

Comp 311

Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

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

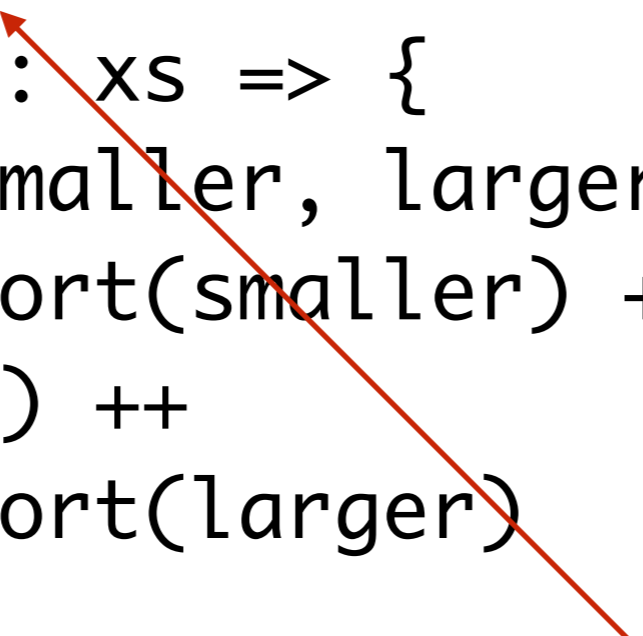
Quicksort

```
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

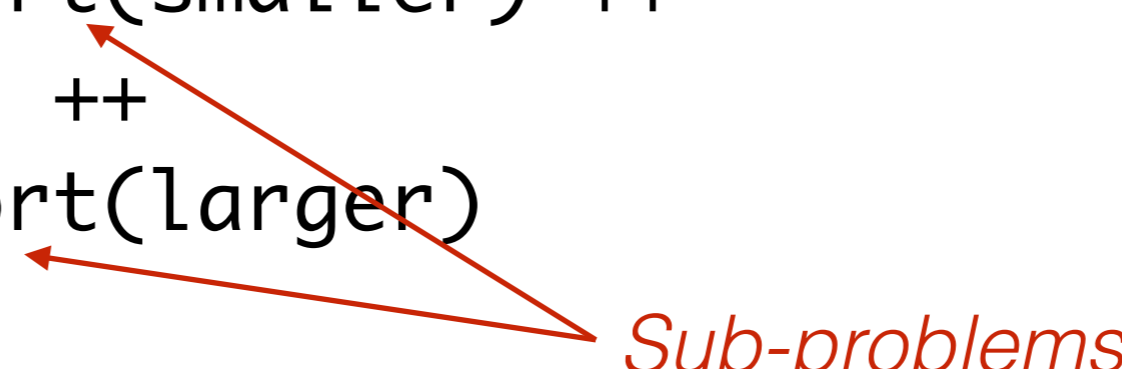
```
def quickSort(xs: List[Int]): List[Int] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => {  
      val (smaller, larger) = separate(xs, x)  
      quickSort(smaller) ++  
      List(x) ++  
      quickSort(larger)  
    }  
  }  
}
```

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)  
    }  
  }  
}
```



The diagram consists of two red arrows pointing from the text "Sub-problems" to the recursive calls in the code. One arrow points to the `quickSort(smaller)` call, and the other points to the `quickSort(larger)` call. The text "Sub-problems" is written in red italics.

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

```
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

Accumulating Knowledge

Accumulating Knowledge

- In recursive calls, we need to remember what nodes we have already visited, so we can prevent infinite regress
- We pass this information to recursive calls via an additional “accumulator” parameter

Reduce to Backtracking Cases

