SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

12: Iterators, Comparators, Selection Sort

Alexandra Papoutsaki
she/her/hers

Today we are starting a new chapter in our class, which will take us into sorting data structures.

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Sorting
- Selection sort

Some slides adopted from Algorithms 4th Edition or COS226

To talk about sorting, we will first talk about how to iterate through a data structure since we will be looking at the contents of unsorted or sorted data structures all the time.

`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

To do that, we will use the Iterator interface. If we choose to implement it, we make the promise of implementing two methods: hasNext and next. hasNext, returns true if there are more elements in the collection that the iterator can take us through, that is, it tells us whether next would return an element instead of throwing an exception. next is responsible for advancing the iterator to the next value. You will notice that there is a default method called remove. My advice is to better avoid it because removing elements while iterating through a data structure is a recipe for a disaster. And finally there is a default method forEachRemaining that we will see in a minute.

Iterator Example

```
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);
}
```

Let's assume we have myList, an ArrayList of Integers. The ArrayList class has a method called iterator that returns an iterator over the elements in this list in proper sequence. We can use this iterator in this typically structured while loop. The condition the while loop will check is that the iterator has more elements to iterate through by calling the hasNext method. And the next method will actually return the next element in the array list. So this piece of code, iterates through all the elements in the array list and prints them one by one.

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));
```

Java version 8 brought significant changes, one of which is that it introduced lambda expressions which you might remember from cs54. For the Iterator interface, that translated to the introduction of a default method called forEachRemaining. forEachRemaining takes one argument which is an action written as a lambda expression and applies to every single element. So for example, instead of needing the while loop we had before, we can now write in a single line
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

Now let's talk about a related interface called Iterable. The Iterable interface requires us to implement a method called iterator that returns an object of type Iterator that allows us to iterate over elements of type E. It also contains a default method called forEach.

If a data structure implements the Iterable interface, it can be passed in a for-each loop, e.g., for(String elt: myList){
  System.out.println(elt);
}

forEach

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

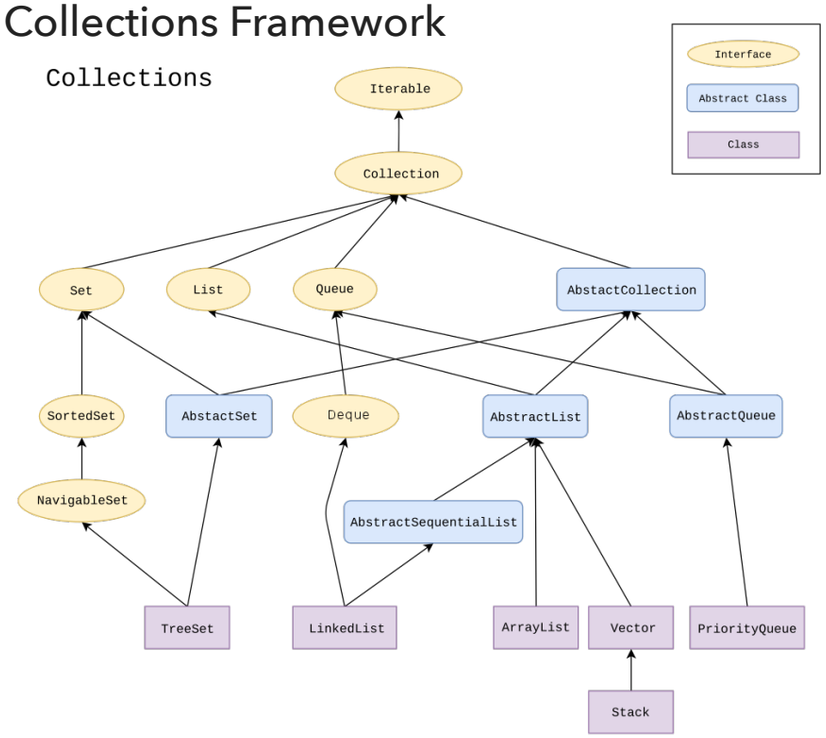
```
default void forEach(Consumer<? super E> action)
```

- 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));
```

Similarly with how the Iterator interface changed with Java 8, so did the Iterable interface which now includes a default method called forEach. Instead of writing a for-each loop, you can use it directly, e.g.,
myList.forEach(s -> System.out.println(s));

The Java Collections Framework



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

If we look at the Java Collections Framework, we will notice that all the data structures we have seen so far implement interfaces that extend the interface Collection which in turn extends the interface Iterable. That means that all collections that are built-in Java are iterable.

