Fun with Scala	Study group 1
How is Binary Search like the Fibonacci numbers ?	April 26, 2016

1 Describing a class of $\mathcal{O}\lg(n)$ Algorithms including Fibonacci

1.1 A pattern based on performance (can ignore the math if you want)

An algorithm that operates in constant time on a recursive call of size $\frac{n}{2}$ is $\mathcal{O}\lg n$. For completeness we will prove this claim below. Here is the generic algorithm

1.1.1 Algorithm *telescope*

Input: size, *halfer*, *combine*, *guard*, *input*
Output: accumulator

1. `guard(input)`
2. (a) `half` $\leftarrow$ `telescope(halfer,combiner,guard,halfer(input))`
   (b) `ret` $\leftarrow$ `combine(half,input)`
   (c) `return ret`

We claim that the class of *telescope* algorithms is $\mathcal{O}\lg n$

Proof.

$$t_n = t_{\frac{n}{2}} + c.$$

Where c is the presumed constant cost of the combiner function. For some k set

$$n =: 2^k$$

Define sequence $r_0, r_1, ...$ such that

$$r_k := t_{2^k}.$$

now

$$
\begin{aligned}
r_k &= r_{k-1} + c. \\
&\text{Setting } r_0 := 1 \text{ and summing both sides by } \sum_{i=1} x^i \\
r_1 x + r_2 x^2 + r_3 x^3 + ... &= r_0 x + r_1 x^2 + r_2 x^3 + ... + c x^1 + c x^2 + ... \\
&\text{let } R(x) = \sum_{i=0} r_i x^i \\
&\text{and simplifying} \\
R(x) - 1 &= xR(x) + \frac{cx}{1-x} \\
R(x) - xR(x) - 1 &= \frac{cx}{1-x} \\
R(x)(1-x) &= \frac{(c-1)x + 1}{1-x} \\
R(x) &= \frac{(c-1)x + 1}{(1-x)^2}
\end{aligned}
\tag{1}
$$

Now r_k is the coefficient of x^k in $R(x)$ which, by the generalized binomial theorem and substituting back for $k = \lg(n)$ is

$$
c - 1\binom{k}{k-1} + \binom{k+1}{k} = (c-1)\lg n + \lg n + 1 = c \lg n + 1.
$$

Since c is a constant we are $\mathcal{O} \lg(n)$. □

The point is not to be overly formal, but to stress that we have a "higher order" recipe which guarantees a certain runtime performance. Here we are emphesising a recursive pattern, but other times our higher order function could handle diffferent execution contexts, such as "map reduce".

2 Lets go into some Scala

We have shown in the previous section the performance of a class of algorithms that takes an array and based on some constant operation either returns a result or recurses on exactly half of the input, results in $\mathcal{O} \lg n$ runtime. Lets look at some examples, and see how we can leverage Scala's declarative abstraction to create a higher order function which leverages common patterns.

2.1 Excercise. Write Scala code to compute x^n, can you get an impure algorithm? A pure algorithm? Can you get a $\mathcal{O} \lg n$ algorithm?

Solutions below:

```
object Powers {

  //linear time -- not the pattern we want
  def oldSchool(x:Int, n:Int):Int = {
    var i:Int = 0
    var p:Int = x
    while(i < n)
    {
      p = p * x
      i = i + 1
     }
    p
  }

  //linear time -- still not the pattern (btw foreach is nice but not pure)
  def newerSchool(x:Int, n:Int):Int = {
    var p:Int = x
    val li = new List(n).fill(x)
    li.foreach(x => p = p*x)
    p
  }

  //linear time but pure -- not the pattern from the first section
  @talrec
  def noLoops(x:Int,n:Int, acc:Int = 1):Int = n match {
    case 0 => acc
    case _ => noLoops(x, n-1, acc*x)
    }

  //lgn runtime. exploits that x^n = x^n/2 * x^n/2. not tail recursive though!
  def logNpow(x:Int, n:Int):Int = {
    if(n == 1)
      x
    else if(n mod 2 == 0)
    {
      val half = logNpow(x,n/2)
      half * half
    }
    else{
      val half = logNpow(x, (n-1)/2)
      half * half * x
      }
  }
  // can we get a tail recursive version of the above?
```

```
  // left to the reader

}
```

Next lets see if we can plug this integer power algorithm into our telescope abstraction. We need to specify size, halfer, combine, guard, and input parameters. Lets start by figuring out their types based on our logNPow function above. The basic input parameter is x so thats an Int. The notion of "size" probably applies to the exponent, here its n which is also an Int. We can see that the first thing we do is to check if $n == 1$ and if so return the input. Thus guard: (Int,Int) $\Rightarrow$ Int...We will list the types below, and annotate with subscript thier corresponding parts from the program.

