Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 7

Dr Nick Hayward

Fibonacci

- fun way to test recursion and stacks (i.e. call stack)
  - *problem of searching Fibonacci series of numbers*
- Fibonacci series is simply an ordered sequence of numbers
  - *each number is the sum of the preceding two...*
- e.g.

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

- might also see the series beginning with `1` instead of `0`
- function should return `n-th` entry in sequence.
  - *e.g. 5th index entry will return* `5`
- Fibonacci may be solved using various techniques and algorithms
  - *e.g. iteration and recursion...*
- a good test of runtime speed and complexity

Fibonacci - iteration example

- initially test an iterative solution
  - *check and return values in the Fibonacci series*
- e.g.

```
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));
```

- O(n) - linear time for iteration
  - *assuming constraints of memory for 64bit system*
- beyond memory bounds and complexity becomes quadratic
  - *$O(n^2)$ or O(n^2)*

Fibonacci - recursion example - part 1

- also consider a solution using *recursion*
- e.g.

```
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));
```

Algorithms and Data Structures

Fibonacci - recursion example - part 2

- 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));
```

Fibonacci - recursion example - part 3

- sample output to help track 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
```

Fibonacci - recursion example - part 4

- 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)
```

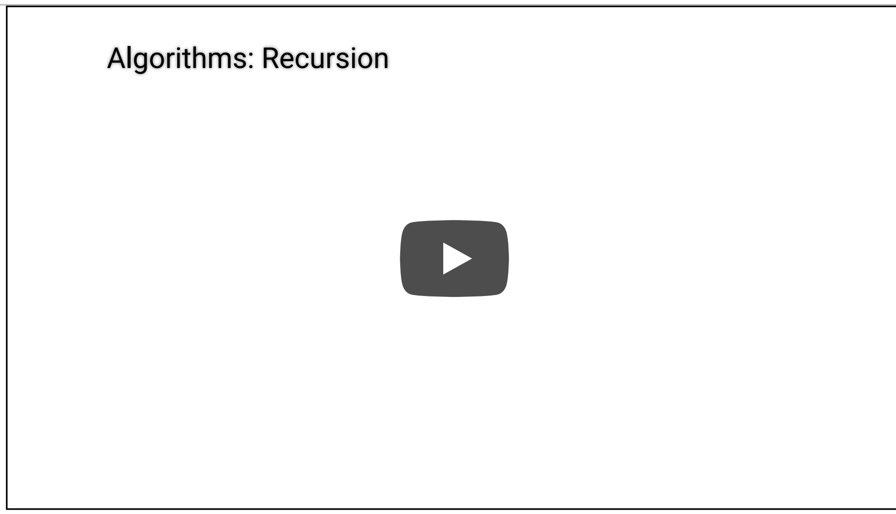
```
                n = 0
                return 0 // base returned
        fib(3-2)
            n = 1
            return 1 // base returned

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

- follow pattern of recursion and base case returns
  - *shows return values needed to calculate index position 5 in Fibonacci series*
  - *i.e.*