ITERATORS

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.

So it seems like the java.util.ArrayList class and all java collections are iterable. How can we make any data structure we implement iterable? We start by making our class implement the Iterable interface. Internally, we will make a private class that will implement the Iterator interface. Remember, the Iterable interface require us to implement the method iterator which returns an object of type Iterator. What we will do is we will implement it so that it returns an instance of the private class that implements the Iterator interface.

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();

          }

     }
```

Here is an example of making the class ArrayList we defined as iterable. We will say that the class will implement the Iterable interface and we will have now to implement the method iterator. What should it return? We are missing one thing here: we need to make a private inner class that implements Iterator. Let's call that class ArrayListIterator. Since it implements Iterator, it has to implement the methods hasNext and next. How do these work? We assume that the iterator will start at index 0 so hasNext will check that our counter is smaller than size, and next will just return the next element AND increase the counter of the index we are in. Overall, the iterator method will return an instance of the ArrayListIterator inner class.

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));
```

Here are multiple ways you can traverse an arrayList and print its elements. You can use a for each loop because it implements the Iterable interface. Similarly, you can use the forEach method and a lambda expression. Or you can work directly with the iterator that is returned from the iterator method and thus access the hasNext and next methods. Or you can use the forEachRemaining method and lambda expressions since it implements the Iterator interface.

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());
    }
}
```

Let's put what we learned about iterators in practice by solving the following problem. 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. Look at the main to see how the class would be used.

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++);
}
}
```

This is how you could use implement the OddIterator class.

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Selection sort

Now that we talked about how to iterate through a data structure, let's talk about how we can compare elements in a data structure. that will also be essential in sorting data structures.

COMPARATORS

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`.

We will use an interface called Comparable that forces us to implement the method compareTo. The convention is that this method returns a positive if we call v.compareTo(w) and v is greater than w, a negative number if v is smaller than w, and 0 if it is equal to w.
Essentially, this interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.
Java classes such as Integer, Double, String, File all implement Comparable.

COMPARATORS

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.

Similarly to how the Iterator and Iterable interfaces are related, the Comparable interface is related to another interface called Comparator. Why would we need another interface? 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 compare. The convention is that this method returns a positive if we call compare(v, w) and v is greater than w, a negative number if v is smaller than w, and 0 if it 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`.

The Collections class that Java offers has a sort method which if we pass to it a data structure, let's say a list, it will sort it for us! The only caveat is the class needs to implement the Comparable interface, otherwise we will get an exception.

Alternative Sorting of `Collections`

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

It also contains an overloaded version of sort that allows us to sort a collection by comparing its elements using an alternative comparator. If the collection's elements do not implement Comparable or cannot be compared with Comparator, throw an exception.

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

Let's put everything about comparators together through this example. Here, we have a class Employee that implements the interface Comparable. An Employee is defined by three instance variables: their id, name, and salary. Since we implement the Comparable interface, we have to implement the compareTo method. A logical way of comparing two employees is by their ids. In the body of the compareTo method, you can see three ways you can implement this comparison. You can do it explicitly with an if-else if-else statement. Or you can take advantage of the fact that the class Integer has a compareTo and compare method. Keep in mind, that if you use the compareTo method, you need to convert the int primitives to their wrapper objects Integer.
Next, you will see two alternative comparators. The syntax is a bit unique, but both implement the compare method by comparing two Employee objects. You can choose whichever you prefer. The second one is more modern syntax. And we also have a toString method.

