Data Structures and Algorithms – W04

Complexity part 4,
Simple sorting algorithms

Contents

- Complexity part 4 - complexity analysis:
  - Types of complexity:
    - Pessimistic (worse-case complexity)
    - Average/expected
  - The accounting method
- Sorting – definition
- Interfaces comparing objects - comparators:
  - `Comparable` (natural order)
  - `Comparator`
- Reverse comparator
- Compound comparator
- Interface `RandomAccess`
- Simple sorting algorithms - $O(n^2)$
  - Insertion sorting
  - Selection sorting
  - Bubble sorting

Computation complexity of alg.

Computation complexity:

- worse-case complexity, pessimistic complexity
- expected/average complexity:
  - analysis of the amortized cost
    - aggregate analysis
    - accounting method

Notation

- An input *I* to an algorithm is the data, such as numbers, character strings and records, on which the operations of the algorithm are performed
- Let F_n denote the set of all inputs of size *n* to an algorithm
- For $I \in \mathsf{F}_n$ let $\tau(I)$ denote the number of basic operations performed when the algorithm is executed with input *I*
- τ is read as *tau*

Worse-case complexity

The *worse-case complexity* of an algorithm is the function *W(n)* such that *W(n)* equals the *maximum* value of $\tau(I)$, where *I* varies over all inputs of size *n*. That is,

$$W(n) = \max\{\tau(I) \mid I \in \mathsf{F}_n\}$$

Worse-case complexity – example

Sorting *n* elements by generating all permutation one by one, checking if the next generated sequence is sorted. If it is true, stop generating the permutations.

- Assumptions:
  - next generation is made in one elementary step
  - checking if the permutation is a solution is made in one elementary step

With both assumptions the worse complexity is:

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

Average (Expected) complexity

Let *p(I)* be a probability that *I* can be an input set

The *average (expected) complexity* of an algorithm with finite input set F_n is defined as:

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

Formulation II

Let p_i denote the probability that the algorithm performs *exactly i* basic operation; that is $p_i=P(\tau=i)$. Then the average complexity formula can be changed to:

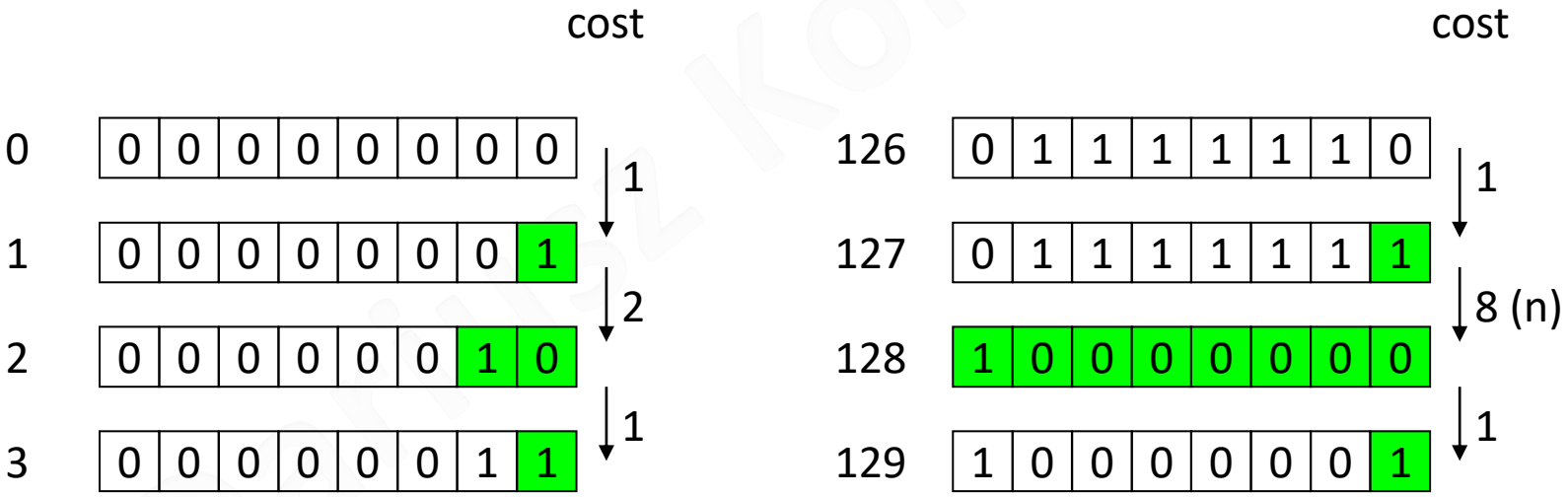
$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} ip_i$$

Aggregate analysis

In ***aggregate analysis***, we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)/n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.

