

Recursion

- Recursion is a technique that is often used to program certain tasks.
- It is essentially a way to loop or iterate over different states.
- It can be very natural and greatly simplify certain problems.
- It can also be quite inefficient and more clumsy iterative techniques can be more efficient.

16

Fibonacci Numbers

- Fibonacci introduced a sequence of numbers defined by the n -th element F_n
 $F_0 = 0,$
 $F_1 = 1$
 $F_n = F_{n-1} + F_{n-2}, n > 1$
- It is a sequence that arises in many different contexts and has amazing mathematical properties.
- For our purposes, note that the value for n is computed from previously computed values, i.e. for $n - 1$ and $n - 2$.

17

Fibonacci function

- Let's write an R function to calculate the value of the Fibonacci sequence for a given n .
- ```
fibonacci =
function(n) {
 if(n == 0 || n == 1)
 return(n)

 fibonacci(n - 1) + fibonacci(n - 2)
}
```
- This is nice and simple.  
The function calls itself - recursion.

18

# Basics of Recursion

- The function
  - calls itself  
with a different argument!
  - does some computations to solve the simple or special cases on the original argument, e.g.  $n = 0, 1$ .
- Any recursive algorithm can be written in an iterative manner - i.e. using loops.

19

# Iterative Fibonacci

```
fib2 =
function(n)
{
 if(n == 0 || n == 1) return(n)
 if(n == 2) return(1)

 f1 = f2 = 1
 for(i in seq(2, n-1)) {
 f = f1 + f2
 f2 = f1
 f1 = f
 }
 f
}
```

20

- Let's compare these in terms of speed  
Which one will be faster?
- Can you compare them in your head or on paper?
- Or empirically  
Create a simple experiment to measure the time  
do 20 repetitions each calculating  $F_{20}$ .

21

- How do we measure time for a computation:  
`system.time(command)`
- Get back a vector with 5 elements:  
user time, system time, cpu time (and sub-processes)

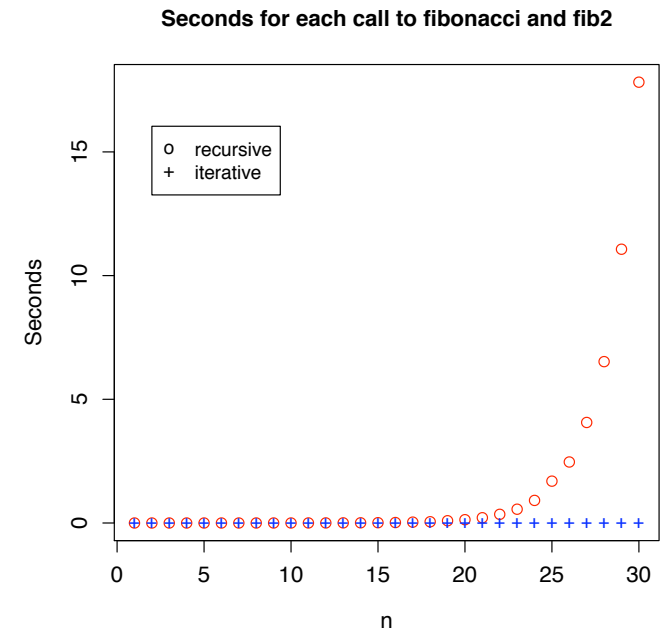
22

- `system.time(fib2(20))`  
[1] 0 0 0 0 0
- Finite resolution that depends on the operating system.  
(usually 1/100 second)
- So repeat the calculations many times to get longer  
times. Then divide by the number of times performed.
- `fib2.times =`  
`system.time(sapply(1:1000, function(x) fib2(20)))`  
[1] 0.10 0.00 0.11 0.00 0.00
- So total time of .11 seconds, per call .11/1000

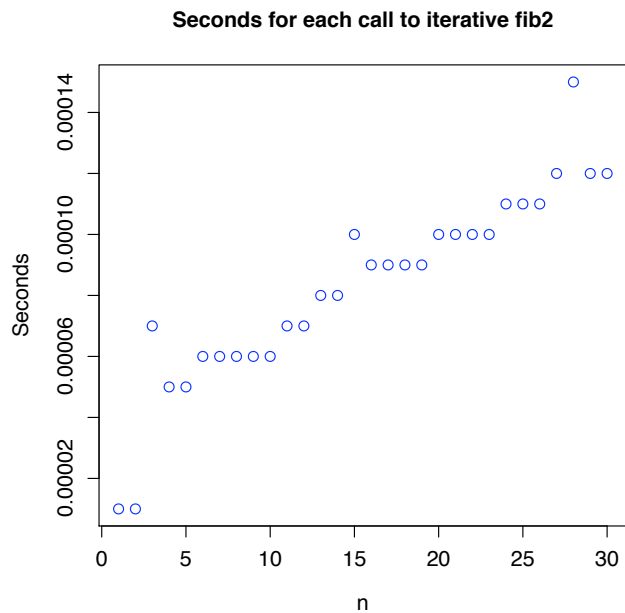
23

- ⦿ The iterative version is much, much faster as n gets big.
- ⦿ Repeat it for various values of n, e.g. 1, 2, ..., 30
- ⦿ Then plot n against time taken and see if you see any relationship.
- ⦿ `fib2.times =`  
`sapply(1:30,`  
`function(i)`  
`system.time(sapply(1:1000,`  
`function(x) fib2(i))))`
- ⦿ Same for fibonacci, and dynFibonacci.

24



25



When plotted on the appropriate scale, it is approximately linear.

