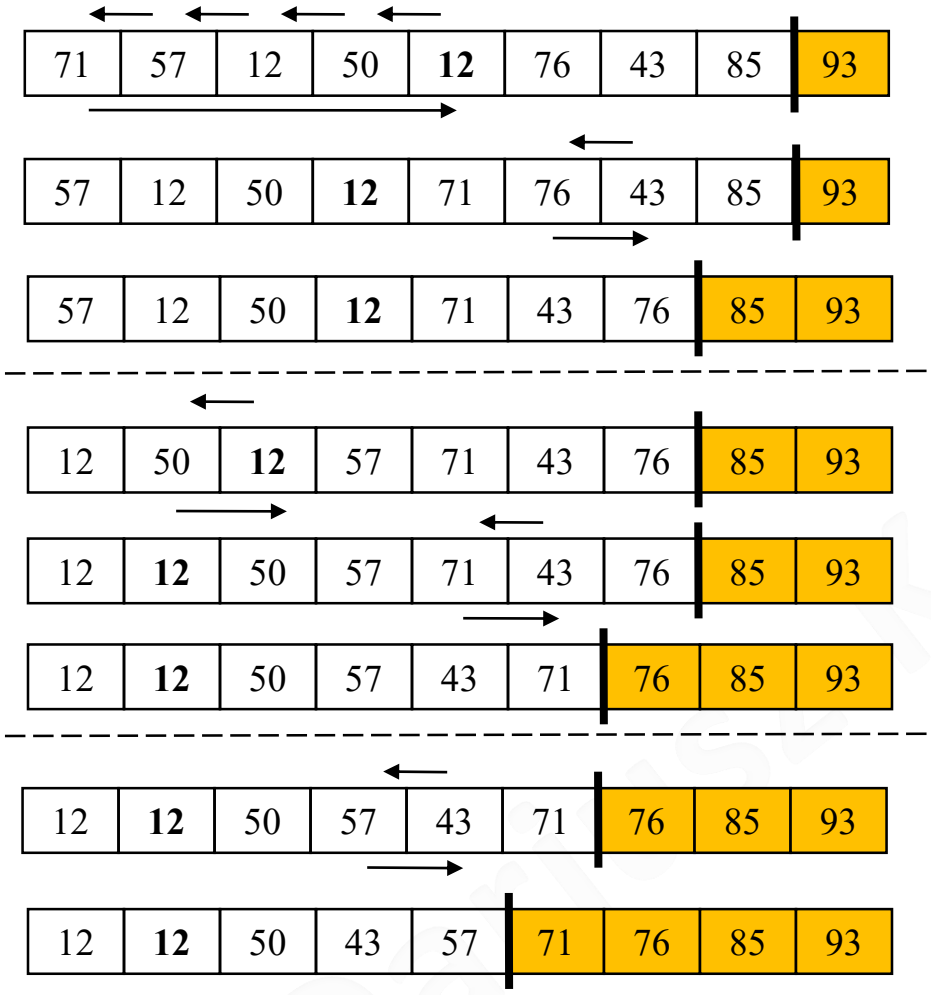
Jeden duży krok algorytmu

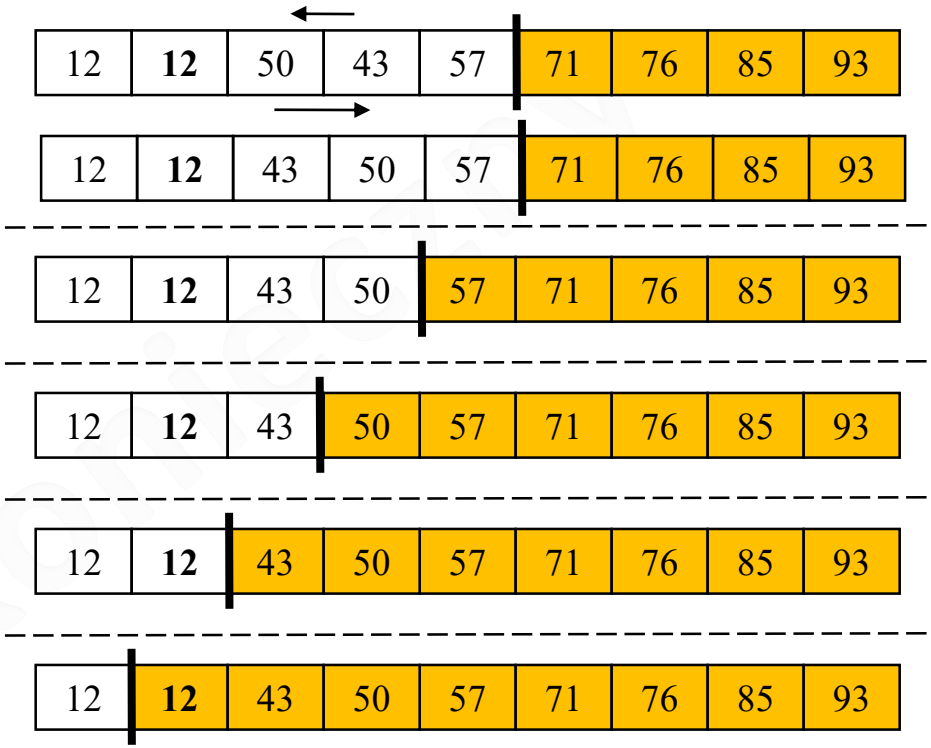

zamiana


bez zamiany



# Sortowanie bąbelkowe – c.d.



# Sortowanie bąbelkowe - kod

```
public class BubbleSort<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public BubbleSort(Comparator<T> comparator)
    { _comparator = comparator; }
  // the result is a sorted primary list
  // najbardziej prymitywna wersja
  public IList<T> sort(IList<T> list) {
    int size = list.size();
    for (int pass = 1; pass < size; ++pass) {
      for (int left = 0; left < (size - pass); ++left) {
        int right = left + 1;
        if (_comparator.compare(list.get(left), list.get(right)) > 0)
          swap(list, left, right);
      }
    }
    return list;
  }
  private void swap(IList<T> list, int left, int right) {
    T temp = list.get(left);
    list.set(left, list.get(right));
    list.set(right, temp);
  }
}
```

# S. bąbelkowe - analiza

- Złożoność obliczeniowa pesymistyczna i średnia – $O(n^2)$
  - Wolne działanie tego algorytmu wynika z faktu że pojedyncza zamiana sąsiednich elementów zmienia stopień nieuporządkowania ciągu (liczony jako liczba inwersji - ile razy wartość większa występuje przed mniejszą) tylko o 1.
- Bardzo łatwy do implementacji
- Sortowanie stabilne
- Sortowanie w miejscu
- Bliska odległość między porównywanymi elementami
- Możliwe usprawnienia:
  - Sprawdzać, czy ciąg nie jest już posortowany (nie było żadnej zamiany)
  - Pamiętać miejsce ostatniej zmiany (dalej ciąg jest już posortowany i nie ulegnie zmianie)
  - Zamiast ciągu lokalnych zamian (operacja `swap`) robić przesunięcia w lewo – 3 razy mniej podstawień.
  - Poruszać się raz w prawo, raz w lewo (ShakerSort) – dobre dla ciągów w większości posortowanych (z pierwszym usprawnieniem)
- Usprawnienia mogę pogorszyć czas wykonania gdy ciąg jest losowy, a porównanie elementów szybkie – dominuje czas potrzebny na dodatkowe operacje.