Aggregate analysis – Example – bit counter

- We have a *n*-bits counter, which can grows by one (the only operation) , where *n* – size of input.
- Elementary operation (cost) – change of **one** bit in a counter
- An example below for *n*=8 bit counter:



Bit counter – *W(n), A(n)*

- Worse-case complexity: *W(n)=O(n)*
- Average complexity ? Maybe *A(n)=O(n/2)*? No!

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

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

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

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

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

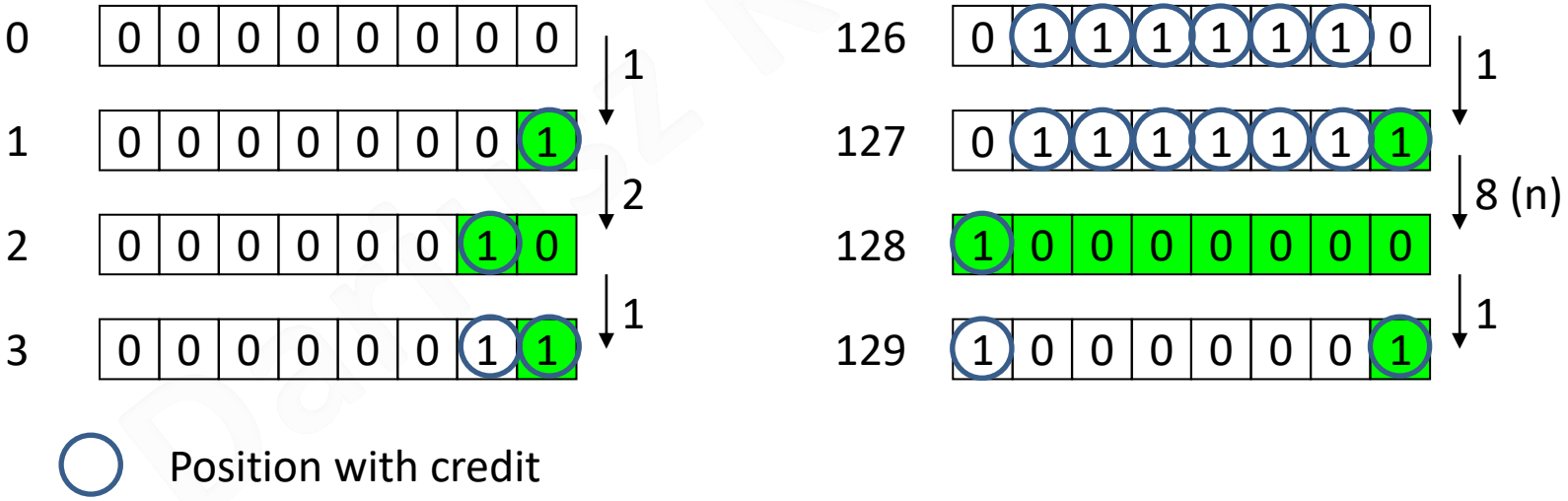
$$A(n) < \frac{2^{n+1}}{2^n} = 2 \qquad \Rightarrow \qquad A(n) = O(1)$$

The accounting method

- In the ***accounting method*** of amortized analysis, we assign ***differing charges to different operations***, with some operations charged more or less than they actually cost.
- The amount we charge an operation is called its ***amortized cost.*** When an operation's amortized cost exceeds its actual cost, ***the difference*** is assigned to specific objects in the data structure as ***credit.***
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.
- Thus, one can view the amortized cost of an operation as being **split** between its **actual cost** and **credit** that is either deposited or used up.

Incrementing a binary counter

- For the amortized analysis, let us **charge** an **amortized cost of 2 dollars** to set a bit to 1. When a bit is set, we use **1 dollar** (out of the 2 dollars charged) **to pay** for the actual setting of the bit, and we place the **other dollar** on the bit as credit to **be used later** when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.
- So the average amortized cost of the increase by one on this counter is exactly 2.



Sorting - definition

- **Sorting: the process** in which the collection of elements **is set in a specific order**
- Ordering operations**:**
  - **Comparision**
  - **Swap** or **assigment**
- Ordering is usually according to a certain key
  - A more **universal** approach is to use a **comparator**

Formal definition:

For the initial array *S*, consisting of *N* elements, create the result array S′ such that:

1) $S'_i \leq S'_{i+1}$, for $0 < i < N$ (elements are sorted) and
2) S′ is the permutation of S.

