CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

12: Iterators, Comparators, Selection Sort

Alexandra Papoutsaki
she/her/hers

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Sorting
- Selection sort

Some slides adopted from Algorithms 4th Edition or COS226

Iterator Interface

- Interface that allows us to traverse a collection (i.e. data structure) one element at a time.

```
public interface Iterator<E> {
  //returns true if the iterator has more elements
  //that is if next() would return an element instead of throwing an exception
  boolean hasNext();

  //returns the next element in the iteration
  //post: advances the iterator to the next value
  E next();

  //removes the last element that was returned by next
  default void remove(); //optional, better avoid it altogether

  //Performs the given action for each remaining element until all elements are processed
  default void forEachRemaining(Consumer<? super E> action);
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterator Example

```java
List<Integer> myList = new ArrayList<Integer>();
//… operations on myList

Iterator<Integer> listIterator = myList.iterator();

while(listIterator.hasNext()){
  Integer elt = listIterator.next();
  System.out.println(elt);
}
```

forEachRemaining

- Java8 introduced lambda expressions and `Iterator` interface now contains a new method.

```
default void forEachRemaining(Consumer<? super E> action)
```

- Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

```
listIterator.forEachRemaining(s -> System.out.println(s));
```

Iterable Interface

- Interface that allows an object to be the target of a for-each loop:

```
interface Iterable<E>{
  //returns an iterator over elements of type E
  Iterator<E> iterator();

  //Performs the given action for each element of the Iterable until all elements
  //have been processed or the action throws an exception.
  default void forEach(Consumer<? super E> action);

}

public class ArrayList<E> implements Iterable<E>{...}

for(String elt: myList){
  System.out.println(elt);
}
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

forEach

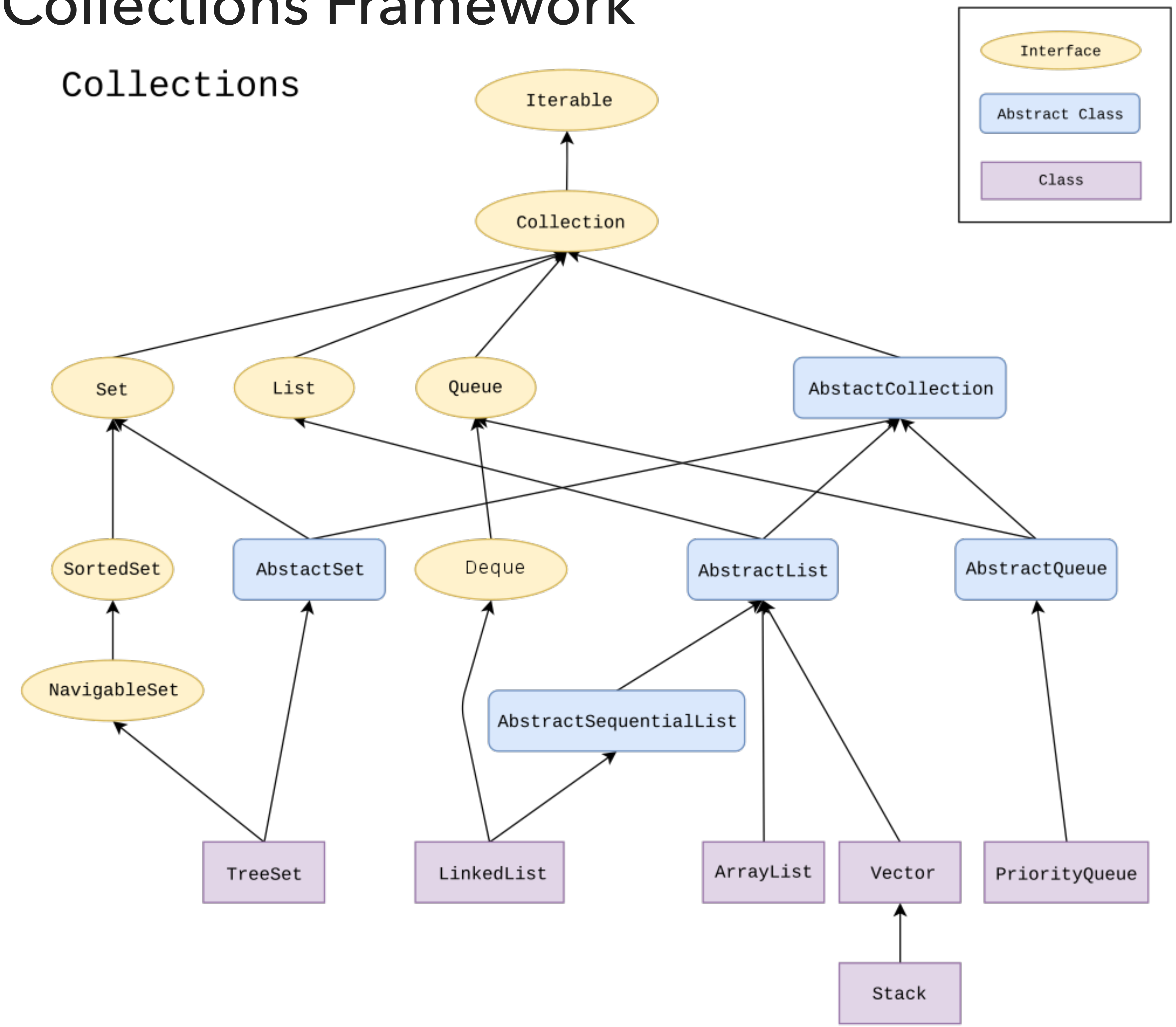
- Java8 introduced lambda expressions and `Iterable` interface now contains a new method.

```
default void forEach(Consumer<? super E> action)
```

- Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

```
myList.forEach(s -> System.out.println(s));
```

The Java Collections Framework



https://en.wikipedia.org/wiki/Java_collections_framework

How to make your data structures iterable?

1. Implement `Iterable` interface.
2. Make a private class that implements the `Iterator` interface.
3. Implement `iterator()` method to return an instance of the private class.

Example: making `ArrayList` iterable

```
public class ArrayList<E> implements List<E>, Iterable<E> {
     //…
     public Iterator<E> iterator() {

          return new ArrayListIterator();
     }


     private class ArrayListIterator implements Iterator<E> {

          private int i = 0;

          public boolean hasNext() {
               return i < size;

          }

          public E next() {

               return data[i++];

          }

          public void remove() {
               throw new UnsupportedOperationException();

          }

     }
```

Traversing ArrayList

- All valid ways to traverse ArrayList and print its elements one by one.

```
 // because it implements the Iterable interface
 for(int elt:myList) {
    System.out.println(elt);
 }

 // because it implements the Iterable interface
myList.forEach(elt -> System.out.println(elt));

 // because it contains a private class that implements the Iterator interface
Iterator<Integer> listIterator = myList.iterator();
while(listIterator.hasNext()){
      Integer elt = listIterator.next();
      System.out.println(elt);
  }

  // because it contains a private class that implements the Iterator interface
Iterator<Integer> listIterator = myList.iterator();
listIterator.forEachRemaining(elt-> System.out.println(elt));
```

PRACTICE TIME - WORKSHEET

A programmer discovers that they frequently need only the odd numbers in an `ArrayList` of `Integers`. As a result, they decided to write a class `OddIterator` that implements the `Iterator` interface. Please help them implement the constructor and the `hasNext()` and `next()` methods so that they can retrieve the odd values, one at a time. For example, if the `ArrayList` is [7, 4, 1, 3, 0], the iterator should return the values 7, 1, and 3.

```
import java.util.*;

public class OddIterator implements Iterator<Integer> {

// The array whose odd values are to be enumerated
private ArrayList<Integer> myArrayList;

//any other instance variables you might need

//An iterator over the odd values of myArrayList
public OddIterator(ArrayList<Integer> myArrayList){

}
//should run in O(n) time
public boolean hasNext(){


}
//should run in O(1) time
public Integer next(){

}
}
```

```
public static void main(String[] args) {
    ArrayList<Integer> myList = new ArrayList<Integer>(Arrays.asList(7, 4, 1, 3, 0));
    OddIterator oi = new OddIterator(myList);
    while(oi.hasNext()){
        System.out.println(oi.next());
    }
}
```

PRACTICE TIME - ANSWER

```
 import java.util.*;

public class OddIterator implements Iterator<Integer> {

// The array whose odd values are to be enumerated
private ArrayList<Integer> myArrayList;

//any other instance variables you might need
int counter;


//An iterator over the odd values of myArrayList
public OddIterator(ArrayList<Integer> myArrayList){
     this.myArrayList = myArrayList;
     counter = 0;
}

//runs in O(n) time
public boolean hasNext(){
      for (int i=counter; i<myArrayList.size(); i++){
          if(myArrayList.get(i)%2 == 1){
               counter = i;
               return true;
          }
     }
     return false;
}



//runs in O(1) time
public Integer next(){
     return myArrayList.get(counter++);
}
}
```

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Selection sort

`Comparable`

- Interface with a single method that we need to implement: `public int compareTo(T that)`
- Implement it so that `v.compareTo(w)`:
  - Returns >0 if v is greater than `w`.
  - Returns <0 if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.
- Corresponds to natural ordering.
- Java classes such as `Integer`, `Double`, `String`, `File` all implement `Comparable`.

