

Merge Sort / Recursion

Lecture Question

Restriction: No state is allowed in this question. Specifically, the keyword "var" is banned. (ie. You are expected to use a recursive solution)

Question: In a package named "functions" create an **object** named Numbers with a method named fib that:

- Takes an Int as a parameter and returns the n^{th} fibonacci number
- Fibonacci numbers are equal to the sum of the previous two fibonacci numbers starting with 1 and 1 as the first two numbers in the sequence
- Fibonacci numbers: 1, 1, 2, 3, 5, 8...
- `fib(1) == 1`
- `fib(2) == 1`
- `fib(3) == 2`
- `fib(4) == 3`
- Your method will not be tested with inputs < 1
- Your method will not be tested with inputs > 46

Testing: In a package named "tests" create a class named "TestFib" as a test suite that tests all the functionality listed above. (Do not test with inputs < 1 or > 46)

Runtime Analysis

- Last time we said Selection sort is inefficient
- Let's be more specific
- We'll measure the asymptotic runtime of the algorithm
 - Often use big-O notation
- Count the number of "steps" the algorithm take
 - A step is typically a basic operation (+, -, &&, etc)

Runtime Analysis

- Asymptotic runtime
 - Measures the order of magnitude of the runtime in relation to the size of the input
 - Name the input size **n**
 - For sorting - Size of the input is the number of values in the data structure
 - Ignore constants
- Ex. Runtime of $O(n)$ grows linearly with the size of the input

Selection Sort - Runtime

- Abridged runtime analysis

**Outer loop
runs once
for each
index**

**Runs $O(n)$
times**

```
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 - Runtime

- Abridged runtime analysis

Inner loop
runs once
for each
index from
i to the end
of the list

Runs for
each
iteration of
the outer
loop with
a worst
case of
 $O(n)$

```
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 - Runtime

- Abridged runtime analysis

**Run $O(n)$
iterations
 $O(n)$ times
results in
an $O(n^2)$
total
runtime**

```
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 - Runtime

- Abridged runtime analysis

