Sorting - Revisited

With First-Order Functions

Lecture Question

Question: In a package named "functions" create an **object** named Generics with a method named mapFilter that:

- Takes a type parameter T
- As parameters takes
  - A Map of Ints to T's
  - A function that takes an Int and returns a Boolean
- Returns a List of T's
  - The returned List will contain only the values in the input map that are mapped to by keys for which the input function returns true
  - eg. We can think of the input function as a filter that decides which Map values will be added to the return list. If the filter returns true on a key, they value the key maps to will be in the returned list
  - The order of the output List is not defined in this problem (You should be sorting lists in your testing)

Testing: In a package named "tests" create a class named "TestFilterMap" as a test suite that tests all the functionality listed above

Sorting

Order elements in a data structure according to a comparator function

Sorting in Scala

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sorted
println(numbersSorted)
```

List(-23, -8, -4, 5, 7, 10)

Sorting in Scala

- The sorted method returns a new List containing the same elements as the original, but in sorted order
- Integer values have a default comparator
  - Less than function
  - If an element is less than another element, it must be placed before the other element

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sorted
println(numbersSorted)
```

List(-23, -8, -4, 5, 7, 10)

Custom Sorting in Scala

- Sorting a list by the result of a function/method
- Calls the provided function/method on each element and sorts by the default ordering of the returned values

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by the result of a method (like setting the key in Python sorting)
val numbersSorted = numbers.sortBy(Math.abs)
println(numbersSorted)
```

List(-4, 5, 7, -8, 10, -23)

Custom Sorting in Scala

- Uses first-order functions/methods
- We just passed a method as an argument of another method
  - Yes, you can do that!
  - And you will do this often over the few weeks..

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by the result of a method (like setting the key in Python sorting)
val numbersSorted = numbers.sortBy(Math.abs)
println(numbersSorted)
```

List(-4, 5, 7, -8, 10, -23)

Custom Sorting in Scala

- Passing a function/method allows us to use the default sorting order with a computed value

- What if we don't want to sort by the default ordering?
  - Ex. Sort ints by decreasing order

Custom Sorting in Scala

- Sorting a list using a comparator function/method
- The comparator takes two values of the type being sorted
  - Return true if the first parameter should come before the second in the sorted order
  - Return false otherwise (including ties)

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sortWith((a: Int, b: Int) => a > b)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

List(10, 7, 5, -4, -8, -23)

Custom Sorting in Scala

- This is a first-order function
- Provide the Parameter list and the body of the function
- For sortWith, write a function that:
  - Takes 2 parameters matching the type of the List being sorted
  - Returns a Boolean

```
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = numbers.sortWith((a: Int, b: Int) => a > b)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

List(10, 7, 5, -4, -8, -23)

Custom Sorting in Scala

- Alternate setup
- We can create the function and store it in a variable
  - Type is (Int, Int) => Boolean
- First-order functions are just values!
  - Can be stored in variables, passed as arguments, returned from methods

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by a comparator function/method. This function sorts in decreasing order
val comparator: (Int, Int) => Boolean = (a: Int, b: Int) => a > b
val numbersSorted = numbers.sortWith(comparator)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

List(10, 7, 5, -4, -8, -23)

First-Order Functions

- This is the entire definition of a first-order function
  - Creates an object of type function and returns its reference
  - Parameter list in parentheses using usual syntax
  - Use => to separate the parameter list from the body of the function
  - Code that follows is the body of the function

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by a comparator function/method. This function sorts in decreasing order
val comparator: (Int, Int) => Boolean = (a: Int, b: Int) => a > b
val numbersSorted = numbers.sortWith(comparator)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

First-Order Functions

- Can use the usual code block syntax with {}
  - Use this syntax if you want more than 1 line of code in your function

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by a comparator function/method. This function sorts in decreasing order
val comparator: (Int, Int) => Boolean = (a: Int, b: Int) => {
  a > b
}
val numbersSorted = numbers.sortWith(comparator)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

First-Order Functions

- This is the type of the function
  - Types of the parameters in parentheses
  - Use => to separate the parameter types from the return type
  - Then the return type

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by a comparator function/method. This function sorts in decreasing order
val comparator: (Int, Int) => Boolean = (a: Int, b: Int) => a > b
val numbersSorted = numbers.sortWith(comparator)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

First-Order Functions

- A function is a value with a type
  - A function is an object stored on the heap
- Can be used just like any other type

```
val numbers = List(5, -23, -8, 7, -4, 10)
// sort by a comparator function/method. This function sorts in decreasing order
val comparator: (Int, Int) => Boolean = (a: Int, b: Int) => a > b
val numbersSorted = numbers.sortWith(comparator)
// can be shortened to - numbers.sortWith(_ > _)
println(numbersSorted)
```

First-Order Functions

- First-order functions in calculator
  - All operations take 2 Doubles and return a Double
  - Can store operations in a variable
  - Can reduce the number of states and complexity of your Calculator

```
var operation: (Double, Double) => Double = (x: Double, y: Double) => x * y
```

Sorting in Scala

- Sorting a list using a comparator **method**
- Can sort custom types with custom methods
  - Pass methods by name just like passing a variable storing a function
- There's no stopping the ways you can sort!