PRACTICE TIME - Worksheet

```
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

Based on what we discussed, what do you think the following main method will print?

PRACTICE TIME - Answer

```
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

The first one is the unsorted list as elements were added. The second is sorted based on id (natural comparator). The second one is sorted in lexicographic order by name and the third one in increasing order of salary.

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Sorting
- Selection sort

We are now ready to start talking about sorting. But why study sorting?

SORTING

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).

In the early CS days, common wisdom was that up to 30% of all computing cycles was spent sorting. Sorting is cheaper today but remains important. It has applications in transaction processing, combinatorial optimization, astrophysics, genomics, linguistics, weather prediction etc. Think of sorting flights by price, contacts by last name, files by size, emails by day sent, neighborhoods by zipcode, etc.
Pedagogically, it is also a great vehicle to learn how to compare the performance of different algorithms for the same problem. Additionally, some sorting algorithms relate to data structures, like heap sort uses heaps. Finally, sorting your data will often be a good starting point when solving other problems (keep that in mind for interviews).

SORTING

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
- ...

You might be wondering how many different algorithms for sorting can there be? A TON! There are decades of research on coming up with different sorting algorithms. The ones in bold are the ones we will examine in this class.

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.

Let's establish some vocabulary. By sorting, we mean the process of arranging n elements of a collection in non-decreasing order (e.g., numerically or alphabetically).