26

## Dynamic Programming

- ⦿ Another approach is to use the simple recursive algorithm but to store the values we have previously computed. Access these in subsequent calls.

27

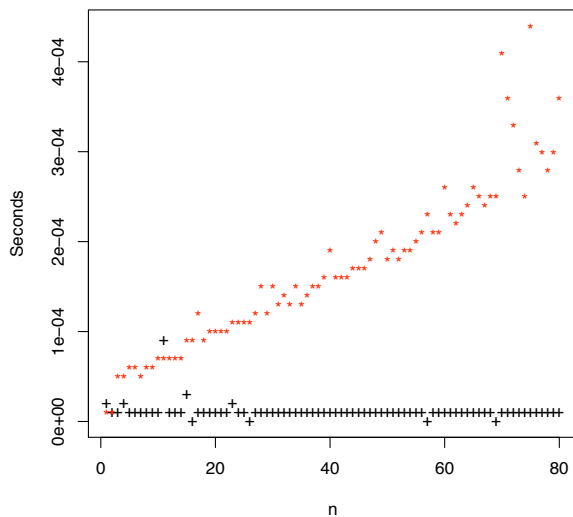
# Dynamic Fibonacci

```
.Fibonacci<- c(1, 1)
dynFibonacci <-
function(n)
{
 top = length(.Fibonacci)
 if(top >= n)
 return(.Fibonacci[n])

 for(i in seq(top + 1, n)) {
 .Fibonacci[i] <- .Fibonacci[i - 1] + .Fibonacci[i - 2]
 }
 .Fibonacci[n]
}
```

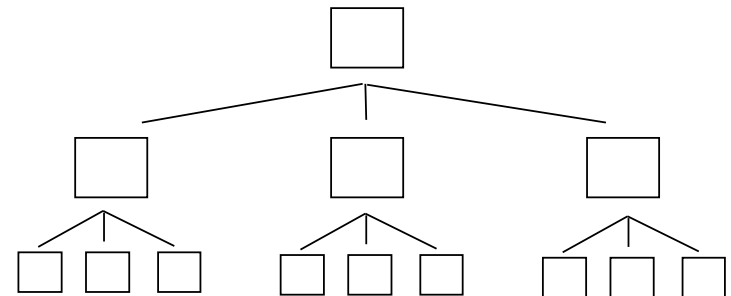
- Call dynFibonacci(10), then dynFibonacci(6) and it is already calculated and stored.
- dynFibonacci(20) can start from the previous highest element, i.e.  $F_{10}$ .
- This is called memoization and is essentially Dynamic Programming.  
Solve a problem by solving smaller problems and store the results for these smaller problems for repeated reuse.

Time per call to fib2 and dynFibonacci



# More Recursion

- Suppose we have a document, e.g. HTML.
- And it has links to other documents and these have links to other documents.



# Recursive links

- ◉ To fetch all the links, again we can use a recursive algorithm.
- ◉ `getLinks =`

```
function(doc) {
 links = extractLinksFromDoc(doc)
 for(i in links) {
 # i is now the link.
 links = c(links, getLinks(i))
 }
}
```
- ◉ `getLinks()` calls `getLinks()`, so recursive.
- ◉ How would we go about getting the links in an HTML document?

32

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>XML Package for R and S-Plus</title>
</head>

<body>
<h1>An XML package for the S language</h1>
<p align=right>Source: XML_0.97-0.tar.gz
<p align=right>Windows binary: XML_0.97-0.zip
<p align=right>Last Release: Thu Aug 12 07:56:21 PDT 2004
<h2>Documentation</h2>
<dl>
 <dt>
 A reasonably detailed overview
 of the package and what we might use XML for.
 <dd>
 <dt>
 A manual in
 and a quick guide to the package (PDF).
 <dd>
</dl>
</body>
```

# Recursive data structure

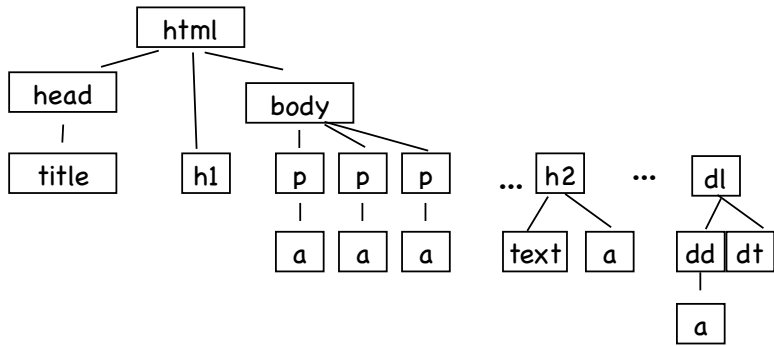
- ◉ Each node has
  - ◉ a name, e.g. `h1`, `a`, `title`
  - ◉ some attributes, e.g. `href="..."`
  - ◉ children nodes which have the same structure
- ◉ This is recursive because we have nodes nested within nodes, i.e. the children within parent nodes.

34

# How to extract links.

- ◉ Regular expressions?
- ◉ `htmlTreeParse()` in the XML package on CRAN
- ◉ This actually breaks the HTML into tokens or elements and creates a tree of nodes with children nodes.
- ◉ So we can walk or traverse that tree with a function that processes the node and then moves onto each of the children nodes.

35



This is the basic structure of the tree:  
node and its children

```

extractLinks =
function(node) {
 links = character()

 # process this node
 if(xmlName(node) == "a") {
 links = c(links, xmlGetAttr(node, "href"))
 }

 # now the children
 for(i in xmlChildren(node)) {
 links = c(links, extractLinks(i))
 }

 links
}

```