Comparator

- Sometimes the natural ordering is not the type of ordering we want.
- Comparator is an interface which allows us to dictate that kind of ordering we want by implementing the method:
  `public int compare(T this, T that)`
- Implement it so that `compare(v, w)`:
  - Returns >0 if v is greater than `w`.
  - Returns <0 if v is smaller than `w`.
  - Returns 0 if v is equal to `w`.

Sorting `Collections`

- Collections class contains a `sort` method:
- `Collections.sort(list)`
  - If collection's elements do not implement the `Comparable`, throws `ClassCastException`.

Alternative Sorting of `Collections`

- `Collections.sort(list, someComparator);`
  - If collection's elements do not implement `Comparable` or cannot be compared with `Comparator`, throw `ClassCastException`.

Example - Employee

```java
public class Employee implements Comparable<Employee> {

    private int id;
    private String name;
    private int salary;

    public Employee(int id, String name, int salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public int compareTo(Employee e) {
        if (this.id < e.id) {
            return -1;
        } else if (this.id > e.id) {
            return 1;
        } else
            return 0;
        // return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));
        // return Integer.compare(this.id, e.id);

    }

    public static Comparator<Employee> nameComparator = new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2) {
            return e1.name.compareTo(e2.name);
        }
    };

    public static Comparator<Employee> salaryComparator(){
        return (Employee e1, Employee e2) -> Integer.compare(e1.salary, e2.salary);
    }

    public String toString() {
        return "Name: " + name + " ID: " + id + " Salary: " + salary;
    }
```

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator

PRACTICE TIME - Worksheet

```java
public static void main(String[] args) {

    Employee e1 = new Employee(5, "Yash", 100000);
    Employee e2 = new Employee(8, "Tharun", 25000);
    Employee e3 = new Employee(4, "Yush", 10000);
    List<Employee> list = new ArrayList<Employee>();
    list.add(e1);
    list.add(e2);
    list.add(e3);

    System.out.println(list);

    Collections.sort(list);
    System.out.println(list);

    Collections.sort(list, Employee.nameComparator);
    System.out.println(list);

    Collections.sort(list, Employee.salaryComparator());
    System.out.println(list);

}
```

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator

PRACTICE TIME - Answer

```java
public static void main(String[] args) {

    Employee e1 = new Employee(5, "Yash", 100000);
    Employee e2 = new Employee(8, "Tharun", 25000);
    Employee e3 = new Employee(4, "Yush", 10000);
    List<Employee> list = new ArrayList<Employee>();
    list.add(e1);
    list.add(e2);
    list.add(e3);

    System.out.println(list);
    //[Name: Yash ID: 5 Salary: 100000, Name: Tharun ID: 8 Salary: 25000, Name: Yush ID: 4 Salary: 10000]

    Collections.sort(list);
    System.out.println(list);
    //[Name: Yush ID: 4 Salary: 10000, Name: Yash ID: 5 Salary: 100000, Name: Tharun ID: 8 Salary: 25000]

    Collections.sort(list, Employee.nameComparator);
    System.out.println(list);
    //[Name: Tharun ID: 8 Salary: 25000, Name: Yash ID: 5 Salary: 100000, Name: Yush ID: 4 Salary: 10000]

    Collections.sort(list, Employee.salaryComparator());
    System.out.println(list);
    //[Name: Yush ID: 4 Salary: 10000, Name: Tharun ID: 8 Salary: 25000, Name: Yash ID: 5 Salary: 100000]

}
```

https://stackoverflow.com/questions/2266827/when-to-use-comparable-and-comparator

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Sorting
- Selection sort

Why study sorting?

- It's more common than you think: e.g., sorting flights by price, contacts by last name, files by size, emails by day sent, neighborhoods by zipcode, etc.
- Good example of how to compare the performance of different algorithms for the same problem.
- Some sorting algorithms relate to data structures.
- Sorting your data will often be a good starting point when solving other problems (keep that in mind for interviews).

How many different algorithms for sorting can there be?

- Adaptive heapsort
- Bitonic sorter
- Block sort
- Bubble sort
- Bucket sort
- Cascade mergesort
- Cocktail sort
- Comb sort
- Flashsort
- Gnome sort
- **Heapsort**
- **Insertion sort**
- Library sort
- **Mergesort**
- Odd-even sort
- Pancake sort
- **Quicksort**
- Radixsort
- **Selection sort**
- Shell sort
- Spaghetti sort
- Treesort
- …

Definitions

- Sorting: the process of arranging n elements of a collection in non-decreasing order (e.g., numerically or alphabetically).
- Key: assuming that an element consists of multiple components, the key is the property based on which we sort elements.
  - Examples: elements could be books and potential keys are the title or the author which can be sorted alphabetically or the ISBN which can be sorted numerically.
- Let's say we want to sort an array of objects of type T.
- Our class `T` should implement the `Comparable` interface and we will need to implement the `compareTo` method.

