

Comp 311

Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments

How to Decide Between Structural and Generative Recursion

- Structural recursion is typically:
 - Easier to design
 - Easier to understand
- Generative recursion can be faster (sometimes!)

How to Decide Between Structural and Generative Recursion

- As a general guideline:
 - Start with structural recursion
 - If it turns out to be too slow:
 - Explore generatively recursive approaches

Strategies for Generative Recursion

Binary Search

- The strategy of searching over a sequence by breaking in half and searching over just one of them
- Our search for blue-eyed ancestors falls into this category
- We could also use binary search for root finding
- Newton's Method could be viewed as an optimization on binary search for root finding

Divide and Conquer

- The strategy of breaking a problem into smaller sub-problems of the same type
- Quicksort falls into this category

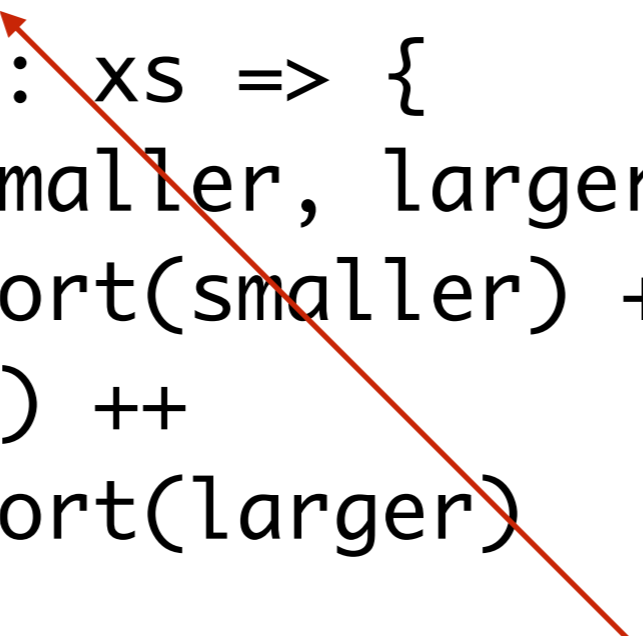
Quicksort

```
def quickSort(xs: List[Int]): List[Int] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => {  
      val (smaller, larger) = separate(xs, x)  
      quickSort(smaller) ++  
      List(x) ++  
      quickSort(larger)  
    }  
  }  
}
```

Quicksort

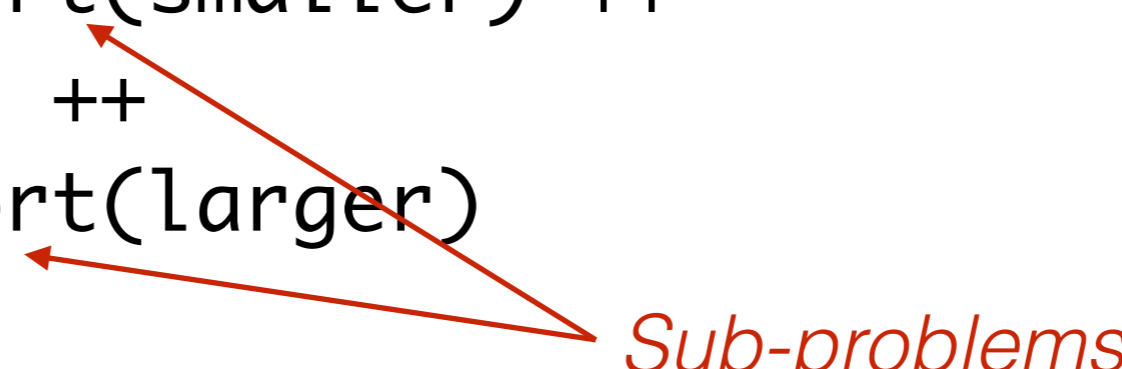
```
def quickSort(xs: List[Int]): List[Int] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => {  
      val (smaller, larger) = separate(xs, x)  
      quickSort(smaller) ++  
      List(x) ++  
      quickSort(larger)  
    }  
  }  
}
```

Trivially solvable



Quicksort

```
def quickSort(xs: List[Int]): List[Int] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => {  
      val (smaller, larger) = separate(xs, x)  
      quickSort(smaller) ++  
      List(x) ++  
      quickSort(larger)  
    }  
  }  
}
```



Sub-problems

The diagram consists of two red arrows pointing from the text 'Sub-problems' to the recursive calls 'quickSort(smaller)' and 'quickSort(larger)' in the code. This highlights the recursive nature of the algorithm where each call processes a smaller sub-list.

