Notes on Recursion

Jake Elmstedt

5/5/2020

What is Recursion?

Recursion in programming is a paradigm where you solve a problem by breaking it into ever smaller pieces, The idea being, the answer to the whole problem could easily be answered if you knew the answer to a simpler, smaller problem. You proceed asking smaller and smaller versions of the question until you get to the simplest (base) case for which you know the answer.

A simple example of this you might be familiar with is calculating Fibonacci numbers, where finding the n^{th} Fibonacci number is trivial if we knew the $(n-1)^{th}$ and the $(n-2)^{th}$. The base case here we will say is `1`.

So, what does this look like in practice?

Example

Recall the Fibonacci sequence is $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$


```
recursive_fibo <- function(n) {
  if (n <= 2) {
    1
  } else {
    recursive_fibo(n - 1) + recursive_fibo(n - 2)
  }
}
```

So, here we have two base cases, when `n = 2` and when `n = 1` (we need two base cases because we need two initial values to compute a third).

In each of those cases we return `1`, in anything other than a base case we return the sum of two smaller cases.

This particular case works well for small values of `n`, but it quickly becomes impossible to do computationally as `n` grows, we'll see a modification to this shortly which solves this issue.

Let's look at some other simple examples first.

Some Simple Examples

Finding the Length of a Vector

There are many ways to get the length of a vector, but this recursive approach is perhaps my favorite.

Idea

The recursive idea is the length is always going to be one more than the length of the vector without its first element. The base case will be `0`, but we can't use `length()` so we'll need another way to know when we have a length-0 vector. We'll use the idea that, if we remove an element from a length-0 vector, it is still a length-0 vector. e.g. `identical(x, x[-1])` will return `TRUE` when `x` is length-0.

Code

```r
recursive_length <- function(x) {
  if (identical(x, x[-1])) {
    0
  } else {
    1 + recursive_length(x[-1])
  }
}
recursive_length(integer(0))
```



```
## [1] 0
```



```r
recursive_length(1:4)
```

```
## [1] 4
```



```r
recursive_length(1:100)
```

```
## [1] 100
```


Finding the Minimum of a Vector

There are many ways to get the length of a vector, but this recursive approach is perhaps recursive favorite.

Idea

The recursive idea is the minimum of a vector with be the minimum of a vector with the larger of the the first or the second element removed. The base cases will be when `x` is length-1 or length-0, we're just going to return `x` or `Inf` respectively.

Code

```r
recursive_min <- function(x) {
  if (recursive_length(x) == 0) {
    Inf
  } else if (recursive_length(x) == 1) {
    x
  } else {
    if (x[1] > x[2]) {
```

```
      recursive_min(x[-1])
    } else {
      recursive_min(x[-2])
    }
  }
}
recursive_min(integer(0))
```

```
## [1] Inf
```

```
recursive_min(7)
```



```
## [1] 7
```

```
recursive_min(c(23, 7, 13, 5, 19, 19, 37))
```



```
## [1] 5
```

Finding the Sum of a Vector (1)

Idea 1

The recursive idea is the sum of a vector will be the result of the addition of the the first element of `x` and the sum of all of the elements of `x` except the first. The base case will be when `x` length-0, we're just going to return `0`.

Code


```
recursive_sum1 <- function(x) {
  if (recursive_length(x) == 0) {
    0
  } else {
    x[1] + recursive_sum1(x[-1])
  }
}
recursive_sum1(numeric(0))
```

```
## [1] 0
```

```
recursive_sum1(1:3)
```

```
## [1] 6
```

```
recursive_sum1(1:10)
```

```
## [1] 55
```

Finding the Sum of a Vector (2)

Idea 2

The recursive idea is the sum of a vector will be the result of the addition of the sums of two halves of that vector. The base cases will be when `x` is length-1 or length-0, we're just going to return `x` or `0` respectively.

Code

```
recursive_sum2 <- function(x) {
  if (recursive_length(x) == 0) {
    0
  } else if (recursive_length(x) == 1) {
    x
  } else {
    idx <- seq_along(x) <= recursive_length(x) / 2
    recursive_sum2(x[idx]) + recursive_sum2(x[!idx])
  }
}
recursive_sum2(numeric(0))
```



```
## [1] 0
```

```
recursive_sum2(1:3)
```



```
## [1] 6
```

```
recursive_sum2(1:10)
```



```
## [1] 55
```

Reversing the Order of a Vector

Idea

The recursive idea is we can reverse the order of a vector if we attach the first element of the vector to the reverse of the vector without the first element. The base case is when `x` has a length less than or equal to `1` we will simply return `x`.