Two useful abstractions

- We will refer to data only through comparisons and exchanges.
- Comparisons: Is `v` less than `w`?

```
v.compareTo(w) < 0;
```

- Exchanges: swap element in array `a[]` at index `i` with the one at index `j`.

```
T temp = a[i];
a[i]=a[j];
a[j]=temp;
```

Rules of the game - Cost model

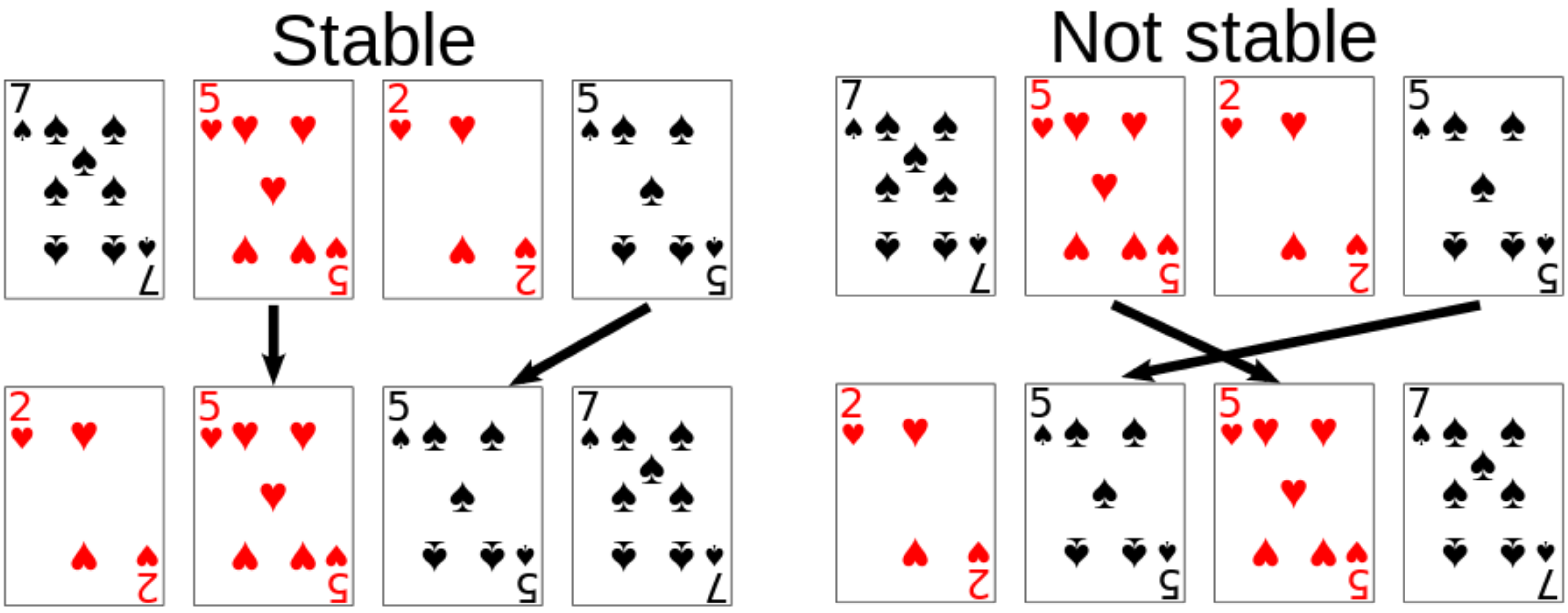
- Sorting cost model: we count compares and exchanges. If a sorting algorithm does not use exchanges, we count array accesses.
- There are other types of sorting algorithms where they are not based on comparisons (e.g., radixsort). We will not see these in CS62 but stay tuned for CS140.

Rules of the game - Memory usage

- Extra memory: often as important as running time. Sorting algorithms are divided into two categories:
  - In place: use constant or logarithmic extra memory, beyond the memory needed to store the elements to be sorted.
  - Not in place: use linear auxiliary memory.

