Notes - Algorithms & Data Structures - Recursion

- Dr Nick Hayward

A brief intro to *recursion*, and its general use in algorithms and programming.

Contents

Intro

As we see with data structures, including custom structures such as a linked list, access and iteration is a key consideration for general use, and effective re-use.

For example, we have no existing index for each item in the linked list. However, we may choose to use a pattern such as *recursion* to check and access the list.

Recursion is a common technique used in the design of many algorithms, and app development in general.

A key benefit of recursion is the option to define a *base* case, and a useful *recursive* case to help solve a given problem. In effect, recursion commonly provides an elegant way to solve complex problems.

However, its usage may also be seen as somewhat divisive, and controversial depending upon context.

Recursion for fun

If we consider a jar of 10,000 sweets with various colours, but only a single winning *gold* sweet, we might design an algorithm with recursion.

Initially, we have two procedures we may define

- pick a sweet

- check the sweet's colour

The second procedure may also be used to check if the sweet's colour matches the prize *gold* sweet, thereby defining whether we need a recursive call to the first procedure or the process has finished.

required steps

So, we might outline the required steps to achieve the overall process of finding the gold sweet. Each of the above procedures will include various tasks to help resolve the overall process.

For example,

1. open the jar of sweets to begin the search
2. choose a sweet from the jar and check its colour
3. if the sweet is *not* gold, add it to a second jar
4. if the sweet is *gold*, the prize has been found and the process ends
5. repeat...

As we noted above, we may define a general series of steps as follows,

1. check each sweet in the jar
2. if the sweet is *not* gold...repeat step 1
3. if the sweet is *gold*, you win the prize...

So, we can see the difference between these initial approaches to solving the same problem.

an initial example - iteration

The first example might use a simple `while` loop, e.g.

- `while the jar of sweets is not empty, choose a sweet and check its colour...`

Whilst the second example uses *recursion*. In effect, we keep calling the first step, or function, until a break is achieved.

We might consider this problem using two sample implementations, which reflect the above outlines.

The first example uses a `while` loop, which might be outlined as follows using pseudocode,

```
search_sweets(main_jar)
  while main_jar is not empty
    sweet = main_jar.pick_a_sweet()
    if sweet.is_not_gold()
      second_jar.add_sweet(sweet)
    else
      print "gold sweet found, you win!"
      exit
```

So, whilst the main sweet jar still contains sweets, the `pick_a_sweet()` function will choose a sweet. This function will need to return the chosen sweet, so we may check its colour, and remove it from the main jar.

We may then check the current sweet's colour, and add it to the second jar if it's not *gold*. If the sweet is gold, you win and the loop will exit.

A basic example in JavaScript is as follows

```
// FN: search passed jar of sweets
function searchSweets(main_jar) {
    // declare
    const second_jar = [];
    while (main_jar.length > 0) {
        // pop last item in main_jar array - or use shift() for first item...
        const sweet = main_jar.pop();
        // check if sweet is gold
        if (sweet !== 'gold') {
          console.log(`${sweet} sweet is not gold...`);
          // if not gold, add to second jar
          second_jar.push(sweet);
        } else {
          // you win...gold sweet found in main jar
          console.log(`you win, ${sweet} sweet found!`);
          // exit loop...
          return;
        }
    }
}

// define main jar with variety of sweets
const main_jar = ['blue', 'green', 'red', 'orange', 'gold', 'yellow', 'pink'];
// check main jar for a gold sweet...
searchSweets(main_jar);
```

We're able to loop through the passed jar of sweets, and check each one until we find the winning gold sweet.

We make a number of assumptions, such as the passed jar as an array, and the value of the sweet's colour as a string.

It's also a slow search as we only have information for the required colour of the winning sweet. So, we need to iterate through the whole jar.

an initial example - recursion

For the second implementation, we may consider a solution using recursion.

We may initially consider the algorithm using pseudocode,

```
search_sweets(main_jar)
  if main_jar is not empty
    sweet = main_jar.pick_a_sweet()
    if sweet.is_not_gold()
      second_jar.add_sweet(sweet)
```

```
        search_sweets(main_jar)
      else gold sweet found
    else jar is empty
```

The recursive example follows the same underlying pattern of checks and balances we saw for the `while` option. However, instead of loop we may now call the `search_sweets()` method for all the sweets in the jar, or until we find the gold sweet.