Why non-decreasing and not increasing order? Increasing means that every element is greater than the one before it. Non-decreasing means that no element is less than the element before it, or in other words: that every element is greater than OR equal to the one before it.
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 or clothes and stars versus price versus relevance in an e-commerce store.
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;
```

We will use two useful abstractions. We will refer to data only through comparisons and exchanges. Comparisons will take the form of is v less than w which we will accomplish by calling v.compareTo(w) < 0;
Exchanges will result to swapping an element in an array at index i with 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.

Our cost model for sorting will focus on counting compares and exchanges. If a sorting algorithm does not use exchanges, we will 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.

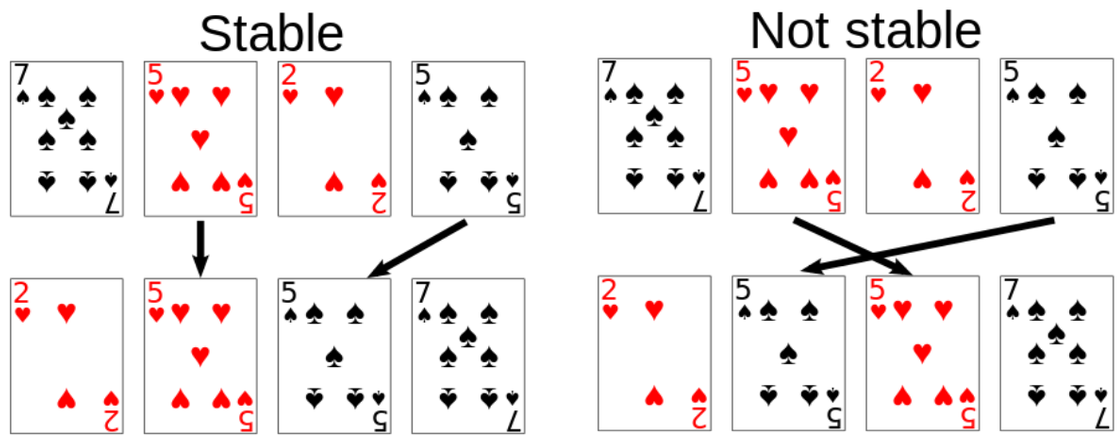
Rules of the game - Memory usage

- 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.

An important consideration when examining a sorting algorithm is its memory usage which is often as important as the 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

A final definition will have to do with whether a sorting algorithm is stable. Stable sorting algorithms sort repeated elements in the same order that they appear in the input. For example, if we sort a deck of cards by the number and we have two fives, a stable algorithm guarantees that the 5 of hearts which we encounter first in the unsorted array will be before the 5 of spades in the sorted array. A sorting algorithm that is not stable makes no such guarantees.

Lecture 12: Iterators, Comparators, Selection Sort

- Iterators
- Comparators
- Sorting
- Selection sort

And with that, we are ready to see our very first sorting algorithm, selection sort!

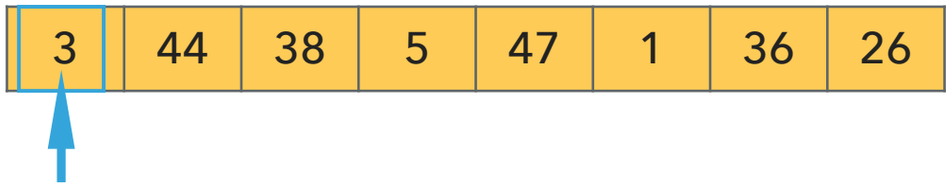
SELECTION SORT

Selection sort

- 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.

The basic premise of selection sort is:
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.
In the subsequent slides, you will see an illustration of how the selection sort algorithm plays out.

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.

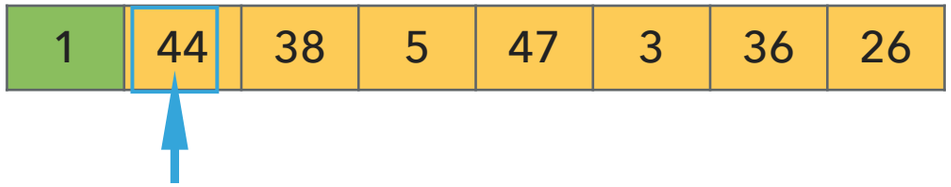
Selection sort

- 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

- 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.

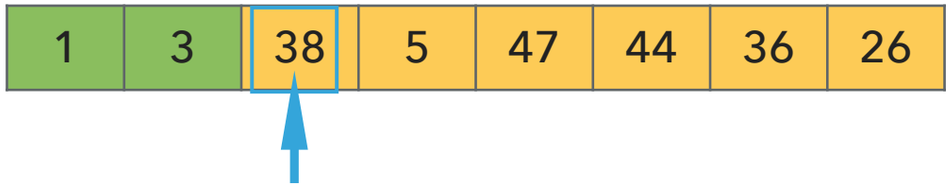
Selection sort

- 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

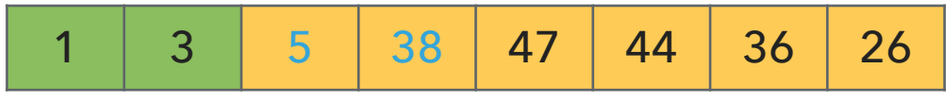
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

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

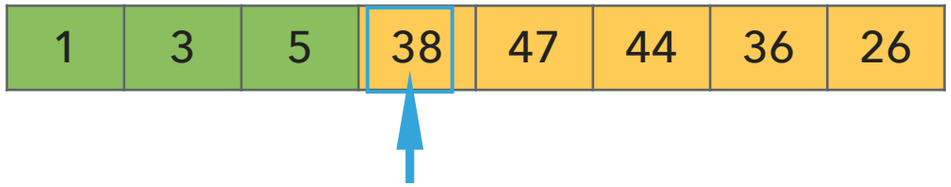


- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort

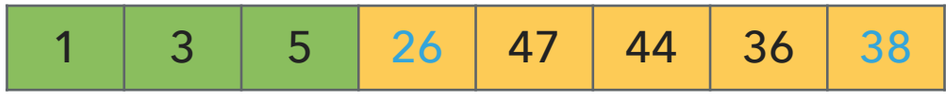
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



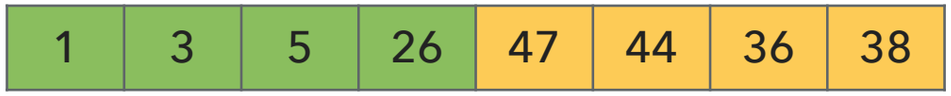
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



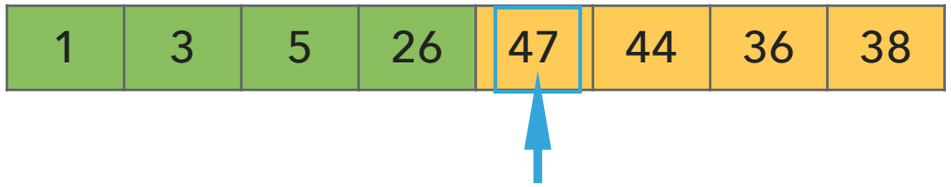
- Repeat:
  - Find the smallest element in the unsorted subarray.
  - Exchange it with the leftmost unsorted element.
  - Move subarray boundaries one element to the right.

Selection sort



- 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.

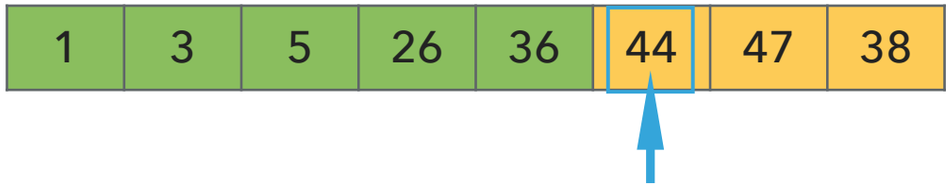
Selection sort

- 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

- 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.

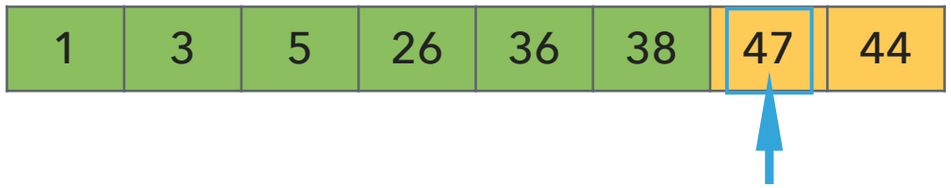
Selection sort

- 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

- 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.

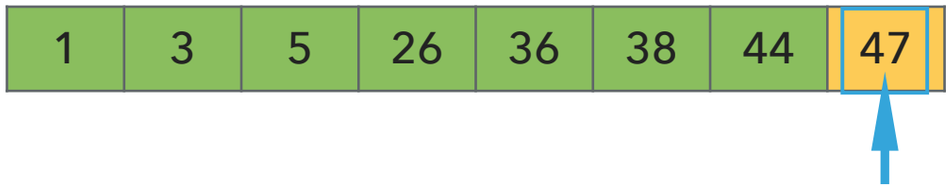
Selection sort

- 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

- 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

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

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

If you want to watch a video, you can click at the link or download this keynote presentation.

PRACTICE TIME - Implement Selection sort

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









}
```