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

Video - Algorithms and Data Structures

Recursion and Fibonacci



Recursion - UP TO 4:30

Source - Recursion & Fibonacci - YouTube

Fibonacci - recursion example - part 5

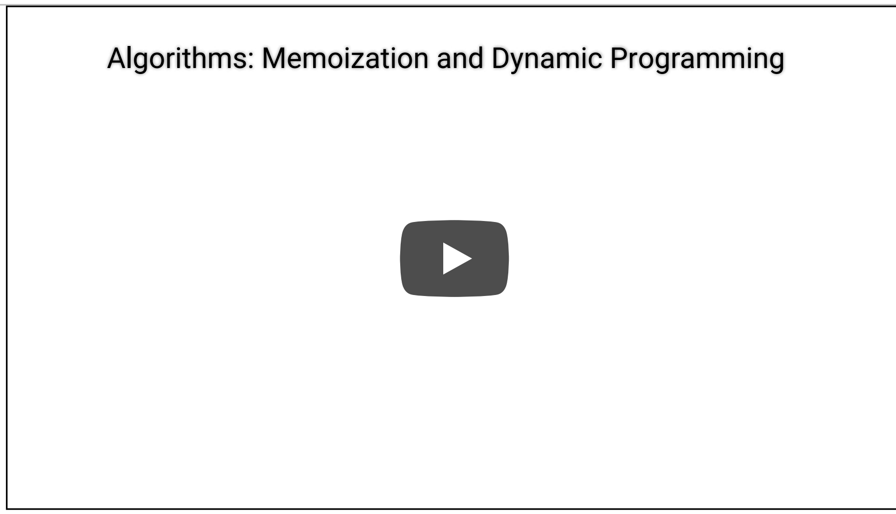
- why does this JavaScript recursive solution actually work as expected?
- as function is called recursively
  - *only returns value for base case*
  - *i.e. either 0 or 1*
- as it continues down from value of passed `n-th` position in series,
  - *it is storing each return*
  - *then returns total for that position in series*
- for current JavaScript example
  - *may consider execution of functions to better understand pattern*
- e.g. function where another function is called
  - *paused whilst inner execution is completed*
  - *i.e. outer will be paused as inner is executed...*

Fibonacci - recursion example - part 6

- recursive solution will produce an *exponential time* for the complexity
- i.e. as `n-th` value increases
  - *so will time required to find a value in the series...*
- commonly define complexity for a recursive solution as exponential
  - *O(2^n)*
- improvements may be made to this recursive algorithm
  - *e.g. using memoisation*
- due to repetitive calls to same values for `fib()`
  - *e.g. multiple calls to `fib(3)`*

Video - Algorithms and Data Structures

memoisation - part 1



What is Memoisation - UP TO 2:51

Source - Memoisation - YouTube

Fibonacci - memoisation - part 1

- store arguments of a given function call along with computed result
- e.g. when `fib(4)` is first called
  - *computed value will be stored in memory*
  - *a temporary cache in effect*
- then call stored return
  - *e.g. each and every subsequent call to `fib(4)`*

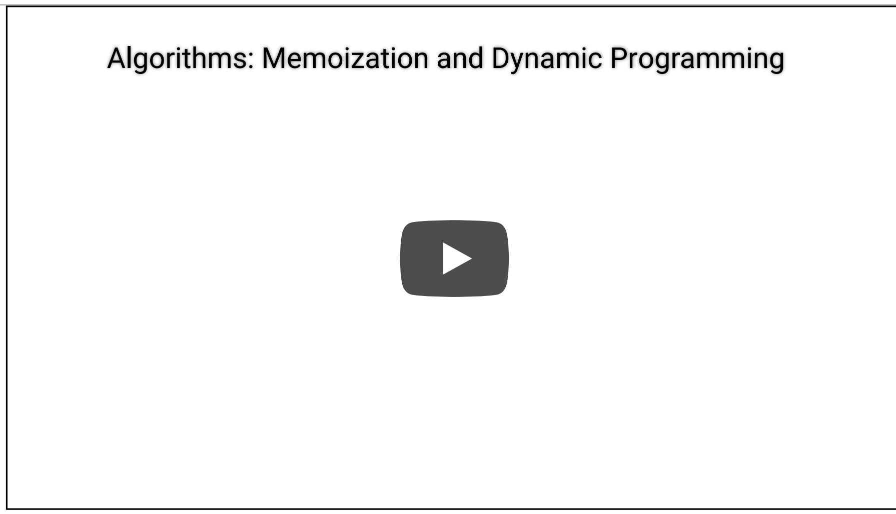
Algorithms and Data Structures

Fibonacci - memoisation - part 2

- now improve performance of recursive algorithm
  - *e.g. for Fibonacci series*
  - *add support for memoisation*
- abstract functionality to a separate, re-usable *memoisation* function
- then use this function to add memoisation to an algorithm, application &c.
- main part of function
  - *records used and repeated functions &c.*
  - *then call again as needed*
- i.e. a *cache* for the function
- derive speed improvement for passed function