```
def findRoute(start: String, end: String, graph: Graph,
              visited: List[String] = Nil):
Option[List[String]] = {
  if (start == end) Some(List(end))
  else if (visited contains start) None
  else for (route <- routeFromOrigins(graph(start), end, graph,
                                      start :: visited))
    yield start :: route
}
```

Reduce to Backtracking Cases

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

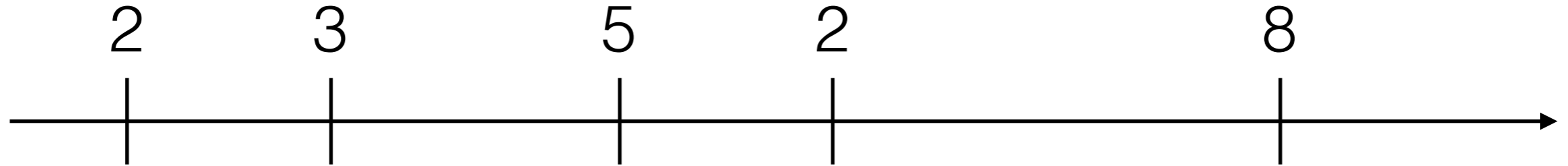

Accumulators

- Keeping an accumulator parameter allows us to “remember” knowledge from one recursive call to another
- Often essential for correctness in generative recursion
- Also useful for saving space in structural recursion

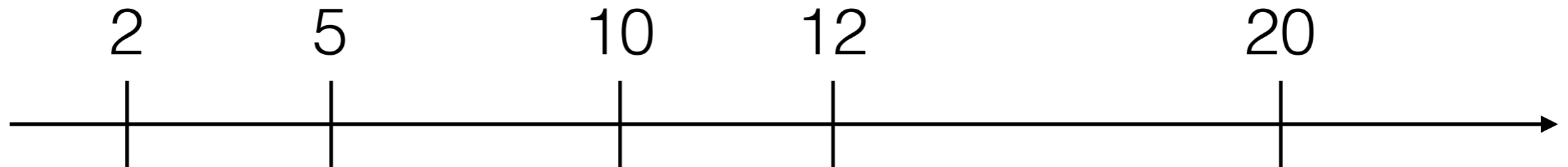
Accumulators for Structural Recursion

- Let us define a function `relativeToAbsolute`, which:
 - Takes a list of `Double` values, with each value denoting a relative distance to the point to its left
 - Returns a list of `Double` values denoting the absolute distances to the origin

Accumulators for Structural Recursion



becomes



Defining relativeToAbsolute

```
def relativeToAbsolute[T](xs: List[T]) = {  
  xs match {  
    case Empty => Empty  
    case x :: xs => x :: relativeToAbsolute(map(_ + x)(xs))  
  }  
}
```

Defining relativeToAbsolute

```
def relativeToAbsolute(xs: List[Double]): List[Double] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => x :: relativeToAbsolute {  
      for (x1 <- xs) yield x + x1  
    }  
  }  
}
```

How many steps does it take to compute an application of relativeToAbsolute, in comparison to the length of the list?

The Cost of relativeToAbsolute

```
relativeToAbsolute(List(2,3,5,2,8)) ↪
  List(2,3,5,2,8) match {
    case Empty => Empty
    case x :: xs => x :: relativeToAbsolute(map(_ + x)(xs))
  } ↪
2 :: relativeToAbsolute(map(_ + 2)(List(3,5,2,8))) ↪*
2 :: relativeToAbsolute(5 :: map(_ + 2)(List(5,2,8))) ↪*
2 :: relativeToAbsolute(5 :: 7 :: map(_ + 2)(List(2,8))) ↪*
2 :: relativeToAbsolute(5 :: 7 :: 4 :: map(_ + 2)(List(8))) ↪*
2 :: relativeToAbsolute(5 :: 7 :: 4 :: 10 :: map(_ + 2)(List())) ↪*
2 :: relativeToAbsolute(5 :: 7 :: 4 :: 10 :: Nil) ↪ *
...
```

The cost of relativeToAbsolute

- Each recursive call requires a map over the argument list, which takes n steps for a list of length n

$$\sum_{i=1}^n i = \frac{(n)(1+n)}{2} = O(n^2)$$

Big O Notation

- We say:

$$f(x) = O(g(x)) \text{ as } x \rightarrow \infty$$

- To mean that there is a constant k and some value x_0 such that

$$|f(x)| \leq k|g(x)| \text{ for all } x \geq x_0$$

Big O Notation

- Typically the part:

$$\text{as } x \rightarrow \infty$$

- is implicit
- Effectively, we are defining equivalence classes of functions

Accumulating Distance to the Origin

- We could reduce the time taken by instead accumulating the distance to the origin in a parameter

Accumulating Distance to the Origin

```
def relativeToAbsolute(xs: List[Double]) = {  
  def inner(xs: List[Double], distanceToOrigin: Double):  
    List[Double] = {  
    xs match {  
      case Nil => Nil  
      case x :: xs => {  
        val xToOrigin = x + distanceToOrigin  
        xToOrigin :: inner(xs, xToOrigin)  
      }  
    }  
  }  
  inner(xs, 0)  
}
```

Guidelines for Using Accumulators in Functions

- Start with the standard design recipes!
- Add an accumulator *only after* the initial design attempt

Guidelines for Using Accumulators in Functions

- Recognize the benefit to the function of having an accumulator
- Understand what the accumulator denotes

Recognizing the Benefit of an Accumulator

- If the function is structurally recursive and uses an auxiliary function, consider an accumulator
- Study hand evaluations to see if an accumulator helps in reducing time or space costs

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {  
  xs match {  
    case Nil => Nil  
    case x :: xs => makeLastItem(x, invert(xs))  
  }  
}
```

```
def makeLastItem[T](x: T, xs: List[T]): List[T] = {  
  xs match {  
    case Nil => List(x)  
    case y :: ys => y :: makeLastItem(x, ys)  
  }  
}
```