We may then implement this algorithm using recursion with JavaScript

```javascript
// FN: search passed jar of sweets
function searchSweets(main_jar) {
    // declare second_jar for removed sweets...
    const second_jar = [];
    // check main_jar has sweets left...
    if (main_jar.length > 0) {
        // get a sweet and remove from main_jar
        const sweet = main_jar.pop();
        // check sweet colour - gold wins prize...
        if (sweet !== 'gold') {
          console.log(`${sweet} sweet is not gold...`);
          // if not gold, add to second jar
         second_jar.push(sweet);
         // recursive call - pass remainder of main_jar
         searchSweets(main_jar);
       } else {
         // you win...gold sweet found in main jar
        console.log(`you win, ${sweet} sweet found!`);
      }
    } else {
        // main_jar is empty - no gold sweet found...
        console.log(`jar is now empty...you lose, try again!`);
    }1,

}

// define main jar with variety of sweets
const main_jar = ['blue', 'green', 'red', 'orange', 'golden', 'yellow', 'pink'];
// check main jar for a gold sweet...
searchSweets(main_jar);
```

So, we still need to check the availability of sweets in the jar, which then allows us to check for the winning gold sweet.

The conditional statements follow the same pattern as the previous JavaScript example, but we may now recursively call the `searchSweets()` function.

Recursion in practice - linked list

If we consider a linked list, for example, we may define an algorithm for implementing procedures on a list using recursion.

For example, the following algorithm may be outlined to find the last element of a defined list,

```
last(list) {
  if ( isEmpty(list) )
    error('error - list empty...')
  else if ( isEmpty(rest(list)) )
    return first(list)
  else
    return last(rest(list))
}
```

If we consider the *complexity* of this algorithm, we may note that the procedure has **linear time complexity**. i.e. if the length of the list is increased, execution time will likewise increase by the same factor.

This performance does not mean that lists are always inferior to arrays. However, we may note that lists are not an ideal data structure, regardless of applied common algorithms, when an application needs to access the last element of a longer list.

Recursion and the call stack

A stack may be used, for example, to represent execution logic for a recursive function.

For example, if we consider the following Python code for calculating the factorial of a passed number,

- an example for `3!` - factorial(3)

```python
def factor(x):
   if x == 1:
       return 1
   else:
       return x * factor(x-1)

print(factor(3))
```

we may start to discern a pattern to recursive calls and the call stack they use.

call stack - app execution

As we call the function with a passed value of `3`, we may outline the call stack as follows

- the inital value of `x` is the passed value of `3`
    - `x = 3`

```
------------
|  factor  |
|----------|
| x  |  3  |
------------
```

- then we check `x` against a value of `1`
    - not `1`
    - move to else
    - return `x` multiplied by `factor(x-1)` - first *recursive* call
        - `factor(2)` is added to the call stack and executed

```
------------
|  factor  |
|----------|
| x  |  2  |
------------
|  factor  |
|----------|
| x  |  3  |
------------
```

```
------------
|  factor  |
|----------|
| x  |  2  |  ----
------------     |
|  factor  |     |-- n.b. both calls have a variable `x` with different values
|----------|     |
| x  |  3  |  ----
------------
```

- we're now executing the top of the stack - `factor(2)`
    - `x = 2`
    - check `x` against a value of `1`
    - move to else
    - return `x` multiplied by `factor(x-1)` - second *recursive* call
        - `factor(1)` is added to the call stack and executed
- we now have three calls in the stack

```
------------
|  factor  |
|----------|
| x  |  1  |  ----
------------     |
|  factor  |     |-- n.b. value cannot be accessed outside function context
|----------|     |
| x  |  2  |  ----
------------     |
|  factor  |     |-- n.b. both calls have a variable `x` with different values
|----------|     |
| x  |  3  |  ----
------------
```

- we're now executing the top of the stack - `factor(1)`
  - `x = 1`
  - we can now return `1`
  - pop `factor(1)` from call stack
  - this is the first call we may `return` from...
- now return to second recursive call
  - `return 2 * 1`
  - pop `factor(2)` from call stack
- now return to first recursive call
  - `return 3 * 2`
  - pop `factor(3)` from call stack
- print `6`

pattern of execution

We may see the following pattern of execution for this function,

```
factor(3)
    x = 3
    return x * factor(3-1) // recurse 1
    factor(2)
        x = 2
        return x * factor(2-1) // recurse 2
        factor(1)
            x = 1
            return 1 // pop factor(1) from call stack
        return 2 * 1 // 1 is returned from recurse 2
        return 2 // pop factor(2) from call stack
    return 3 * 2 // 2 is returned from recurse 1
    return 6 // pop factor(3) from call stack

print 6 // stack now clear, execution ends...
```

We can see how useful a stack is to the execution of recursion. It may be used as a record of execution, and a clear order of remaining execution.

In effect, the call stack acts as a record of half-completed function calls, each with its own record of incomplete execution waiting to finish.