Rules of the game - Stability

- Stable: sorting algorithms that sort repeated elements in the same order that they appear in the input.



https://en.wikipedia.org/wiki/Sorting_algorithm#/media/File:Sorting_stability_playing_cards.svg

Lecture 12: Iterators, Comparators, Selection Sort

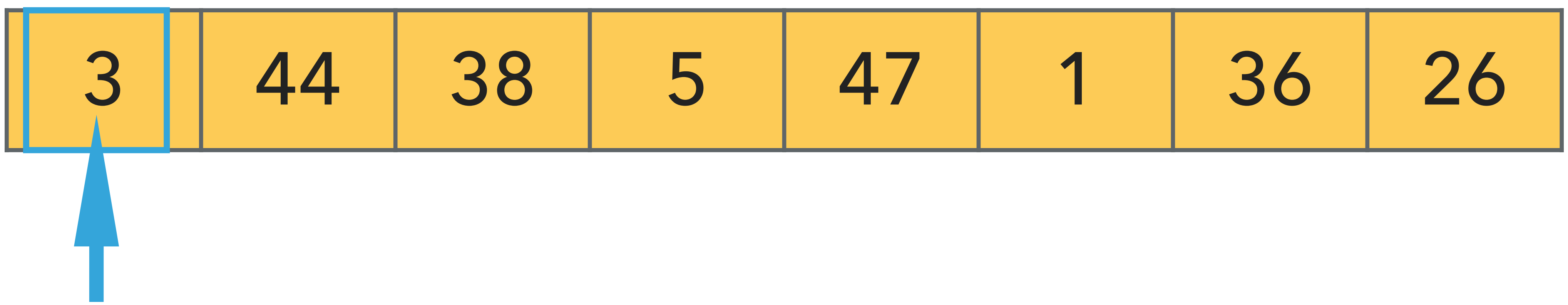
- Iterators
- Comparators
- Sorting
- Selection sort

Selection sort

3	44	38	5	47	1	36	26

- Divide the array in two parts: a sorted subarray on the left and an unsorted on the right.
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	44	38	5	47	3	36	26

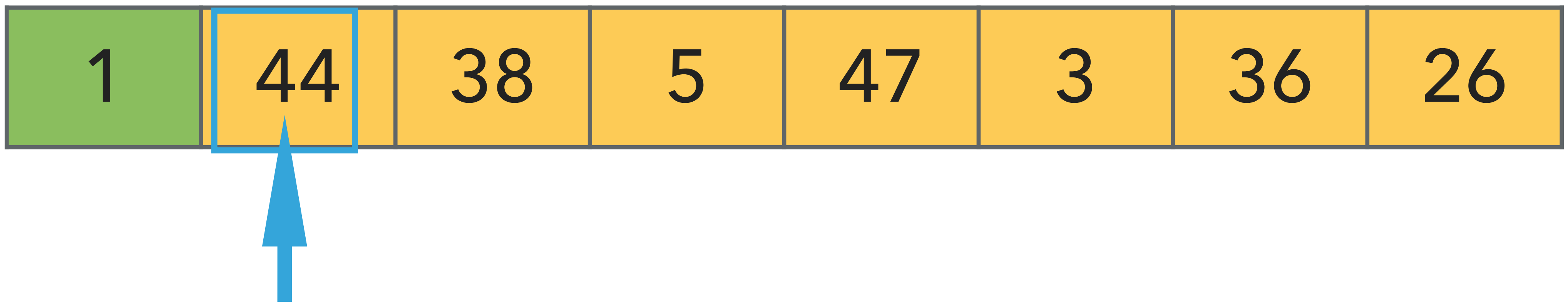
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	44	38	5	47	3	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

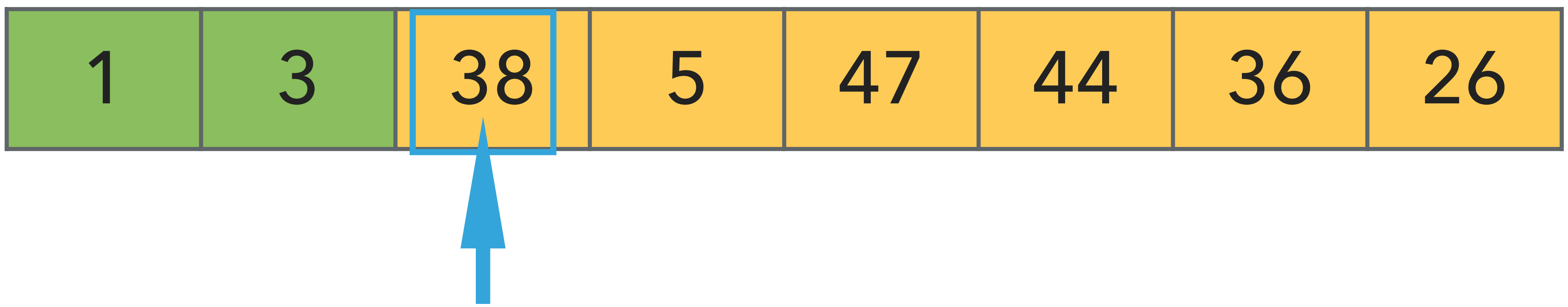
1	3	38	5	47	44	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	38	47	44	36	26