Sorting features 1/2

- Methods of using elements
  - By comparing elements - a comparator
  - Without direct comparison of elements
- Stability - whether elements with identical keys will keep the original order:
  - **Stable**
  - Unstable
- The distance between the elements compared:
  - **Close**
  - Outlying
- Can be applied to a partially sorted collection:
  - Sorting always the entire collection
  - **„In-sorting" of new elements**

(key, data)

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

Stability of sorting:

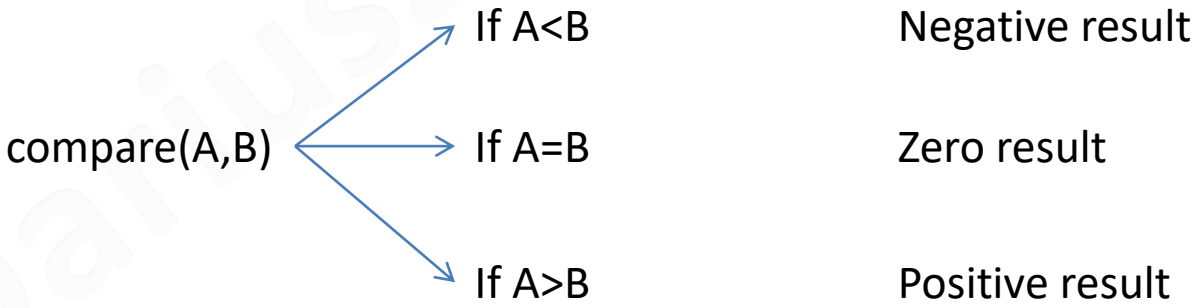
(2,4),(2,5),(5,3),(5,1),(5,2),(6,4)

Sorting features 2/2

- Computational complexity:
  - **Simple: O(n^2)**
  - Intermediate: O($n^{3/2}$) and others
  - **Fast: O(n log n)**
  - **Special application: O(n)**
- Need of extra memory:
  - **in place - without additional memory greater than O(1)**
  - not in-place - needed memory greater than O(1)
- The degree of complexity of the implementation
- Place or structure of input data:
  - **RAM**
  - File
  - **Table, bound list**
  - Stream
- Number of comparing or rewriting elements (in C ++ you can rewrite whole elements, in Java - rather references)
- Sensitivity to input data cases:
  - random data (the most common situation)
  - data sorted (mostly sorted)
  - data sorted in reverse
  - data all equal (or many equal)
  - other

Comparator - idea

- The comparator consists basically of one function with two parameters A and B of type X, which returns information whether element A should be in a chosen order before element B, should it be after element B, or are they the same elements (due to on the established order, which does not mean that they are **identical** elements).
- This function should be determined for all possible pairs of a given type X and determine a **linear order**.
- The **linear order** (<=) has properties (for any A, B, C of type X):
  - reflexivity: (A <= A)
  - Connexity: (A <= B) v (B <= A)
  - Antisymmetry ((A <= B) and (B <= A)) => (A = B)
  - Transitivity: ((A <= B) and (B <= C)) => (A <= C)
- The comparator must return a total of 3 types of results:



Interface `Comparable`

- It has only one method
- Designates the so-called the natural order of the elements.
- The result should match the previous slide.
- Compares the current object (**`this`**) with the argument `other`: compare (**`this`**, `other`).
- Disadvantage: in this way the object can only provide one order.

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

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

- Interesting fact: for the `String` class, the value returned by the `compareTo ()` method means the difference of character codes in the first place, where the strings differ or the length difference when one is the prefix of the other.

`Student` class and natural order

```
public class Student implements Comparable<Student>{
  String name;
  int indexNo;
  double scholarship;
  public Student(String name, int no, double value){
    this.name=name;
    indexNo=no;
    scholarship=value; }
  public void increaseScholarship(double value){
    sholarship+=value; }
  @Override
  public String toString(){
    return String.format("%6d %8.2f\n", indexNo, scholarship); }
  public boolean equals(Student stud) {
    return name.equals(stud.name);}
  @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 name.compareTo(o.name);}
}
```

Interface `Comparator`

- There is another interface in Java: `Comparator` used in many collections and methods.
- It has only one method
- You can create many such interfaces, because both arguments are given as parameters.
- Each implementation of such an interface is a new class.
- In order not to create names for classes with such a comparator, you can create anonymous classes.
- Instead of creating two implementations for two kinds of interfaces in the methods / structures that use the comparator, the `Comparable` interface is transformed to the `Comparator` interface

```
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(){   // anonymous class
      public int compare(T first, T other){
        return first.compareTo(other); // non-compilable
      }
    }
  }
```