Video - Algorithms and Data Structures

memoisation - part 2



Memoisation and Complexity - UP TO 4:12

Source - Memoisation - YouTube

Fibonacci - memoisation - part 3

e.g. define initial *memoisation* function

```
// 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;
  };
}
```

Fibonacci - memoisation - part 4

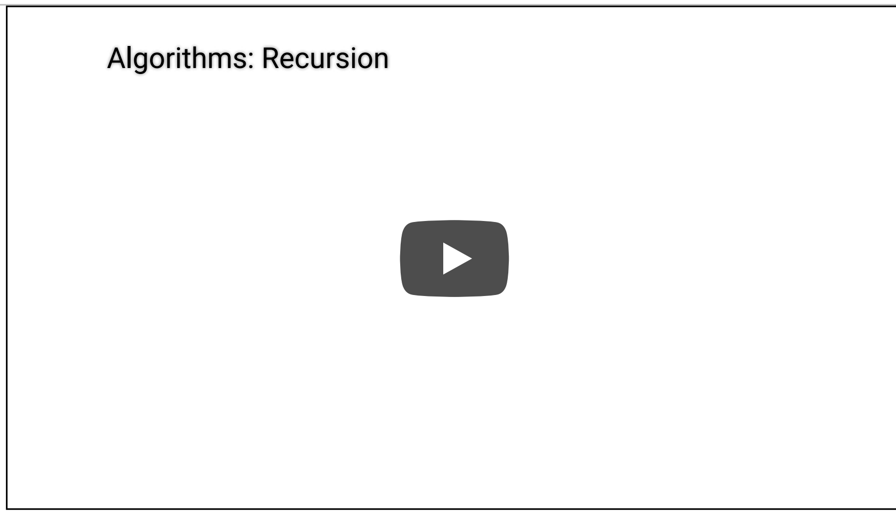
- use memoisation with Fibonacci function

```
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));
```

- now able to check higher index values in Fibonacci series
  - *without previous memory issues...*
- e.g. 100th position in the Fibonacci series is,
  - *354224848179262000000*
- position 1000 = `4.346655768693743e+208`

Video - Algorithms and Data Structures

Recursion and Fibonacci - memoisation



Recursion - UP TO END

Source - Recursion & Fibonacci - YouTube

divide and conquer - intro

- algorithms and development - often trying to solve a problem in a given context
- many techniques we may consider to solve a problem
  - *might start with a common option to help us get started...*
- *Divide and conquer* is a general technique
  - *e.g. used to solve various problems in application development and data usage*
- *Divide and conquer* is a well known *recursive* technique for solving various problems
  - *e.g. an option for analysing and solving such problems*
- consider use of *divide and conquer* from different perspectives
  - *use various examples to help outline its general usage...*

Video - Algorithms and Data Structures

Recursion & Divide and Conquer - part 1

Recursion and Divide and Conquer - UP TO 4:08

Source - Divide and Conquer - YouTube

Algorithms and Data Structures

divide and conquer - part 1

- start with a common example problem
  - *helps define basic structure and usage of divide and conquer*
- e.g. consider a parcel (plot or lot) of land
  - *need to sub-divide it evenly into **square** plots*
  - *need these plots of land to be as large as possible*
  - *fit all of the available space in original parcel of land*
- land has been measured to the following size
  - *1680 feet by 640 feet*
  - *approximately same as 6.74 Jumbo Jet planes in length*
  - *or 560 yd (a decent length par 5 in golf)*
- **n.b.** to solve this problem effectively
  - *can't simply divide this land in half - not two even squares*
  - *nor 20x20 squares, which are too small...*
- need to ensure we can always find maximum size for a square
  - *then divide the specified parcel of land...*

divide and conquer - part 2

- how do we calculate largest square
  - *i.e. largest used for a defined parcel of land*
- we may use *divide and conquer* to help solve this problem
- divide and conquer is a recursive technique
- divide and conquer algorithms are recursive algorithms...
- begin by defining two initial steps for the algorithm
  - *define base case - should be as simple as possible*
  - *divide and decrease underlying problem until it is base case*

divide and conquer - part 3

- consider the base case:
  - *begin by considering and defining base case for this algorithm*
  - *e.g.*