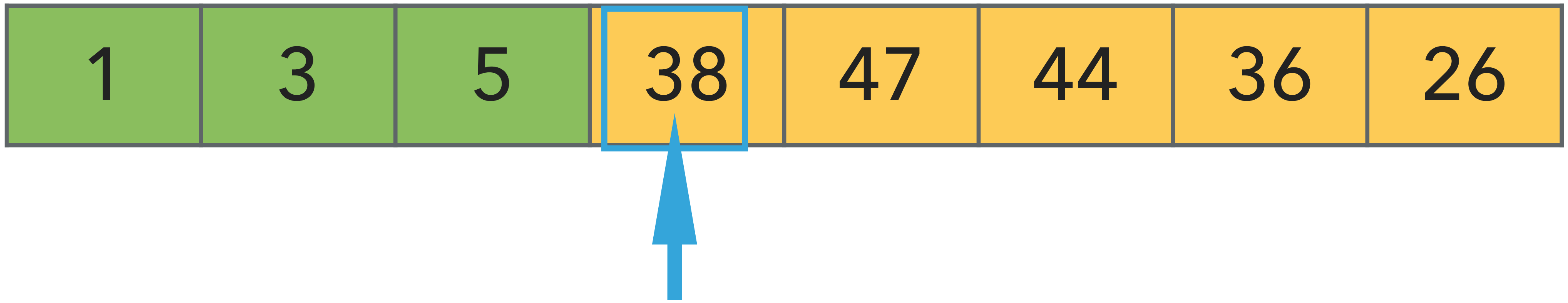
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	38	47	44	36	26

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	47	44	36	38

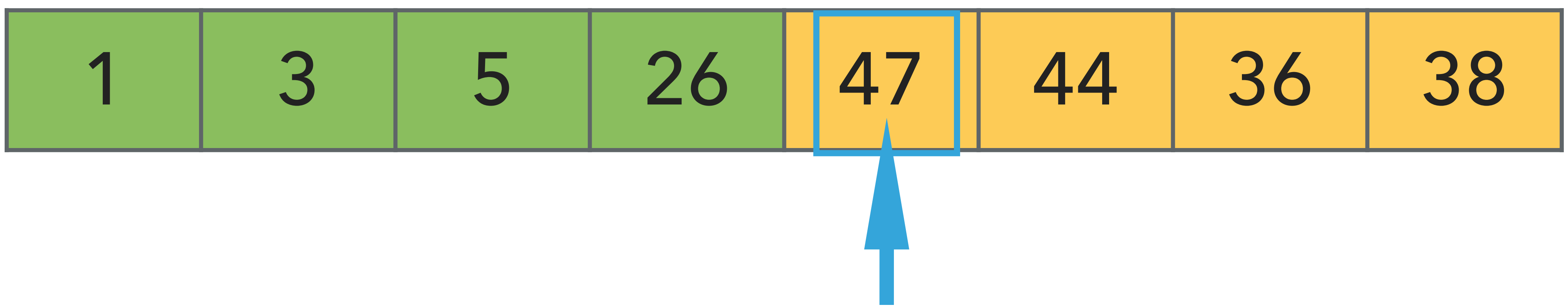
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	47	44	36	38

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	44	47	38

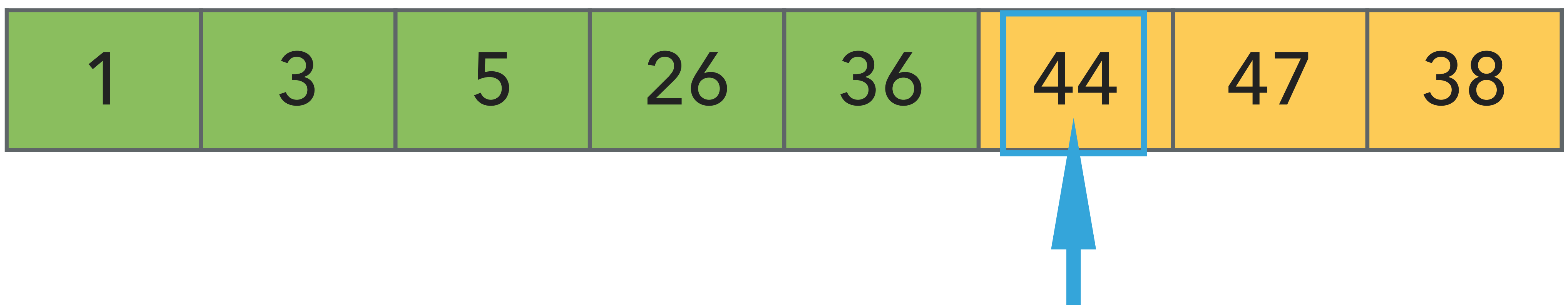
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	44	47	38