This is the skeleton code, non-compilable

Natural order comparator

- You can create a generic class to create natural comparators comparing according to the natural order given in the class parameter

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

Reverse Comparator

You can create a class that will allow you to create a reverse comparator from a given input comparator

```
public class ReverseComparator<T extends Comparable<? super T>> implements
        Comparator<T> {
  // input comparator
  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);
  }
}
```

- Correct object-oriented writing in Java for such classes would require additional explanation, hence in further materials it will be abandoned, in order to focus on algorithmics, not Java generic classes specifications.
- Instead, you will be used to disable warnings about possible problems with incorrect casting.

Compound comparator

- Instead of creating new, complicated comparators each time, you can create a comparator by assembling simpler comparators.
- In this case, the comparison of elements will take place first using the first comparator. When he determines that the elements are equal, the second comparator is activated, etc.
- The comparators will be stored in the list/array.

```
public class CompoundComparator<T> implements Comparator<T> {
  // array of comparators; from the most important
  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;
  }
}
```

Testing comparators

```
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(„natural order:");
  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(„reverse comparator:");
  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.name.compareTo(o2.nazwisko);}
  });
  complexComp.addComparator(new Comparator<Student>(){
    public int compare(Student o1, Student o2) {
  return o1.indexNo-o2.indexNo;}
  });
  complexComp.addComparator(new Comparator<Student>(){
    public int compare(Student o1, Student o2) {
      if (o1.scholarship<o2.scholarship)       return -1000;
      else if(o1.scholarship>o2.scholarship)   return 1000;
      else return 0;}
  });
  System.out.println(„Compound comparator:");
  System.out.println(complexComp.compare(stud1,stud2));
}
```

natural order:
0
3
3
-3
reverse comparator:
-3
Compound comparator:
4

Interface `RandomAccess`

- One of several interfaces without methods!
- It serves as a designation of the collection as such in which the methods `get(int index)` and `set(int index, T element)` are fixed O(1).
- For example, methods from the `Collections` class check whether the collection implements this interface or not and starts the corresponding internal action.
- The `ArrayList` class provides the `RandomAccess` interface.
- The `LinkedList` class **does not** provide the `RandomAccess` interface.

```
interface RandomAccess{
}
```