```
def compareAnimals(a1: Animal, a2: Animal): Boolean = {
  a1.name.toLowerCase() < a2.name.toLowerCase()
}
```

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = animals.sortWith(compareAnimals)
println(animalsSorted)
```

List(Finn, Garfield, morris, Snoopy)

But how does it work?

Selection Sort

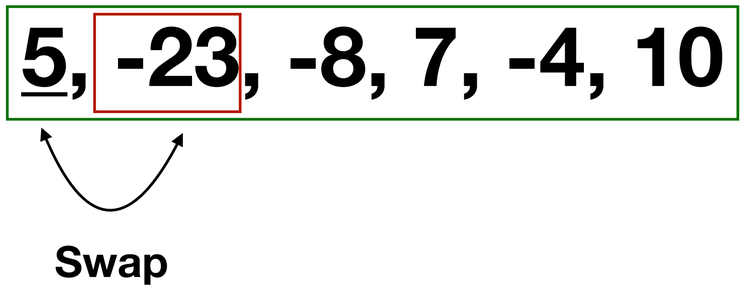
- Iterate over the indices of a list
  - For each index, select the element that belongs there in the final sorted order
  - Swap the current value with the correct one

Given: **5, -23, -8, 7, -4, 10**

Correct Order: **-23, -8, -4, 5, 7, 10**

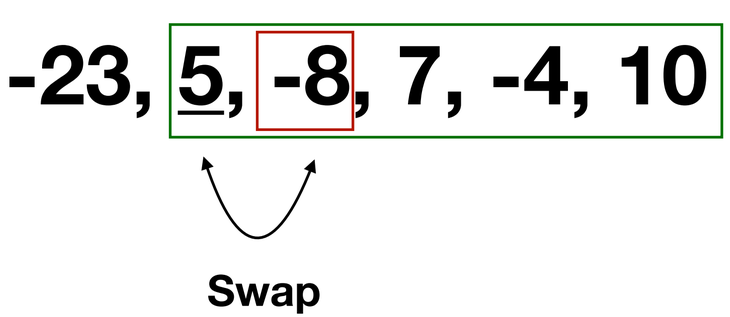
Selection Sort

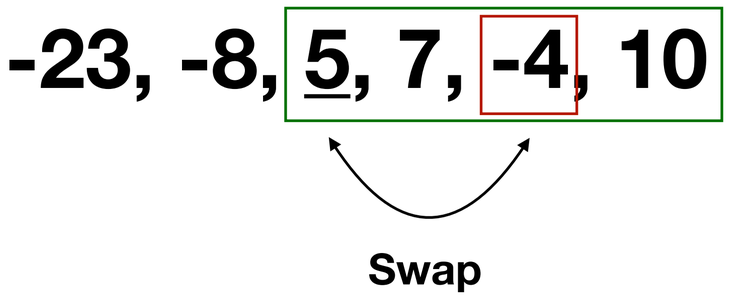
- Start with the first index
- Find the element that belongs there by taking the min of all values
- Swap the values
- Don't have to recheck elements that are already at the correct index