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	47	44

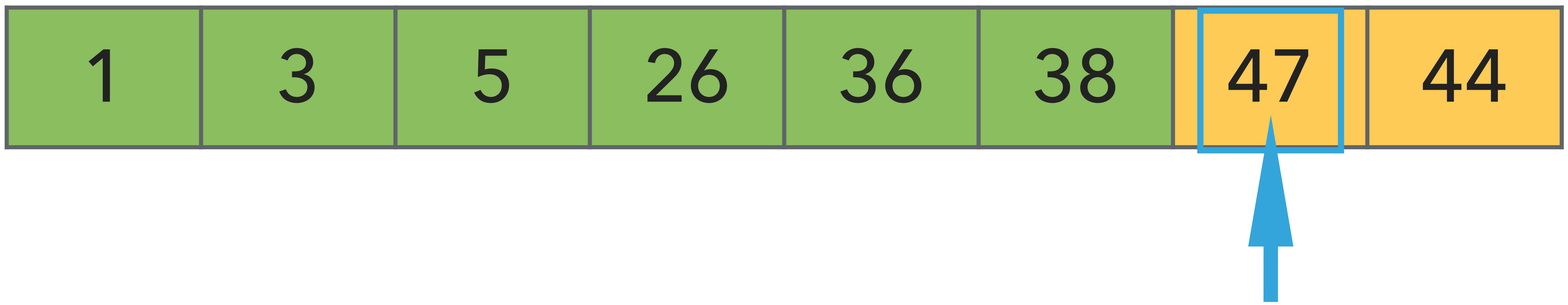
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	47	44

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

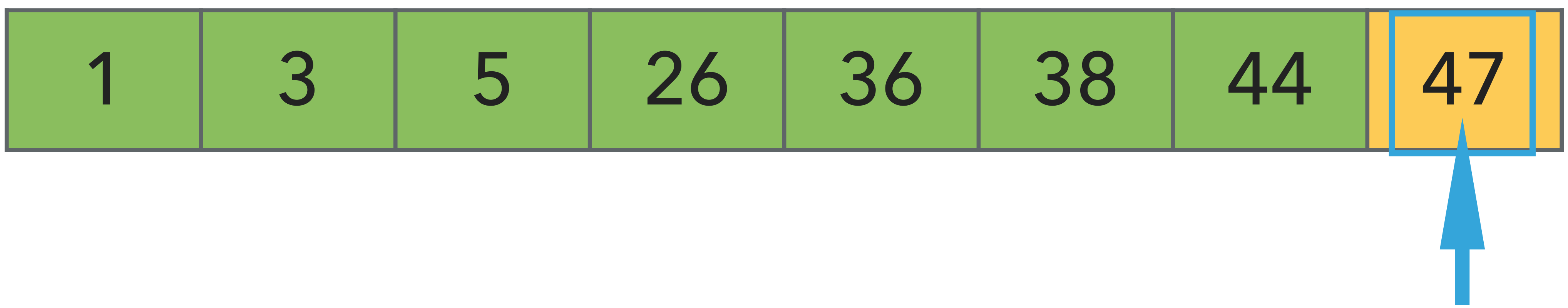
- Repeat:
    - Find the smallest element in the unsorted subarray.
    - Exchange it with the leftmost unsorted element.
    - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

1	3	5	26	36	38	44	47

- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

http://algs4.cs.princeton.edu

2.1 SELECTION SORT DEMO

https://algs4.cs.princeton.edu/lectures/demo/21DemoSelectionSort.mov

PRACTICE TIME - Implement Selection sort

```
public static <E extends Comparable<E>> void selectionSort(E[] a)
{

}
```

Selection sort

```
public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (a[j].compareTo(a[min])<0){
                    min = j;
                }
            }
            E temp = a[i];
            a[i]=a[min];
            a[min]=temp;
        }
    }
```

← At iteration i

← Find the index min of the smallest remaining array

← swap a[i] and a[min]

- Invariants: At the end of each iteration i:
    - the array a is sorted in ascending order for the first i+1 elements a[0…i]
    - no entry in a[i+1…n-1] is smaller than any entry in a[0…i]

Selection sort: mathematical analysis for worst-case

```
public static <E extends Comparable<E>> void selectionSort(E[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (a[j].compareTo(a[min])<0){
                    min = j;
                }
            }
            E temp = a[i];
            a[i]=a[min];
            a[min]=temp;
        }
    }
```

- Comparisons: $1 + 2 + \ldots + (n-2) + (n-1) \sim n^2/2$, that is $O(n^2)$.
- Exchanges: n or $O(n)$, making it useful when exchanges are expensive.
- Running time is quadratic, even if input is sorted.
- In-place, requires almost no additional memory.
- Not stable, think of the array [5_a, 3, 5_b, 1] which will end up as [1, 3, 5_b, 5_a].