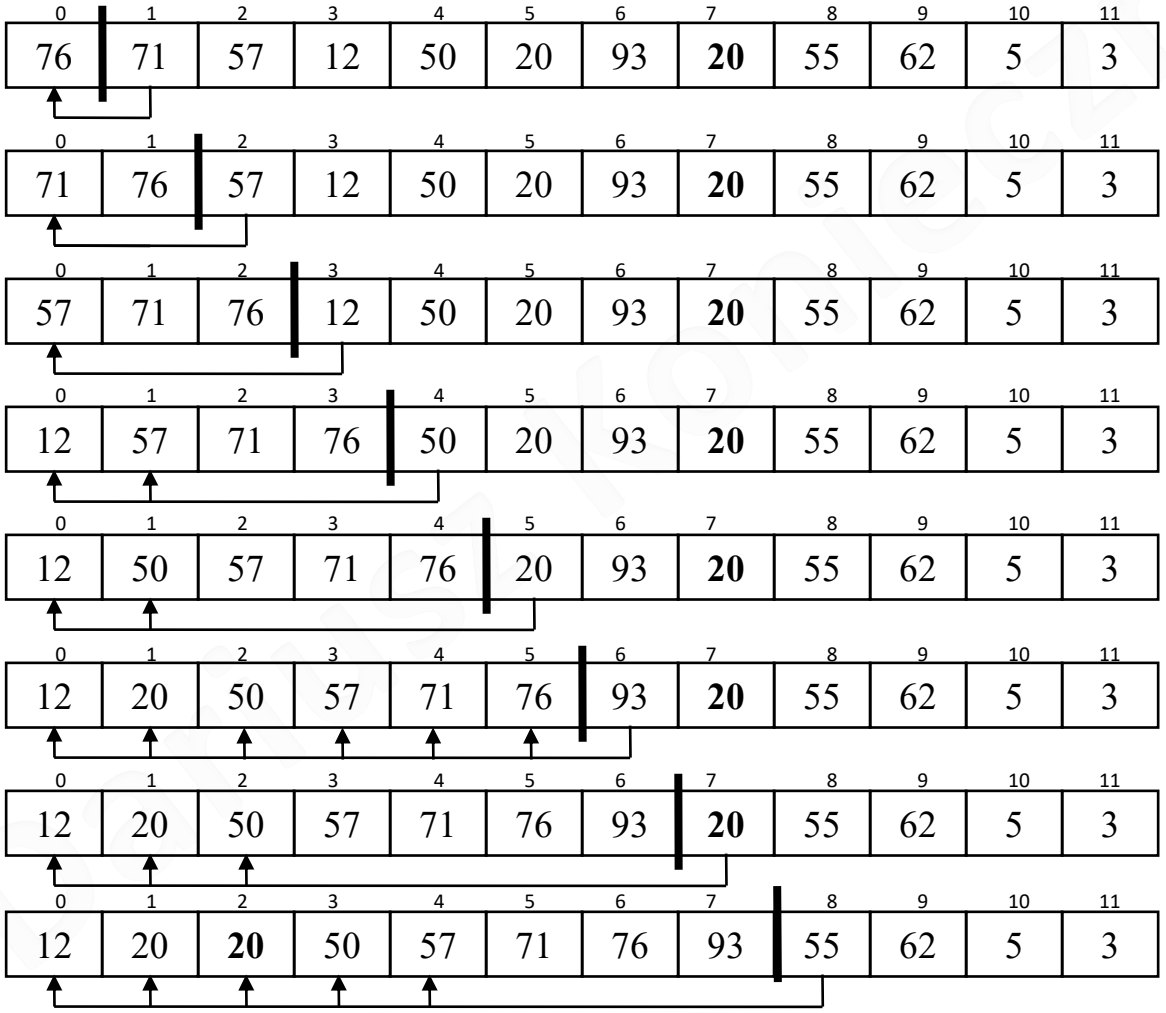
Interface `ListSorter`

- For teaching purposes, let's create an interface for sorting classes `ListSorter<T>`.
- Each class with a sorting algorithm will have to implement this interface.
- Most of the methods presented during the lecture will assume that the input list provides the `RandomAccess` interface.
- Implementation of presented algorithms in the version for *linked lists* is a good practical exercise.

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

Insertsort - example

- How to arrange playing cards after dealing to players
- Idea: the next element is added to the already sorted part, looking for the correct position for him.



Insertsort - code

- Version with searching for a place to insert from the right side
- When searching for a place - moving elements.

```
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; // it will be used outside the loop
      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;
  }
}
```