val size : Int_n

val input Int_x

def guard $(\text{Int}_n,\text{Int}_x) \Rightarrow_{cond} \text{Int}_{res}$

def halfer $(\text{Int}_n,\text{Int}_x) \Rightarrow \text{Int}_{half}$

def combine $(\text{Int}_n, \text{Int}_x, \text{Int}_{half}) \Rightarrow \text{Int}_{res}$

def telescope (size input guard halfer combine) $\Rightarrow$ Int

Lets now write these and then write the integer power function in terms of these functions. *halfer* needs to recursively call the higher level function *telescope*. *guard* is interesting since it only returns under some of the input, otherwise it is ignored. We will use a PartialFunction to model this behavior. The rest are pretty straightforward:

```
 val guard:PartialFunction[(Int,Int),Int] = {
   case (1,x) => x
   case (0,_) => 1
 }

 def halfer(n:Int, x:Int):Int = n \% 2 match {
   case 0 => telescope(n/2,x,guard,halfer,combine)
   case 1 => telescope((n-1)/2,x,guard,halfer,combine)
 }

def combine(n:Int, x:Int, half:Int) = n \% 2 match {
   case 0 => half * half
   case 1 => half * half * x //uhhh i think this is right..
}

//lets code this meta beast..
```

```
def telescope(n:Int, x:Int, guard:PartialFunction[(Int,Int),Int], halfer:
    (Int,Int)=>Int, combine:(Int,Int,Int)=>Int):Int = {
  if(guard.isDefinedAt(n,x))
    guard(n,x)
  else{                 //follow the bouncing ball...(it kind of writes itself)
    val half = halfer(n,x)
    combine(n,x,half)
    }
}
//actually that really wasnt so bad, the function signature was a bit much but
    scala has conveniant ways of dealing with this...
//(believe it or not this sh** actually compiles)
scala> telescope(3,2,guard,halfer,combine)
res0: Int = 8
```

2.2 Lets Abstract the Types

val size N_n

val input T_x

def guard $(\mathrm{N}_n,\mathrm{T}_x)\Rightarrow_{cond}\mathrm{T}_{res}$

def halfer $(\mathrm{N}_n,\mathrm{T}_x)\Rightarrow\mathrm{T}_{half}$

def combine $(\mathrm{N}_n,\ \mathrm{T}_x,\ \mathrm{T}_{half})\Rightarrow\mathrm{T}_{res}$

def telescope (size input guard halfer combine) $\Rightarrow$ T

Again, its not so bad. There are only two types. One for the object of type T, and one for the size of type N. We can now rewrite telescope to be much more general

---

```
def telescope[N,T](n:N, x:T, guard:PartialFunction[(N,T),T], halfer: (N,T)=>T,
    combine:(N,T,T)=>T):T = {
    if(guard.isDefinedAt(n,x))
      guard(n,x)
    else{
      val half = halfer(n,x)
      combine(n,x,half)
    }
  }
//guess what, it still works without any further changes!
scala> telescope(5,2,guard,halfer,combine)
res1: Int = 32
```

```
// we can clean it up using currying and implicits
  def telescope[N,T](n:N, x:T)(implicit guard:PartialFunction[(N,T),T], halfer:
      (N,T)=>T, combine:(N,T,T)=>T):T = {
    if(guard.isDefinedAt(n,x))
      guard(n,x)
    else{
      val half = halfer(n,x)
      combine(n,x,half)
    }
  }

  scala> def powInt = telescope(_:Int,_:Int)(guard,halfer,combine)
  powInt: (Int, Int) => Int

  scala> powInt(2,6)
  res7: Int = 36

//or maybe better
def telescope[N,T](guard:PartialFunction[(N,T),T], halfer: (N,T)=>T,
    combine:(N,T,T)=>T)(n:N, x:T):T = {
    if(guard.isDefinedAt(n,x))
      guard(n,x)
    else{
      val half = halfer(n,x)
      combine(n,x,half)
    }
  }

scala> def powInt = telescope(guard,halfer,combine) _
powInt: (Int, Int) => Int

scala> powInt(2,4)
res0: Int = 16
```

2.3 Excercise. Write Scala code to compute M^n where M is a 2×2 matrix

```
//one possibility for simplistic two by two matrix representation..
case class TwoByTwo(r00:Int, r01:Int, r10:Int, r11:Int)
...
```

hint: *matrix powers and integer powers are somewhat similar*

2.4 Excercise. Write Scala code to compute the nth fibonacci.

hint : $M_{fib} := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ *what is* M_{fib}^3 *in terms of the fibonaccis ?*

2.5 write binary search

use a different notion of size for $N := (Int, Int)$