Practice Time - Worksheet

- Using selection sort, sort the array with elements [12,10,16,11,9,7].
- Visualize your work for every iteration of the algorithm.

SELECTION SORT

Answer

https://subscription.packtpub.com/book/application_development/9781785888731/13/ch13lvl1sec89/selection-sort

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Selection sort

Readings:

- Recommended Textbook:
  - Chapter 2.1 (pages 244-262)
  - Chapter 2.5 (Pages 338-339)
- Recommended Textbook Website:
  - Elementary sorts: https://algs4.cs.princeton.edu/21elementary/
- Oracle Documentation:
  - Comparable: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
  - Comparator: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Code

- Lecture 12 code

Worksheet

- Lecture 12 worksheet

Practice Problem 1 - Recommended textbook 2.1.1

- Show all the steps of how selection sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each exchange.

a[]

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Practice Problem 2 - Recommended textbook 2.1.2

- What is the maximum number of exchanges involving any particular element during selection sort? What is the average number of exchanges involving one specific element x?

Practice Problem 3 - Recommended textbook 2.1.3

- Give an example of an array of n elements that maximizes the number of times the test `a[j].compareTo(a[min])<0` succeeds (and, therefore, `min` gets updated) during the operation of selection sort.

ANSWER 1

- Show all the steps of how selection sort would sort [E, A, S, Y, Q, U, E, S, T, I, O, N] in the style of the following trace which visualizes the array contents just after each exchange.

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		E	A	S	Y	Q	U	E	S	T	I	O	N
0	1	E	A	S	Y	Q	U	E	S	T	I	O	N
1	1	A	E	S	Y	Q	U	E	S	T	I	O	N
2	6	A	E	S	Y	Q	U	E	S	T	I	O	N
3	9	A	E	E	Y	Q	U	S	S	T	I	O	N
4	11	A	E	E	I	Q	U	S	S	T	Y	O	N
5	10	A	E	E	I	N	U	S	S	T	Y	O	Q
6	11	A	E	E	I	N	O	S	S	T	Y	U	Q
7	7	A	E	E	I	N	O	Q	S	T	Y	U	S
8	11	A	E	E	I	N	O	Q	S	T	Y	U	S
9	11	A	E	E	I	N	O	Q	S	S	Y	U	T
10	10	A	E	E	I	N	O	Q	S	S	T	U	Y
11	11	A	E	E	I	N	O	Q	S	S	T	U	Y
		A	E	E	I	N	O	Q	S	S	T	U	Y

ANSWER 2

- What is the maximum number of exchanges involving any particular element during selection sort? What is the average number of exchanges involving one specific element x?

- The maximum number of exchanges is n. See the example below:

		a[ ]											
i	min	0	1	2	3	4	5	6	7	8	9	10	11
		Z	A	B	C	D	E	F	G	H	I	J	K
0	1	Z	A	B	C	D	E	F	G	H	I	J	K
1	2	A	Z	B	C	D	E	F	G	H	I	J	K
2	3	A	B	Z	C	D	E	F	G	H	I	J	K
3	4	A	B	C	Z	D	E	F	G	H	I	J	K
4	5	A	B	C	D	Z	E	F	G	H	I	J	K
5	6	A	B	C	D	E	Z	F	G	H	I	J	K
6	7	A	B	C	D	E	F	Z	G	H	I	J	K
7	8	A	B	C	D	E	F	G	Z	H	I	J	K
8	9	A	B	C	D	E	F	G	H	Z	I	J	K
9	10	A	B	C	D	E	F	G	H	I	Z	J	K
10	11	A	B	C	D	E	F	G	H	I	J	Z	K
11	11	A	B	C	D	E	F	G	H	I	J	K	Z
		A	B	C	D	E	F	G	H	I	J	K	Z

- The average number of exchanges for a specific element is exactly 2, because there are exactly n exchanges and n items (and each exchange involves two items).

ANSWER 3

- Give an example of an array of n elements that maximizes the number of times the test `a[j].compareTo(a[min])<0` succeeds (and, therefore, `min` gets updated) during the operation of selection sort.

- Any array in reverse order would do, for example, [6, 5, 4, 3, 2, 1].