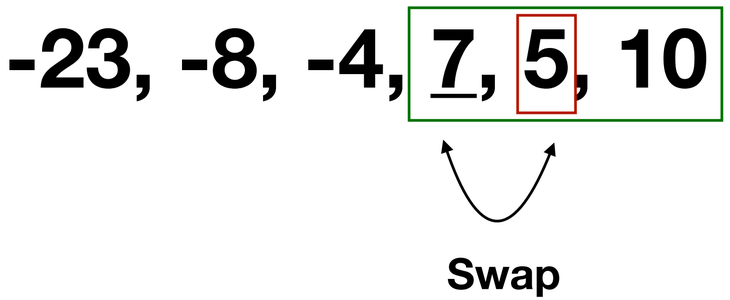


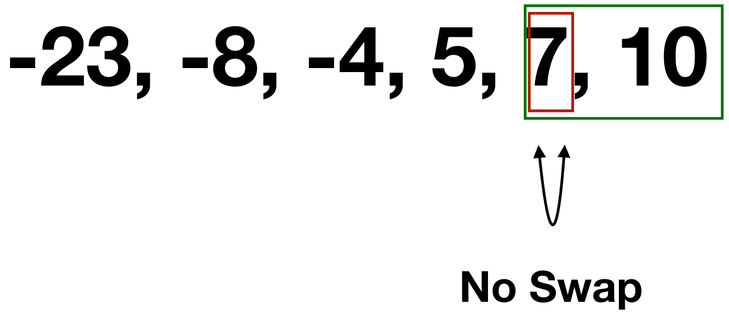
-23, 5, -8, 7, -4, 10

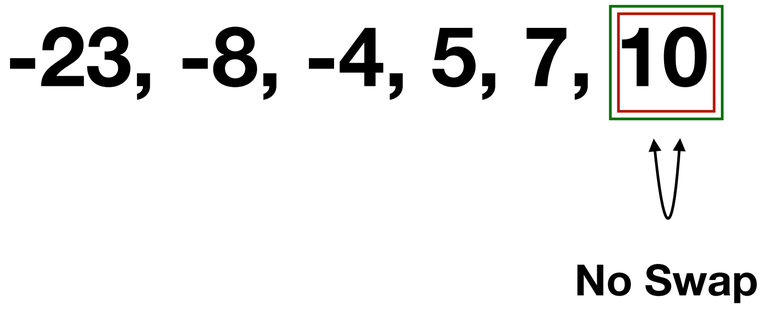
Selection Sort

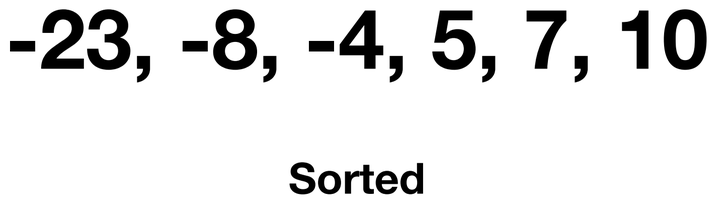












Selection Sort

- The algorithm only needs to know how to compare 2 values

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

- But how do we compare 2 values?

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

- Take a comparator as a parameter just like sortWith

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

- Call the comparator whenever we need to compare 2 values

```
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Selection Sort

```scala
val numbers = List(5, -23, -8, 7, -4, 10)
val numbersSorted = intSelectionSort(numbers, (a: Int, b: Int) => a > b)
```

```scala
def intSelectionSort(inputData: List[Int], comparator: (Int, Int) => Boolean): List[Int] = {

  // copy only the reference of the input
  var data: List[Int] = inputData

  for (i <- data.indices) {

    // find the min value/index from i to the end of the list
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)

      // make decisions based on the given comparator (this function can be thought of as a less than operator)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }

    // swap the value at i with the min value
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }

  // return the new list
  data
}
```

Type Parameters

- But what if we want to sort custom types?

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = selectionSort(animals, Animal.compareAnimals)
println(animalsSorted)
```

- Our selection sort only works with Ints
- We can write another method to sort Animals
  - And another for every type we want to sort?.. no
- We'll take the **type** as a **parameter** of our method
  - A "**type parameter**"

Type Parameters

- Type parameters come before the parameter list
- Use [ ] instead of ( )
- Can use this generic type throughout this method

```
def selectionSort[Type](inputData: List[Type], comparator: (Type, Type) => Boolean): List[Type] = {

  var data: List[Type] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Type Parameters

- We can choose the type name
- Generic type names are often shortened to 1 character

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Type Parameters

- Specify the type parameter when calling the method

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = selectionSort[Animal](animals, Animal.compareAnimals)
println(animalsSorted)
```

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Type Parameters

- The type parameter can be inferred as long as the data and comparator types match

```
val animals: List[Animal] = List(new Cat("morris"), new Dog("Finn"), new Dog("Snoopy"), new Cat("Garfield"))
val animalsSorted = selectionSort(animals, Animal.compareAnimals)
println(animalsSorted)
```

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort

- This all works..
- **But it's really slow!**
- The algorithm is inefficient -- O(n^2)
- My implementation is even slower -- O(n^3)
  - Very inefficient use of Lists
- More efficiency coming soon

Lecture Question

Question: In a package named "functions" create an **object** named Generics with a method named mapFilter that:

- Takes a type parameter T
- As parameters takes
  - A Map of Ints to T's
  - A function that takes an Int and returns a Boolean
- Returns a List of T's
  - The returned List will contain only the values in the input map that are mapped to by keys for which the input function returns true
  - eg. We can think of the input function as a filter that decides which Map values will be added to the return list. If the filter returns true on a key, they value the key maps to will be in the returned list
  - The order of the output List is not defined in this problem (You should be sorting lists in your testing)

Testing: In a package named "tests" create a class named "TestFilterMap" as a test suite that tests all the functionality listed above