> *What is the largest possible square we may use to divide the land?*

- easiest base case for this type of problem might be as follows
  - *i.e. if one side was a multiple of the other side...*
- e.g. simple box of 50x50, which may be divided as two boxes of 25x25
  - *largest box we may use is 25x25*
  - *this meets requirement for defined base case as well...*

Algorithms and Data Structures

divide and conquer - part 4

- consider the recursive case:
- once we've defined a base case for the problem
  - *need to consider an appropriate recursive case to achieve base case*
- *divide and conquer* proves useful
  - *effectively reducing the problem to meet the base case*
- divide and conquer states - for each recursive call you need to reduce the problem
- for our land - `1680 feet by 640 feet`
  - *begin by marking largest boxes we may use to divide this size*
- e.g. two boxes of 640x640 and one remaining box of 640x400

```
        -------------------------
        |        |        |     |
    640 |        |        |     |
        |        |        |     |
        -------------------------
         640      640      400
```

- still have land measuring 640x400 to divide
- division of this land may now follow same underlying pattern as original land size
  - *i.e. find largest box to fill this remaining land of* `640x400`
- when we find this size
  - *define largest box for overall land of* `1680x640`
- problem has now been reduced from a land size of `1680x640` to `640x400`

divide and conquer - part 5

- now apply same algorithm to this problem - a land size of `640x400`
- largest box we may define is `400x400`
- still some land remaining after this division, `400x240`

```
    ------------
    |      |   |
400 |      |   |
    ------------
     400    240
```

- continue to apply this algorithm & reduce the problem as follows

```
    ------------
    |      |   |
240 |      |   |
    ------------
     240    160
```

- and then

```
    ------------
    |      |   |
160 |      |   |
    ------------
     160    80
```

- finally arrive at the base case...

```
   ---------------
   |      |      |
80 |      |      |
   ---------------
    80     80
```

- now have two evenly sized boxes of `80x80` with no land left over
  - *i.e. 80 is a factor of 160*
- we may sub-divide original land of `1680x640` into even plots of `80x80`

Video - Algorithms and Data Structures

divide and conquer - Euclid’s algorithm

Euclid and the Greatest Common Divisor - UP TO 8:27

Source - Algorithms - YouTube

divide and conquer - part 6

- we may summarise this use of divide and conquer as follows
  - *define a simple case for the base case*
  - *define how to reduce the problem to reach the base case*
- *divide and conquer*
  - *not itself an algorithm or reductive solution*
  - *can’t apply as is to solve a given problem…*
- *divide and conquer*
  - *give us a clear way of thinking about a problem to reach a solution*

divide and conquer - part 7

- e.g. if we consider following problem
  - *we may clearly see how useful this approach can be to defining an algorithm*
- e.g. for a defined data structure
  - *[6, 9, 13, 5, 11, 16]*
- need to add all of the values and return the total
- might simply use a loop to sum these values

```
def sum(data):
    total = 0
    for x in data:
        total += x
    return total

print(sum([6, 9, 13, 5, 11, 16]))
```

divide and conquer - part 8

- might define a solution using recursion for the same array of values
- e.g. define following steps to create a recursive algorithm to solve this problem
- Step 1 - define base case
  - *i.e. what's simplest array we may sum*
  - *e.g. an array of size 1 or 0 may be passed to the `sum()` function*
  - *this is easy to sum...base case*
- Step 2 - recursive calls
  - *need to reduce problem with each recursive call*
  - *i.e. move closer to defined base case*

Video - Algorithms and Data Structures

Recursion & Divide and Conquer - part 2

Recursion and Divide and Conquer - UP TO 7:53

Source - Divide and Conquer - YouTube

divide and conquer - part 9

- begin by considering how to sum values in a passed array
- e.g.

```
sum([6, 9, 13, 5, 11, 16])
```