# S. bąbelkowe – kod z usprawnieniami

- Kod dla 3 pierwszych usprawnień:

```
public class BubbleSortBetter<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public BubbleSortBetter(Comparator<T> comparator)
  {
    _comparator = comparator;
  }
  // wynikiem jest posortowana lista pierwotna
  // wersja ulepszona wykrywająca wcześniejsze uporządkowanie
   public IList<T> sort(IList<T> list) {
     int lastSwap = list.size()-1; //pozycja ostatniej zamiany
     while(lastSwap>0){
       int end=lastSwap;
       lastSwap=0;
       for (int left = 0; left < end; ++left) {
         if (_comparator.compare(list.get(left), list.get(left+1)) > 0)
         { //ciąg zamian jest zastąpiony ciągiem przepisań
           T temp=list.get(left);
           while(left<end && _comparator.compare(temp, list.get(left+1)) > 0)
           { list.set(left, list.get(left+1)); left++; }
           lastSwap=left;
           list.set(left,temp);
         }
       }
    }
  return list;
  }
}
```

# Podsumowanie

- Sortowania proste o złożoności O($n^2$) są nieefektywne dla dużych kolekcji
- Dla małych kolekcji potrafią być szybsze od algorytmów o złożoności O($n$ log $n$), które będą przedstawione na kolejnym wykładzie
  - Używane są jako algorytmy progowe, gdy kolekcje są małe
- Istnieje algorytm używający algorytmu bąbelkowego jako wewnętrzny – ShellSort – o złożoności O($n^{3/2}$).
- Prawie wszystkie przedstawione implementacje używają metod `get/set` z indeksem i zakładają, że operacje te są o złożoności O(1) (interfejs `RandomAccess`).
  - Jeśli (np. dla `LinkedList`), operacje te są złożoności O($n$), to ostateczna złożoność przedstawionych sortowań będzie O($n^3$) !
- Część z tych algorytmów można przekształcić na wersję używającą iteratorów, wtedy byłaby użyteczna dla list wiązanych
  - ale mogłaby być mniej efektywna dla tablic