```
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  
}
```

We reach
 $O(n^3)$
since apply
takes $O(n)$

More details
next week

Selection Sort - Runtime

- More mathematical analysis
 - Inner loop runs $\sum i$ times where i ranges from n to 1
 - $n + n-1 + n-2 + \dots + 2 + 1 = n^2 / 2 + n/2$
 - For asymptotic we only consider the highest order term and ignore constant multipliers
 - Therefore $n^2 / 2 + n/2$ is $O(n^2)$
 - Selection Sort has $O(n^2)$ runtime

Merge Sort

- We briefly saw in CSE115 that we can do better by using merge sort and reaching $O(n \log(n))$ runtime
- Let's analyze this in more depth

Merge Sort

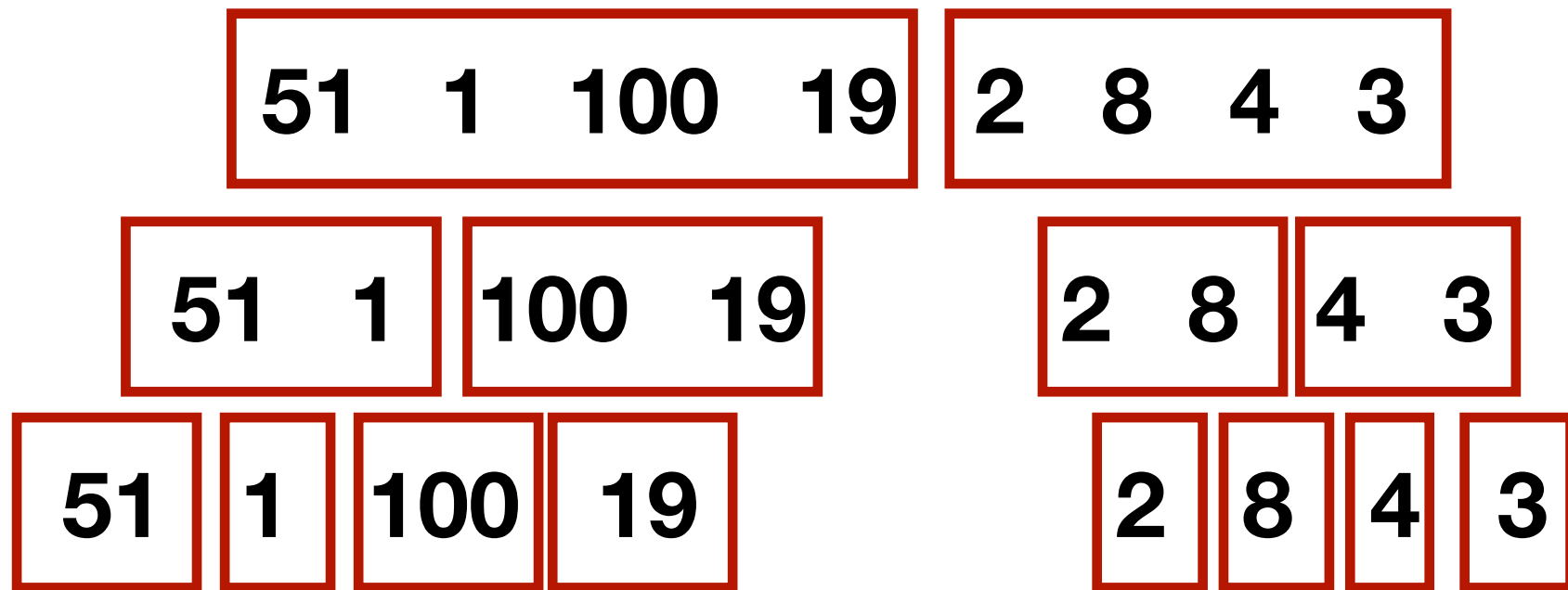
- The algorithm
 - If the input list has < 2 elements
 - Return it (It's already sorted)
 - Else
 - Divide the input list in two halves
 - Recursively call merge sort on each half (Repeats until the lists are size 1)
 - Merge the two sorted lists together into a single sorted list

Merge Sort

- Given an input

51 1 100 19 2 8 4 3

- Divide into two lists recursively until $n=1$



Merge Sort

- Merge lists until the original input is sorted

51 1 100 19

2 8 4 3

1 51 19 100

2 8 3 4

1 19 51 100 2 3 4 8

1 2 3 4 8 19 51 100

Merge Sort - Merge

```
def mergeSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {  
  if (inputData.length < 2) {  
    inputData  
  } else {  
    val mid: Int = inputData.length / 2  
    val (left, right) = inputData.splitAt(mid)  
    val leftSorted = mergeSort(left, comparator)  
    val rightSorted = mergeSort(right, comparator)  
    merge(leftSorted, rightSorted, comparator)  
  }  
}
```

Recursion!

Merge Sort - Runtime

- Each level of the recursion has 2^i lists of size $n/2^i$
- Recursion ends when $n/2^i == 1$
 - $i = \log(n)$
 - $\log(n)$ levels of recursion
- Each level needs to merge a total of n elements across all sub-lists
- If we can merge in $O(n)$ time we'll have $O(n \log(n))$ total runtime

Merge Sort - Merge

- Merge two sorted lists in $O(n)$ time
- Take advantage of each list being sorted
- Start with pointers at the beginning of each list
- Compare the two values at the pointers and find which come first based on the comparator
 - Append it to a new list and advance that pointer
- When a pointer reaches the end of a list copy the rest of the contents

Merge Sort - Merge

1 19 51 100
↑

2 3 4 8
↑



Merge Sort - Merge

1 19 51 100




2 3 4 8




1

Merge Sort - Merge

1 19 51 100




2 3 4 8




1 2

Merge Sort - Merge

1 19 51 100



2 3 4 8



1 2 3

Merge Sort - Merge

1 19 51 100



2 3 4 8



1 2 3 4

Merge Sort - Merge

1 19 51 100



2 3 4 8



When a pointer reaches the end of a list,
copy the rest of the other list to the result

1 2 3 4 8

Merge Sort - Merge

1 19 51 100



2 3 4 8



**When a pointer reaches the end of a list,
copy the rest of the other list to the result**

1 2 3 4 8 19 51 100

Merge Sort - Merge