- actually the same as

```
6 + sum([9, 13, 5, 11, 16])
```

- both examples return same summed value
- **n.b.** second example has started to reduce size of passed array
  - *now reduced size of problem*

divide and conquer - part 10

- define this algorithm as follows
  - *get the passed data*
    - e.g. array of numbers
  - *`if` data is empty*
    - `return` zero
  - *`else` total equals*
    - first number + rest of data
- check expected output as follows

```
sum([6, 9, 13, 5, 11, 16]) - sum = `60`
    6 + sum([9, 13, 5, 11, 16]) 6 + 54 = return `60`
        9 + sum([13, 5, 11, 16]) 9 + 45 = return `54`
            13 + sum([5, 11, 16]) 13 + 32 = return `45`
                5 + sum([11, 16]) - 5 + 27 = return `32`
                    11 + sum([16]) - 11 + 16 = return `27`
                    sum([16]) - base case & first return from execution - return `16`

print 60
```

divide and conquer - part 11

- now implement `sum()` function using divide and conquer with recursion

```
def sum(data):
    if len(data) == 1:
        return data[0]
    else:
        return data[0] + sum(data[1:])

print(sum([6, 9, 13, 5, 11, 16]))
```

Algorithms and Data Structures

divide and conquer - part 12

- JavaScript example 1

```
function sum(data) {
    if (data.length === 1) {
        return data[0];
    } else {
        // slice - return array from index 1 to end...
        return data[0] + sum(data.slice(1));
    }
}

console.log(sum([6, 9, 13, 5, 11, 16]))
```

- JavaScript example 2

```
function sum(data) {
    if (data.length === 1) {
        return data[0];
    } else {
        // destructure data - get head and return rest
        const [head, ...rest] = data;
        return head + sum(rest);
    }
}

console.log(`sum of values = ${sum([6, 9, 13, 5, 11, 16])}`);
```

Recursion & Divide and Conquer - part 3

Efficiency of Recursion and Divide and Conquer - UP TO 13:59

Source - Divide and Conquer - YouTube

sorting - quicksort - part 1

- a brief segue into a consideration of a sorting algorithm, *quicksort*
- faster search option than *selection sort*
  - *a common option for many real-world uses...*
- e.g. implementations of a `qsort` funtion in the C standard library.
- Quicksort also uses a pattern of *divide and conquer*
- e.g. use quicksort to sort our previous array of data
  - `[6, 9, 13, 5, 11, 16]`
- consider our last example of divide and conquer
  - *identified base case as simplest array we could sum*
- same may initially be considered relative to sorting
  - *i.e. clearly identify some arrays that do not need sorting*
- may define *base* case for such sorting as follows
  - `[]` *- empty array*
  - `[16]` *- array with one element*
- empty arrays and arrays with one element
  - *become base case for this sorting*
  - *i.e. return arrays as is without need to sort*

sorting - quicksort - part 2

- then consider an array with three elements
  - *again, use divide and conquer...*
- want to break this array down until we reach base case
  - *i.e. define quicksort as follows*
- *1. choose an element in array*
  - *element is **pivot***
- *2. **partition** the array*
  - *find elements less than pivot*
  - *find elements greater than pivot*
- now have two sub-arrays
  - *sub-array of all elements less than the pivot*
  - *sub-array of all elements greater than the pivot*
- these arrays will not initially be sorted
  - *just partitioned*
- when sub-arrays have been sorted
  - *combine them with pivot to return required sorted array*
- i.e.

```
sub_array[less_than] + pivot + sub_array[greater_than]
```

Algorithms and Data Structures

sorting - quicksort - part 3

- need a way to sort the sub-arrays
  - *where base case is useful again*
- i.e. Quicksort already knows how to sort arrays of two elements
- if we use quicksort with two sub-arrays
  - *then combine results*
  - *we now have a sorted array*
- e.g.

```
quicksort([less_than]) + [pivot] + quicksort([greater_than])
```