Code

```
recursive_rev <- function(x) {
  if (recursive_length(x) <= 1) {
    x
  } else {
    c(recursive_rev(x[-1]), x[1])
  }
}
recursive_rev(character(0))
```

```
## character(0)
```

```r
recursive_rev(1:10)
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```r
recursive_rev(c(3, 1, 4))
```

```
## [1] 4 1 3
```

More Advanced Examples

Recursion with Memory

Sometimes when we do recursion, like with our Fibonacci example, we end up computing the same result over and over again (see if you can figure out how many total function calls `recursive_fibo(6)` runs then try to guess how many `recursive_fibo(60)` will try to run, the answer is here[1]).

We can solve this by keeping track of what we've done already and build up rather than down, we're also going to create a wrapper function around the recursion (this is often a good practice in general for reasons I won't get into here).

Idea

The $(n-1)^{th}$ and n^{th} Fibonacci numbers are the n^{th} and the sum of the $(n-1)^{th}$ and n^{th} Fibonacci numbers. The base case is the 1^{st} and 2^{nd} Fibonacci numbers. Finally, when we have the $(n-1)^{th}$ and n^{th} Fibonacci numbers, we will just return the n^{th}.


```r
recursive_fibo2 <- function(n) {
  recursive_worker <- function(n, fib_seq = c(1, 1)) {
    if (n <= 2) {
      fib_seq
    } else {
      fib <- recursive_worker(n - 1, fib_seq)
      c(fib[-1], sum(fib))
    }
  }
  recursive_worker(n)[2]
}
recursive_fibo2(5)
```

```
## [1] 5
```

```r
recursive_fibo2(10)
```

```
## [1] 55
```

[1] `recursive_fibo(6)` will do 15 total function calls while `recursive_fibo(60)` will need to do 3.1×10^{12}, it will effectively take forever.

```r
recursive_fibo2(60)
```

```
## [1] 1.548009e+12
```

Recursive Matrix Multiplication

Idea

The recursive idea is that matrix multiplication can be done as a series of "block" multiplications.

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \times \left[\begin{array}{c|c} 1 & 2 \\ \hline 3 & 4 \end{array}\right] = \left[\begin{array}{c|c} A \times 1 + B \times 3 & A \times 2 + B \times 4 \\ \hline C \times 1 + D \times 3 & C \times 2 + D \times 4 \end{array}\right]$$

So, we will start out by zero padding both matrices so they are equi-dimensional square matrices with dimension equal to some power of 2. Then each of the four quadrants is the sum of two smaller matrix multiplications. Our base case happens when each sub matrix is dimension 1×1 and we simply return `x * y`. We add one small optimization where if all of the values in `x` or all of the values in `y` are `0`, we just return the appropriately sized 0-matrix.

Code


```r
pad_mat <- function(x, n) {
  xpads <- n - dim(x)
  cbind(
    rbind(x,
          matrix(0, ncol = ncol(x), nrow = xpads[1])),
    matrix(0, ncol = xpads[2], nrow = n))
}

`%r%` <- function(x, y) {
  x <- as.matrix(x)
  y <- as.matrix(y)
  if (ncol(x) != nrow(y)) {
    stop("x and y are non-conformable.")
  }
  odim <- c(row = nrow(x), col = ncol(y))
  n <- 2^ceiling(log(max(dim(x), dim(y)), base = 2))

  x <- pad_mat(x, n)
  y <- pad_mat(y, n)

  if (n == 1) {
    x * y
    } else if (all(x == 0) | all(y == 0)) {
      # Gives major performance boost when many added cols.
      matrix(0, odim["row"], odim["col"])
  } else {
    a <- seq_len(ncol(x)) <= n / 2
    top <- cbind(x[a, a] %r% y[a, a] + x[a, !a] %r% y[!a, a],
```

```r
                 x[a, a] %r% y[a, !a] + x[a, !a] %r%  y[!a, !a])
    bot <- cbind(x[!a, a] %r% y[a, a] + x[!a, !a] %r% y[!a, a],
                 x[!a, a] %r% y[a, !a] + x[!a, !a] %r% y[!a, !a])
    rbind(top, bot)[seq_len(odim["row"]), seq_len(odim["col"])]
  }
}

x <- matrix(1:15, nrow = 3)
y <- matrix(1:15, nrow = 5)
x %r% y
```

```
##      [,1] [,2] [,3]
## [1,]  135  310  485
## [2,]  150  350  550
## [3,]  165  390  615
```



```r
x %*% y
```

```
##      [,1] [,2] [,3]
## [1,]  135  310  485
## [2,]  150  350  550
## [3,]  165  390  615
```


Final Thoughts

Recursion is amazing, elegant, and sometimes beautiful. It won't be the solution to all of your problems and there will almost always be a more straight forward solution. But, when recursion *is* the right way to go, there's nothing else like it.

In almost all cases, you shouldn't try to think of a recursive solution to your problems first (or even second, third, or fourth), but if you find yourself with a problem where the answer can be found by looking at the answer to a simpler, smaller problem, it can be a really clever and fun way to go.