How would you go about implementing selection sort in Java? Here's the signature of the method you should work with. It makes sense for the method to be static as we will just pass an array to it. The array will hold objects of a class that implements the Comparable interface. Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed. The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

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]`

To implement selectionSort, we will use a nested for loop. What should be true at the end of each iteration i is that 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].

Let's think about the running time and memory usage.
In terms of comparisons (calls to compareTo), we will have O(n^2) (see below)
Exchanges will be linear, making it useful when this is an expensive operation.
Unfortunately, running time is quadratic even if the array we are given is already sorted, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.
It is an in-place algorithm since it requires almost no additional memory beyond a handful local variables.
And it is not stable. Think of the array [5_a, 3, 5_b, 1] which will end up as [1, 3, 5_b, 5_a].

For i=0 make n-1 comparisons
For i = 1 make n-2 comparisons
…
For i=n-1 make 0 comparisons
Total: 1+2+…+n-2+n-1 = n(n-1)/2~1/2n^2=O(n^2)

SELECTION SORT

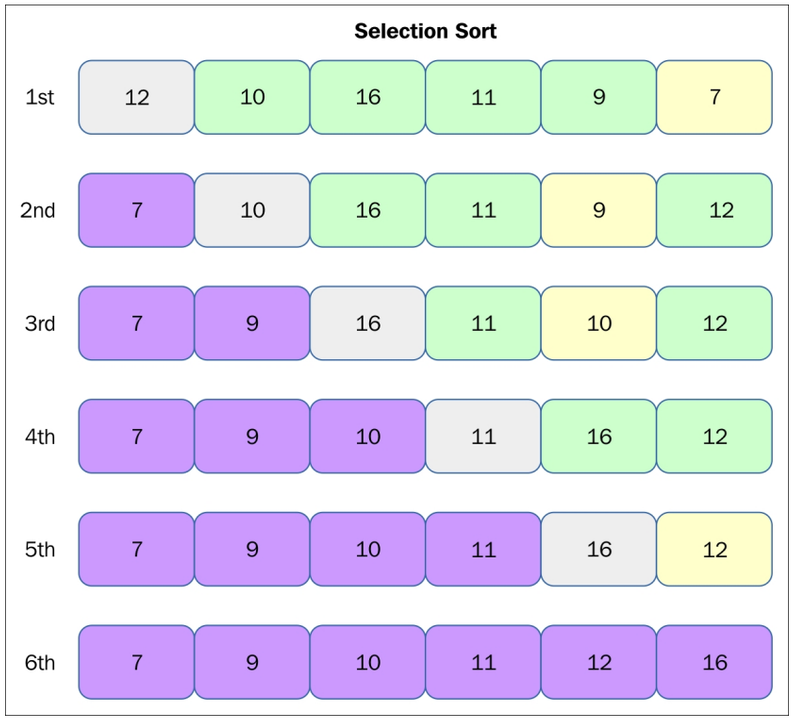
Practice Time - Worksheet

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

Let's make sure we know how to run selection sort.

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

To recap, today we talked about iterators, comparators, sorting, and selection sort. Iterators allow us to iterate through a data structure. To make a data structure iterable, that is the target of a for-each loop, we need to implement the interface Iterable. That ensures we implement the method iterator which needs to return an instance of a class that implements the Iterator interface. The Iterator interface forces us to implement the methods next and hasNext. It's common to see an inner private class within a data structure that implements the Iterator interface. Iterators are useful in allowing us to define how we should traverse a data structure. E.g., it might be a typical traversal in a linear fashion, or we might choose to skip certain elements, or return only certain elements, or go reversely etc. Both Iteravle and Iterator the methods forEach and forEachRemaining, respectively, which allow us to utilize lambda expressions.

Comparators allow us to compare instances of the same class. The Comparable interface defines the natural ordering through the compareTo method. If we want alternative comparisons, we can use the Comparator interface through the compare method. This is often implemented through lambda expressions. If we have a data structure that holds objects of a class that implements the Comparable interface (and for that matter, the Comparator, too), we can pass it to the Collections.sort method which sort it on the spot (it's a void method)

We then talked about sorting which is both a very common strategy in programming and a useful pedagogical tool in allowing us to better understand trade-offs in running time and memory usage for the same problem and established some ground rules. Sorting orders a collection of n elements based on a key (typically in non-decreasing order). Our cost model focuses on number of comparisons and exchanges (or array accesses). We also care about whether a sorting algorithm is in place or requires extra, auxiliary memory. And we care about whether it is stable, that is it maintains the relative order of the items with equal keys.

Finally, we saw the selection sort method, a simple sorting algorithm that has quadratic running time for comparisons, linear time for exchanges, is in place, but is not stable. Running time for selection sort is insensitive to input; it will take about the same time to sort an array that is already in order and one that is completely randomly ordered. Another key characteristic of selection sort is that the number of exchanges is minimal, only n, which is quite unique compared to the rest of the sorting

algorithms we will see.

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.

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].