- this approach will work with any chosen pivot
- now define quicksort for an array of three elements
  - *choose a pivot*
  - *partition array into two sub-arrays*
    - elements less than pivot
    - elements greater than pivot
  - *recursively call quicksort on the two sub-arrays*

Video - Algorithms and Data Structures

quicksort - part 1

Quicksort - UP TO 3:25

Source - Quicksort - Java - YouTube

sorting - quicksort - part 4

- what happens if we now need to sort an array of four elements…
- use a similar, known pattern
- e.g. for an array of `[37, 12, 17, 9]` follow expected steps
  - *choose a pivot*
    - e.g. `37`
  - *select elements less than pivot*
    - e.g. `[12, 17, 9]`
  - *select elements greater than pivot*
    - e.g. `[]`
- we know how to sort an array of three elements
  - *may call quicksort recursively for this array*
- simply combine results to return sorted array
- we may now sort an array of four elements
- if we can sort an array of four elements
  - *may also sort an array of five elements*
  - *then six elements*
  - *& seven elements*
  - *…*

sorting - quicksort - part 5

- i.e. if we consider an array of five elements
  - *[6, 10, 4, 2, 8]*
- we may partition this array as follows
  - *then call quicksort for sub-arrays*
- e.g.

```
          []   2   [6, 10, 4, 8]
         [2]   4   [6, 10, 8]
      [2, 4]   6   [10, 8]
   [2, 6, 4]   8   [10]
[2, 6, 4, 8]  10   []
```

- clearly see how each sub-array has between zero and four elements
  - *already know how to sort arrays of these sizes using quicksort*
- regardless of chosen pivot
  - *recursively call quicksort on two sub-arrays*
- continue this logic for six elements, &c.

Algorithms and Data Structures

sorting - quicksort - part 6

- example implementation in Python is as follows

```python
def quicksort(data):
    if len(data) < 2:
        # base case - 0 or 1 elements already sorted...
        return data
    else:
        # recursive case
        pivot = data[0]
        # sub-array of elements less than pivot
        less_than = [i for i in data[1:] if i <= pivot]
        # sub-array of elements greater than pivot
        greater_than = [i for i in data[1:] if i > pivot]
        # return sorted data
        return quicksort(less_than) + [pivot] + quicksort(greater_than)

print(quicksort([6, 10, 4, 2, 8]))
```

Video - Algorithms and Data Structures

sorting algorithms

Algorithms and Sorting - UP TO 22:03

Source - Algorithms - YouTube

Algorithms and Data Structures

inductive proofs - part 1

- just seen an example of **inductive proofs**
- use such proofs to show an algorithm will work in theory
- each **inductive proof** has two familiar steps
  - *a base case*
  - *an inductive case*
- e.g. we want to prove that a test robot can climb steps
- **inductive case** may define the following
  - *if robot's legs are on a step*
  - *it may put its legs on the next step...*
    - e.g. if it's on the second step, it may now move to the third step, and so on...
- **base case** will define the following
  - *robots legs are on first step*
  - *it can now climb all of the steps*
    - i.e. progressing one step at a time...

inductive proofs - part 2

- we may see a similar logic for our earlier **quicksort** algorithm
- **base case** shown to work as expected for arrays of size `0` and `1`
- **inductive case** - proved that if quicksort worked with an array of size `1`
  - *it would also work with an array of size `2`*
- if it works for an array of size `2`
  - *it will also work for an array of size `3`...*
- by inductive reasoning
  - *the algorithm for **quicksort** will work with an array of any size*
- for real-world usage
  - *obviously making assumptions regarding memory usage, scale, &c.*
  - *but inductive proofs still remain true*

Algorithms and Data Structures

Big O revisited - part 1

- briefly return to a consideration of Big O notation
- comparison of runtimes for various *search* and *sort* algorithms
  - *may help provide some context for quicksort &c...*
  - *e.g.*

binary search	simple search	quicksort	selection sort	traveling salesman
O(log n)	O(n)	O(n log n)	O(n²)	O(n!)
logarithmic	linear	linearithmic	quadratic	factorial