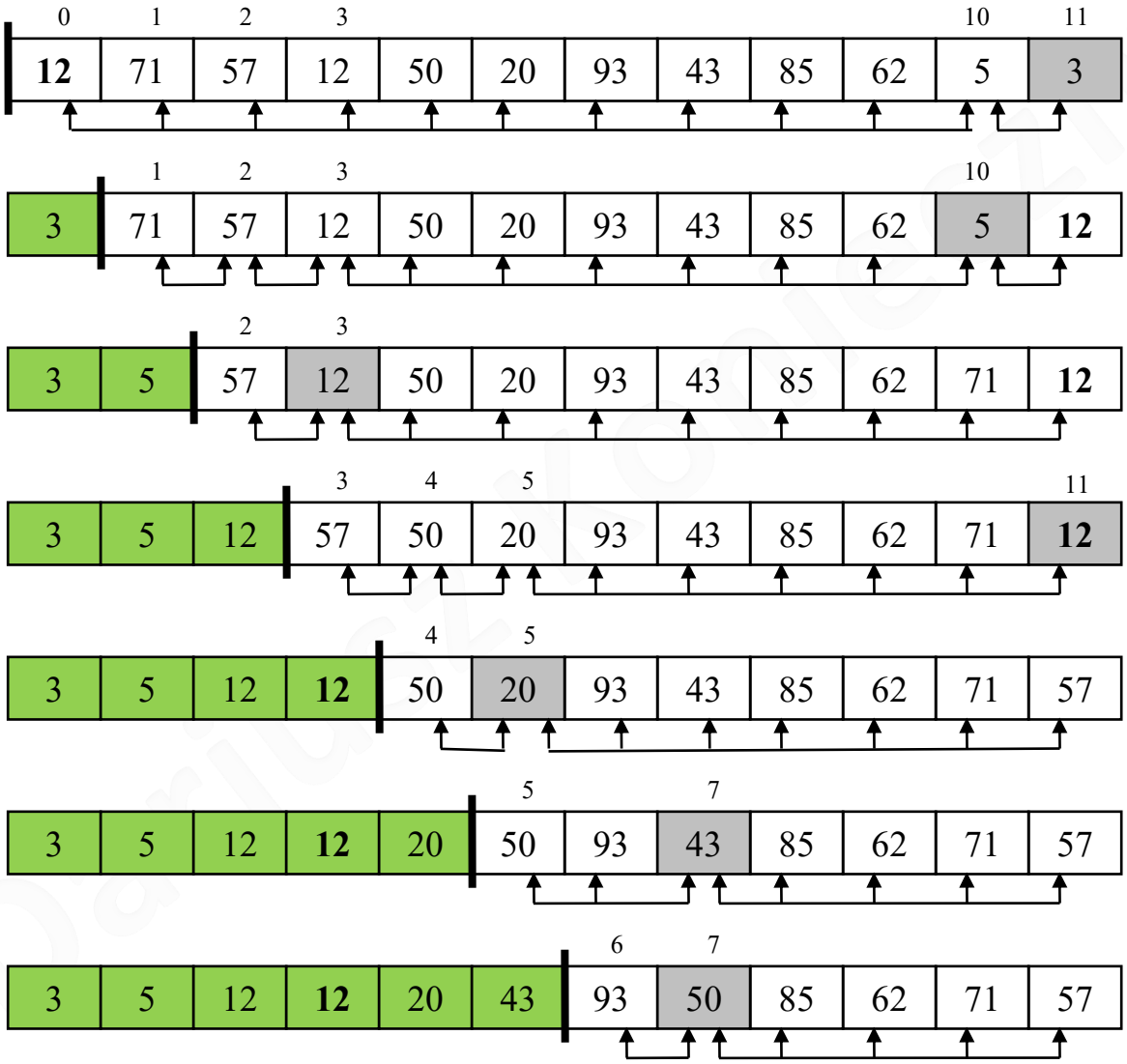
Insertsort - analysis

- Computational complex (pessimistic, average) - $O(n^2)$
- Sorting in place
- Stable sorting
- It works quickly for small lists - a threshold algorithm for a fast algorithm
- The idea of sorting by inserting used in linked lists
- Variants:
  - The sorted part grows from the left / right side
  - We are looking for a place to insert from the left / right side

- Possibility to improve:
  - Look for a place to insert using a binary search - time O(log n)
  - Look for the place from the right side - especially when the collection is already in large fragments sorted: 1,3,5,6,2,7,8,10,11,9,13

Selection sorting

- Idea: we are looking for another minimal (maximum) value and adding it to previously found values.



Selectsort - code

- Selecting the minimum
- Separated swap operation

```
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;  // position of minimum value
      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);
    }
  }
}
```

Selectsort - analysis

- Computational complexity pessimistic and average - $O(n^2)$
- Sorting in place
- Unstable sorting! - adding stability in the comparison comparator of the pre-sorting index in the comparator
- Easy to implement, especially when it is already implemented to search for the position of the minimum element
- A lot of comparisons $O(n^2/2)$, but only $O(n)$ assigments/ swaps

- Improvements:
  - At the same time, look for the minimum and maximum values - fewer comparisons and substitutions of elements.