Obviously, a key benefit to this stack usage is that we do *not* need to keep a manual record of executed and incomplete function calls, the call stack does this for us.

Recursion and memory costs

Whilst using the call stack is convenient, in particular for recursive usage, it does come with a cost.

Adding such records to the call stack requires memory usage, and this can fill quickly when we use recursive function calls.

In such situations where memory is causing an application's execution to freeze or crash, we might consider the following

- consider modifying recursion to iteration
- use a cache for certain functions and function calls - i.e. *memoisation*
    - identify duplicate calculations, calls...
- use an option such as *tail recursion*

tail recursion

An example of tail recursion for `3!` - factorial(3)

```
def factor(x, tail):
    print("factor x =",x)
    if x == 1:
        print("return from (x == 1) = 1")
        return tail
    else:
        print("x =",x)
        return factor(x - 1, x * tail)

# set initial tail to 1
print(factor(3, 1))
```

So, we may see a pattern of execution and call stack usage for tail recursion as follows,

```
factor(3, 1)
    x = 3, tail = 1
    return factor(3 - 1, 3 * 1) // recurse 1
    factor(2, 3)
        x = 2, tail = 3
        return factor(2 - 1, 2 * 3) // recurse 2
        factor(1, 6)
            x = 1, tail = 6
            return 6 // pop factor(1, 6) from call stack
        return 6 // pop factor(2, 3) from call stack
    return 6 // pop factor(3, 1) from call stack

print 6 // stack now clear, execution ends
```

Recursion and the call stack - JavaScript example

The following JavaScript code example uses a *call stack* to ensure expected execution,

```
function findSolution(target) {
    function find(current, history) {
        if (current == target) {
            return history;
        } else if (current > target) {
            return null;
        } else {
            return find(current + 5, `(${history} + 5)`) || find(current * 3,
`(${history} * 3)`);
        }
    }
    return find(1, "1");
}

console.log(findSolution(24));
```

The initial `findSolution()` function is called with the passed parameter of `24`, the value to check.

This function returns an executed `find()` function with initial test values for current and history.

As part of this function's execution, it checks these initial values until it reaches the `else` part of the conditional statement.

This returns the `find()` function, called recursively by itself, initially checking against an addition of 5. It will continue to check possible values with `+ 5` until it either succeeds or moves onto the right side of logical OR, `||`, and a check with `* 3`.

Again, it will either succeed or fail with these recursive checks.

However, the structure that permits this recursion to execute is the *call stack*. It provides a defined pattern to execution, which allows the code to run as expected.

Recursion for fun - Fibonacci

A fun way to test recursion and stacks (i.e. call stack) is the problem of searching the Fibonacci series of numbers.

We might begin by testing our patterns and algorithm design with a fun example such as the *Fibonacci* series.

The Fibonacci series is simply an ordered sequence of numbers where each number is the sum of the preceding two.

e.g.

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

You might also see the series beginning with 1 instead of 0.

So, the function should return the `n-th` entry in this sequence. e.g. 5th index entry will return `5`.

Fibonacci may be solved using various techniques and algorithms.

It's also a good test of runtime speed and complexity for options such as iterative and recursive solutions.

iteration example

We might initially test an iterative solution to check and return values in the Fibonacci series.

For example, an iterative solution may be as follows,

```
function fib(n) {
  // pre-populate array - allow calculation with two initial values
  const result = [0, 1];
  // i starts at index 2...
  for (let i = 2; i <= n; i++) {
    // get the previous two results in array
    const a = result[i-1];
    const b = result[i-2];
    // calculate next value in series & push to result array
    result.push(a + b);
  }
  // get result at specified index posn in series...
  return result[n-1]; // -1 due to array index starting at 0...
}
// log to console...
console.log('index posn 8 in fibonacci series = ', fib(8));
```

recursion example

We may also consider a solution using *recursion*.

For example,

```
function fib(n) {
  // base case
  if (n < 2) {
    console.log(n);
    return n;
  }
  // dynamic calculation of number in sequence
  return fib(n-1) + fib(n-2);
}

console.log('index posn 5 in fibonacci series = ', fib(5));
```

If we add some logging for this recursion, e.g.

```
function fib(n, r) {
  console.log(`n = ${n} and r = ${r}`);
  // base case
  if (n < 2) {
    console.log(n);
    return n;
  }
  // dynamic calculation of number `n` in sequence and recursive call `r`...
  return fib(n-1, 1) + fib(n-2, 2);
}

console.log('index posn 5 in fibonacci series = ', fib(5, 0));
```

we may see the following output to help track the recursive calls and addition.