Recognizing the Benefit of an Accumulator

- There is nothing for invert to forget
- However, we might consider accumulating the items walked over

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {  
  def inner(xs: List[T], accumulator: List[T]): List[T] = {  
    xs match {  
      case Nil => ...  
      case y :: ys => ... inner(... ys ... y ... accumulator ...)  
    }  
  }  
  inner(xs, Nil)  
}
```

Recognizing the Benefit of an Accumulator

- The accumulator must stand for a list
- Maybe it could stand for all elements that precede **XS**

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {  
  def inner(xs: List[T], accumulator: List[T]): List[T] = {  
    xs match {  
      case Nil => ...  
      case y :: ys => ... inner(... ys ... y :: accumulator)  
    }  
  }  
  inner(xs, Nil)  
}
```

Recognizing the Benefit of an Accumulator

- Now it is clear that the accumulator contains all the elements that precede *xs* *in reverse order*

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {  
  def inner(xs: List[T], accumulator: List[T]): List[T] = {  
    xs match {  
      case Nil => accumulator  
      case y :: ys => inner(ys, y :: accumulator)  
    }  
  }  
  inner(xs, Nil)  
}
```

Recognizing the Benefit of an Accumulator

- The key step in the design process is to establish the invariant that describes the relationship between the accumulator and the parameters of a function
- Establish appropriate accumulator invariant is an art that takes practice

Recognizing the Benefit of an Accumulator

```
def sum1(xs: List[Int]): Int = {  
  xs match {  
    case Nil => 0  
    case y :: ys => y + sum1(ys)  
  }  
}
```

An Accumulator for Sum

- We are walking over the elements of a list to return their sum
- The most obvious thing to accumulate is the value of the sum so far

An Accumulator for Sum

```
def sum2(xs: List[Int]): Int = {  
  def inner(xs: List[Int], accumulator: Int): Int = {  
    xs match {  
      case Nil => ...  
      case y :: ys => ...inner(...ys ... y + accumulator)  
    }  
  }  
  inner(xs, 0)  
}
```

An Accumulator for Sum

```
def sum2(xs: List[Int]): Int = {  
  def inner(xs: List[Int], accumulator: Int): Int = {  
    xs match {  
      case Nil => accumulator  
      case y :: ys => inner(ys, y + accumulator)  
    }  
  }  
  inner(xs, 0)  
}
```

An Accumulator for Sum

```
sum1(List(5, 3, 7, 9)) ↦*
5 + sum1(List(3, 7, 9)) ↦*
5 + 3 + sum1(List(7, 9)) ↦*
5 + 3 + 7 + sum1(List(9)) ↦*
5 + 3 + 7 + 9 + sum1(List()) ↦*
5 + 3 + 7 + 9 + 0 ↦
8 + 7 + 9 + 0 ↦
15 + 9 + 0 ↦
24 + 0 ↦
24
```

An Accumulator for Sum

```
sum2(List(5, 3, 7, 9))  $\mapsto^*$   
inner(List(5, 3, 7, 9), 0)  $\mapsto^*$   
inner(List(3, 7, 9), 5 + 0)  $\mapsto^*$   
inner(List(3, 7, 9), 5)  $\mapsto^*$   
inner(List(7, 9), 5 + 3)  $\mapsto^*$   
inner(List(7, 9), 8)  $\mapsto^*$   
inner(List(9), 7 + 8)  $\mapsto^*$   
inner(List(9), 15)  $\mapsto^*$   
inner(List(), 9 + 15)  $\mapsto^*$   
inner(List(), 24)  $\mapsto^*$ 
```

An Accumulator for Sum

- The key advantage of our accumulator version of sum is space
- The advantage is not a matter as to whether the space is used on the stack or in the heap as an argument!
- The ability to reduce the sum as we recur is the primary cause of space savings

This Would Not Save Space

```
def sum3(xs: List[Int]): Int = {  
  def inner(xs: List[Int], accumulator: () => Int): Int = {  
    xs match {  
      case Nil => accumulator()  
      case y :: ys => inner(ys, () => (y + accumulator()))  
    }  
  }  
  inner(xs, () => 0)  
}
```