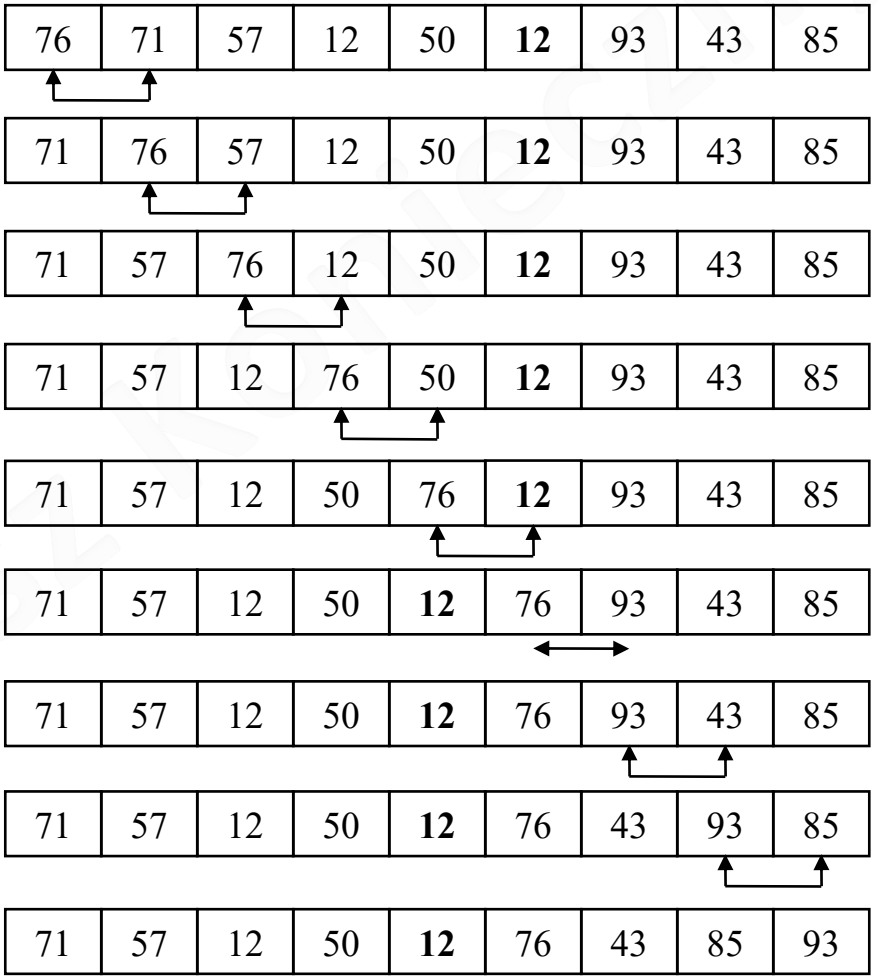
Bubble sorting

- Idea: local swapping between adjacent elements if they are not in the right order. The exchange is made from left to right from the zero element, then from the first one, etc.