e.g.

```
n = 5 and r = 0
n = 4 and r = 1
n = 3 and r = 1
n = 2 and r = 1
n = 1 and r = 1
return base = 1
n = 0 and r = 2
return base = 0
n = 1 and r = 2
return base = 1
n = 2 and r = 2
n = 1 and r = 1
return base = 1
n = 0 and r = 2
return base = 0
n = 3 and r = 2
n = 2 and r = 1
n = 1 and r = 1
return base = 1
n = 0 and r = 2
return base = 0
n = 1 and r = 2
return base = 1
index posn 5 in fibonacci series =  5
```

So, this recursive pattern may be defined as follows

```
fib(5)
    n = 5
    return fib(5-1) + fib(5-2) // recurse
    fib(5-1)
        n = 4
        return fib(4-1) + fib(4-2) // recurse
        fib(4-1)
            n = 3
            return fib(3-1) + fib(3-2) // recurse
            fib(3-1)
                n = 2
                return fib(2-1) + fib (2-2) // recurse
                fib(2-1)
                    n = 1
                    return 1 // base returned - recurse
                fib(2-2)
                    n = 0
                    return 0 // base returned - recurse
            fib(3-2)
                n = 1
                return 1 // base returned - recurse
        fib(4-2)
            n = 2
            return fib(2-1) + fib(2-2) // recurse
            fib(2-1)
                n = 1
                return 1 // base returned
            fib(2-2)
                n = 0
                return 0 // base returned
    fib(5-2)
        n = 3
        return fib(3-1) + fib(3-2) // recurse
        fib(3-1)
            n = 2
            return fib(2-1) + fib(2-2) // recurse
            fib(2-1)
                n = 1
                return 1 // base returned
            fib(2-2)
                n = 0
                return 0 // base returned
        fib(3-2)
            n = 1
            return 1 // base returned

return 5 // sum return values for base
```

If we follow the pattern of recursion and base case returns, we can see the return values needed to calculate index position 5 in the Fibonacci series. i.e.

```
// Fibonacci series to index 5
[0,1,1,2,3,5]
```

fibonacci recursion - pattern of execution

So, why does this JavaScript recursive solution actually work as expected?

As the function is called recursively, it only returns a value for the *base case*, i.e. either 0 or 1.

As it works down from the value of the passed `n-th` position in the series, it is storing each return, and then returns the total for that position in the series.

For the current JavaScript example, we may consider the execution of functions. For example, the function where another function is called will be paused whilst the inner execution is completed. In effect, outer will be paused as inner is executed.

This recursive solution will produce an *exponential time* for the complexity.

i.e. as the `n-th` value increases so will the time required to find a value in the series...

We may commonly define the complexity for a recursive solution as exponential O(2^n).

Improvements may be made to this recursive algorithm using *memoisation*, for example.

This is due to repetitive calls to the same values for `fib()`, e.g. multiple calls to `fib(3)`.

We can store the arguments of a given function call along with the computed result.

e.g. when `fib(4)` is first called, the computed value will be stored in memory, a temporary cache in effect.

Then, we can simply call the stored return each and every subsequent call to `fib(4)`.

improve performance - memoisation

We can now improve the performance of the recursive algorithm for the Fibonacci series by adding support for *memoisation*.

As such, we may abstract this functionality to a separate, re-usable *memoisation* function. We may then use this function whenever we need to add memoisation to an algorithm, application &c.

The main part of this function will record used and repeated functions &c., which we may then call again as needed.

In effect, a *cache* for the function. This is how we derive the speed improvement for the passed function.

For example, we may define the initial *memoisation* function as follows,

```
// pass original function - e.g. slow recursive fibonacci function
function memoise(fn) {
  // temporary store
  const cache = {};
  // return anonymous function - use spread operator to allow variant no. args
  return function(...args) {
    // check passed args in cache - if true, return cached args...
    if (cache[args]) {
      return cache[args];
    }
    // no cached args - call passed fn with args
    const result = fn.apply(this, args);
    // add result for args to the cache
    cache[args] = result;
    // return the result...
    return result;
  };
}
```

To improve the Fibonacci recursion dramatically, we may then use memoisation as follows,

```
function fib(n) {
  // base case
  if (n < 2) {
    console.log(n);
    return n;
  }
  // dynamic calculation of number in sequence
  return fib(n-1) + fib(n-2);

}

// reassign memoised fib fn to fib - recursion then calls memoised fib fn...
fib = memoise(fib);
console.log('index posn 100 in fibonacci series = ', fib(100));
```

So, we're now able to check higher index values in the Fibonacci series.

For example, 100th position in the Fibonacci series is,

- 354224848179262000000

And, position 1000 is

- 4.346655768693743e+208