```
def merge[T](left: List[T], right: List[T], comparator: (T, T) => Boolean): List[T] = {  
  var leftPointer = 0  
  var rightPointer = 0  
  
  var sortedList: List[T] = List()  
  
  while (leftPointer < left.length && rightPointer < right.length) {  
    if (comparator(left.apply(leftPointer), right.apply(rightPointer))) {  
      sortedList = sortedList :+ left.apply(leftPointer)  
      leftPointer += 1  
    } else {  
      sortedList = sortedList :+ right.apply(rightPointer)  
      rightPointer += 1  
    }  
  }  
  
  while (leftPointer < left.length) {  
    sortedList = sortedList :+ left.apply(leftPointer)  
    leftPointer += 1  
  }  
  while (rightPointer < right.length) {  
    sortedList = sortedList :+ right.apply(rightPointer)  
    rightPointer += 1  
  }  
  
  sortedList  
}
```

Use the comparator to make ordering decisions

Writing Recursive Methods

- Suggested approach:
 - Assume your recursive calls return the correct values
 - Write your method based on this assumption
 - Add a base case(s) for an input that has a trivial return value
 - Only write recursive calls that get closer to the base case

Writing Recursive Methods

- Suggested approach:
 - **Assume your recursive calls return the correct values**
 - **Write your method based on this assumption**
 - Add a base case(s) for an input that has a trivial return value
 - Only write recursive calls that get closer to the base case

```
def mergeSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {  
  val mid: Int = inputData.length / 2  
  val (left, right) = inputData.splitAt(mid)  
  val leftSorted = mergeSort(left, comparator)  
  val rightSorted = mergeSort(right, comparator)  
  
  merge(leftSorted, rightSorted, comparator)  
}
```

Writing Recursive Methods

- Suggested approach:
 - Assume your recursive calls return the correct values
 - Write your method based on this assumption
 - **Add a base case(s) for an input that has a trivial return value**
 - **Only write recursive calls that get closer to the base case**

```
def mergeSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {  
  if (inputData.length < 2) {  
    inputData  
  } else {  
    val mid: Int = inputData.length / 2  
    val (left, right) = inputData.splitAt(mid)  
    val leftSorted = mergeSort(left, comparator)  
    val rightSorted = mergeSort(right, comparator)  
  
    merge(leftSorted, rightSorted, comparator)  
  }  
}
```

Writing Recursive Methods

- Assume your recursive calls return the correct values
 - and-
- Write your method based on this assumption
- The primary benefit of writing recursive methods/functions is that we can assume that the recursive calls are correct
- If these calls are not correct, we have work to do elsewhere
 - While writing the top level functionality, assume they are correct and fix the other issues if they are not

Writing Recursive Methods

- Add a base case(s) for an input that has a trivial return value
 - A simple input where the return value is trivial
 - Ex. An empty list, an empty String, 0, 1
- Add a conditional to your method to check for the base case(s)
 - If the input is a base case, return the trivial solution
 - Else, run your code that makes the recursive call(s)

Writing Recursive Methods

- Ensure your recursive calls always get closer to your base case
 - Base case is eventually reached and returned
 - Ex. Base case is 0, each recursive call decreases the input
 - Ex. Base case is the empty String and an each recursive call removes a character from the input
- If your recursive calls don't reach a base case
 - Infinite recursion
 - Stack overflow

Lecture Question

Restriction: No state is allowed in this question. Specifically, the keyword "var" is banned. (ie. You are expected to use a recursive solution)

Question: In a package named "functions" create an **object** named Numbers with a method named fib that:

- Takes an Int as a parameter and returns the n^{th} fibonacci number
- Fibonacci numbers are equal to the sum of the previous two fibonacci numbers starting with 1 and 1 as the first two numbers in the sequence
- Fibonacci numbers: 1, 1, 2, 3, 5, 8...
- `fib(1) == 1`
- `fib(2) == 1`
- `fib(3) == 2`
- `fib(4) == 3`
- Your method will not be tested with inputs < 1
- Your method will not be tested with inputs > 46

Testing: In a package named "tests" create a class named "TestFib" as a test suite that tests all the functionality listed above. (Do not test with inputs < 1 or > 46)