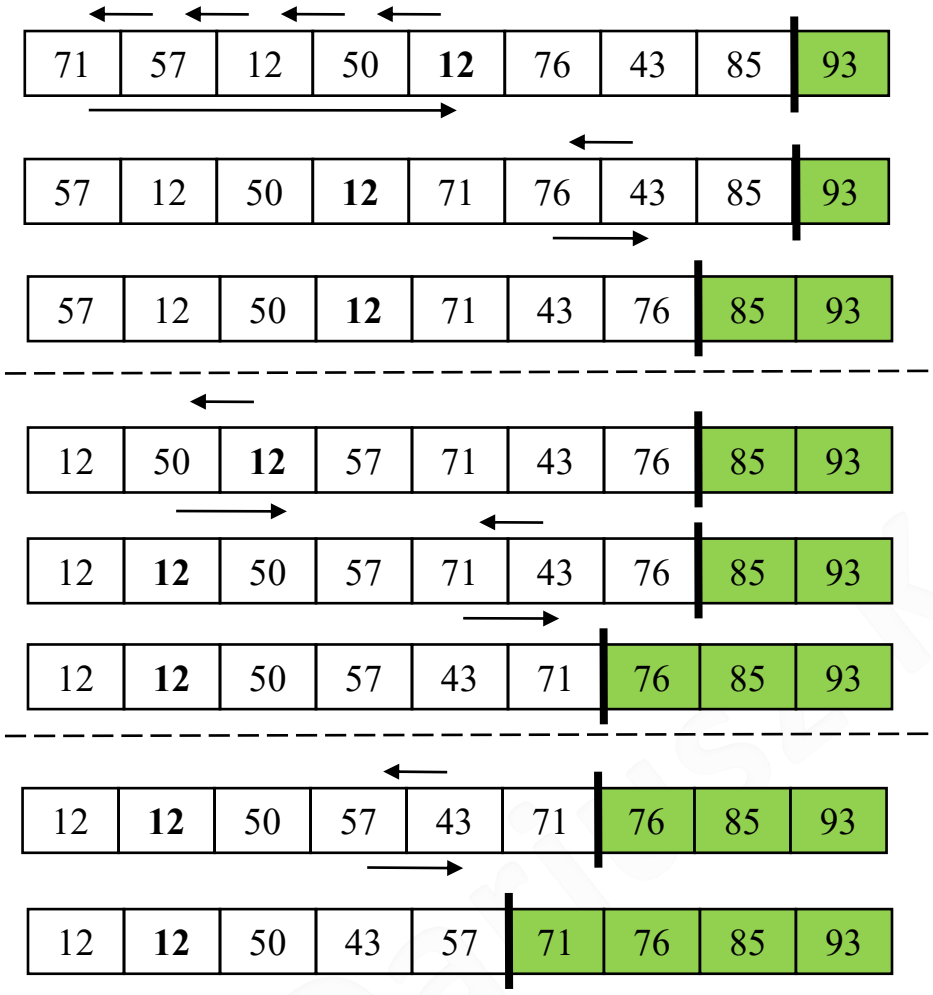
One big step of the algorithm

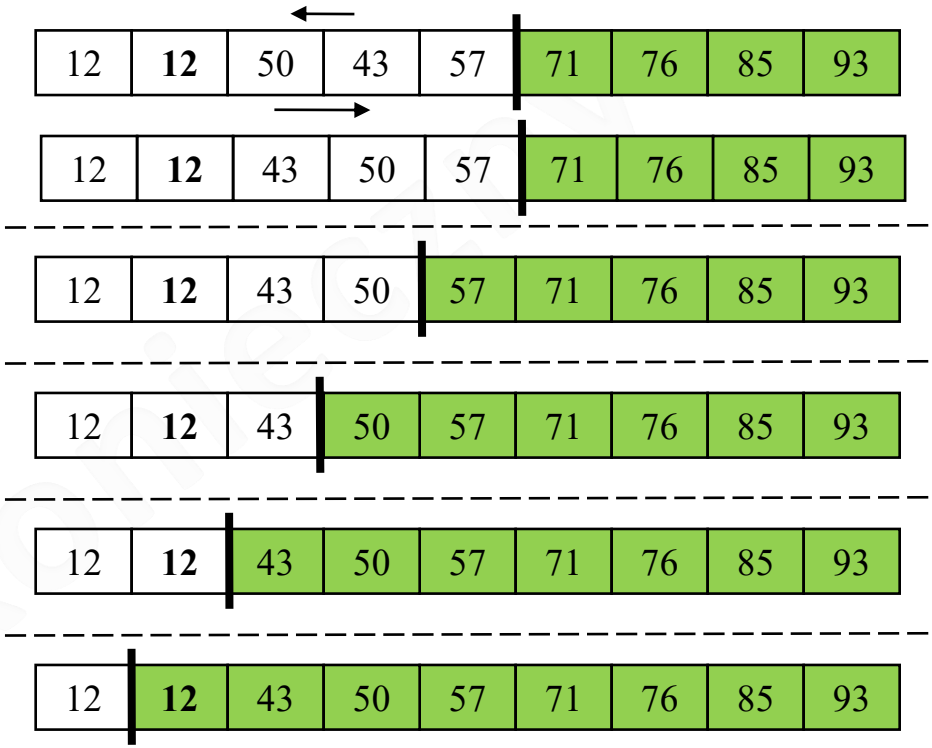
swap

no swap



Bubble sorting – cont.





Bubblesort - code

- Implementation of raw version of bubblesort

```
public class BubbleSort<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public BubbleSort(Comparator<T> comparator)
    { _comparator = comparator; }
  // result is in original list
  // the most primitive version
  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);
  }
}
```

Bubblesort - analysis

- Computational complexity pessimistic and average - $O(n^2)$
  - The slow operation of this algorithm results from the fact that a single replacement of neighboring elements changes the degree of disorder of the sequence (counted as the number of inversions - how many times the larger value occurs before the smaller one) by only 1.
- Very easy to implement
- Stable sorting
- Sorting in place
- Short distance between the compared elements
- Possible improvements:
  - Check if the array is already sorted (there was no swap)
  - Remember the location of the last change (the sequence from that point is already sorted and will not change)
  - Instead of a local swap (swap operation), make shifts to the left - 3 times fewer assignments.
  - Move once to the right, then left etc. (ShakerSort) - good for almost sorted sequences (with the first improvement)
- Improvements can worsen the execution time when the array is random, and the comparison of elements is fast - the time needed for additional operations dominates.

Bubblesort- code with improvements

- Code for the 3 first improvements

```
public class BubbleSortBetter<T> implements ListSorter<T> {
  private final Comparator<T> _comparator;
  public BubbleSortBetter(Comparator<T> comparator)
  {
    _comparator = comparator;
  }
//version with 3 first improvements
   public IList<T> sort(IList<T> list) {
     int lastSwap = list.size()-1; //position of last swap
     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)
         { //a sequence of swaps turned into a sequence of assigments
           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;
  }
}
```

Summary

- Simple sorts with complexity $O(n^2)$ are ineffective for large collections
- For small collections they can be faster than algorithms with complexity O(n log n), which will be presented at the next lecture
- They are used as threshold algorithms when collections are small
- There is an algorithm that uses the bubble sort algorithm as internal - ShellSort - with complexity $O(n^{3/2})$.
- Almost all presented implementations use the `get/set` methods with the index and assume that these operations are of complexity O(1) (`RandomAccess` interface).
  - If (eg for `LinkedList`), these operations are of complexity O(n), then the final complexity of the sorts presented will be $O(n^3)$!
- Some of these algorithms can be converted to a version using iterators, then it would be useful for linked lists
  - but it could be less effective for arrays