Quicksort

```
def quickSort(xs: List[Int]): List[Int] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => {  
      val (smaller, larger) = separate(xs, x)  
      quickSort(smaller) ++  
      List(x) ++  
      quickSort(larger)  
    }  
  }  
}
```

Combination

Separate

```
def separate(xs: List[Int], x: Int): (List[Int], List[Int]) = {  
  xs match {  
    case Nil => (Nil, Nil)  
    case y :: ys => {  
      val (smaller, larger) = separate(ys, x)  
      if (y < x) (y :: smaller, larger)  
      else (smaller, y :: larger)  
    }  
  }  
}
```

Description and Termination Argument

```
/**
```

- * Recurs on two sublists of the given list:
 - * All elements smaller than a given “pivot”
 - * All elements at least as large as the pivot
 - * Appends the recursive solutions.
 - * Because each sublist is strictly smaller
 - * (the pivot was extracted from the list),
 - * we eventually recur on an empty list.
- ```
*/
```

```
def quickSort(xs: List[Int]): List[Int] = {
 ...
}
```

# Backtracking Algorithms

# Graph Algorithms

- Many problems can be expressed as traversals or computations over graphs
  - Travel planning
  - Circuit design
  - Social networks
  - etc.

# Graph Algorithms

- We consider the problem of finding a path from one vertex to another in a graph

# Data Analysis and Design

- We model graphs as Maps of Strings to Lists of Strings

```
case class Graph(elements: (String, List[String])*
extends Function1[String, List[String]] {
 val _elements = Map(elements:_*
 def apply(s: String) = _elements(s)
}
```



# Data Analysis and Design

- We model graphs as Maps of Strings to Lists of Strings

```
val sampleGraph =
 new Graph ("A" -> List("E", "B"),
 "B" -> List("A"),
 "C" -> List("D"),
 "D" -> List(),
 "E" -> List("C", "F"),
 "F" -> List("A", "G"),
 "G" -> List())
```

# What is a Trivially Solvable Problem?

- If the start and end vertices are identical

# How Do We Generate Sub-Problems?

- Find nodes connected to start and recur

# How Do We Relate the Solutions?

- We need only find one solution; no need to combine multiple solutions

# Contract Attempt 1

```
/**
 * Create a path from start to finish in G
 */
def findRoute(start: String, end: String,
 graph: Graph): List[String]
```



*But what if there is no path?*

# Options

- Often the result of a computation is that no satisfactory value could be found
  - Lookup in a table with a key that does not exist
  - Attempting to find a path that does not exist

# Scala Options

```
abstract class Option[+A] {...}
```

```
object None extends Option[Nothing] {...}
```

```
class Some[+A](val contained: A) extends Option[A] {
 ...
}
```

# Options Are Monads!

```
abstract class Option[+A] {
 def flatMap[B](f: (A) => Option[B]): Option[B]
 def map[B](f: (A) => B): Option[B]
 def withFilter(p: (A) => Boolean):
 FilterMonadic[A, collection.Iterable[A]]
}
```



# Contract Attempt 2

```
/**
 * Create a path from start to finish in G, if
 * it exists.
 */
def findRoute(start: String, end: String,
 graph: Graph):
 Option[List[String]]
```

# Reduce to Backtracking Cases

```
def findRoute(start: String, end: String,
 graph: Graph): Option[List[String]] = {
 if (start == end) Some(List(end))
 else for (route <- routeFromOrigins(graph(start), end, graph))
 yield start :: route
}
```

# Recursive Sub-Problems

```
def routeFromOrigins(origins: List[String], destination: String,
 graph: Graph): Option[List[String]] = {
 origins match {
 case Nil => None
 case origin :: origins => {
 findRoute(origin, destination, graph) match {
 case None => routeFromOrigins(origins, destination, graph)
 case Some(route) => Some(route)
 }
 }
 }
}
```

# Termination

- `routeFromOrigins` is structurally recursive:
  - It terminates provided that `findRoute` terminates
- But `findRoute` terminates only if there are no cycles in the graph it traverses