Video - Algorithms and Data Structures

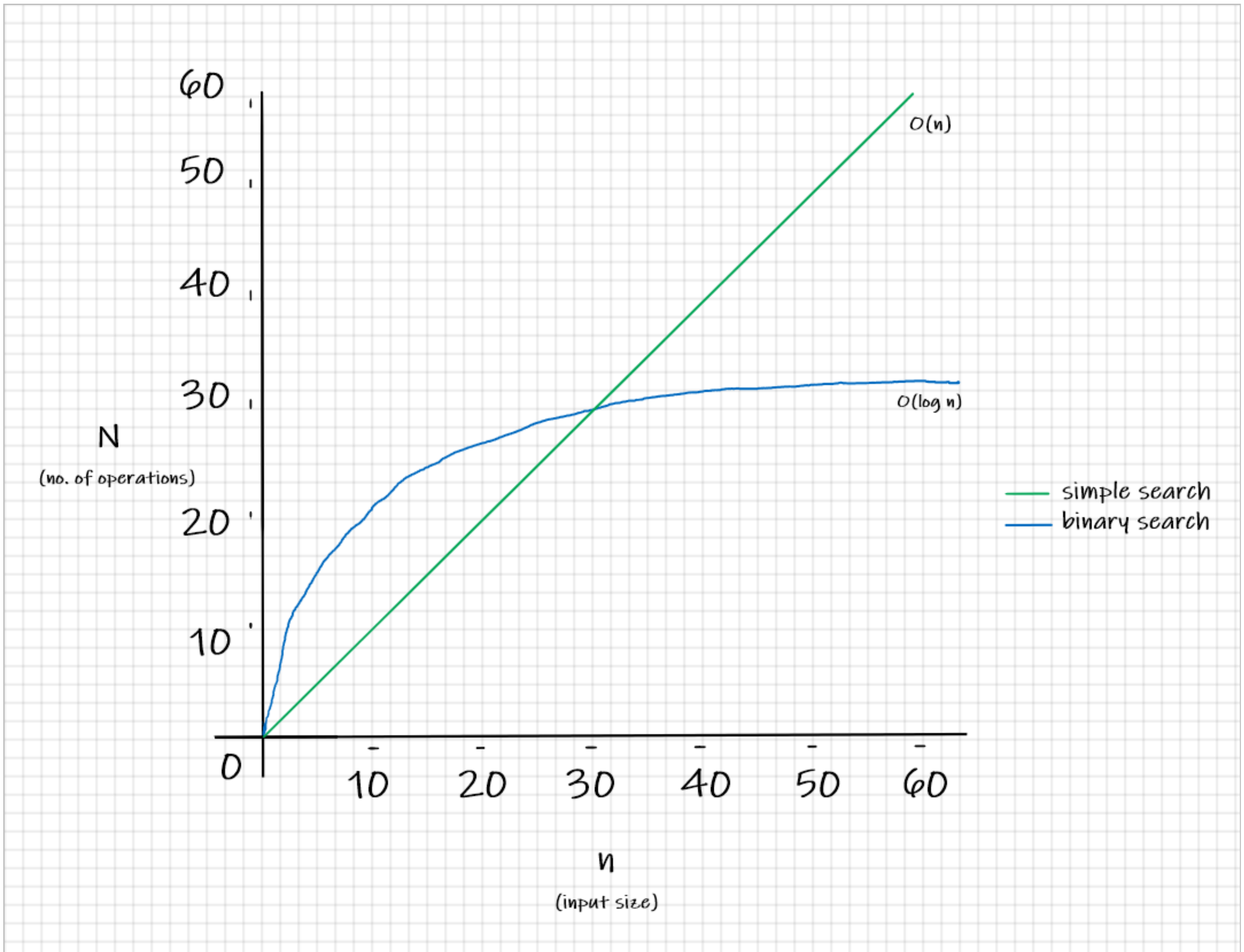
quicksort - part 2

Quicksort - UP TO 4:40

Source - Quicksort - Java - YouTube

Image - Big O revisited

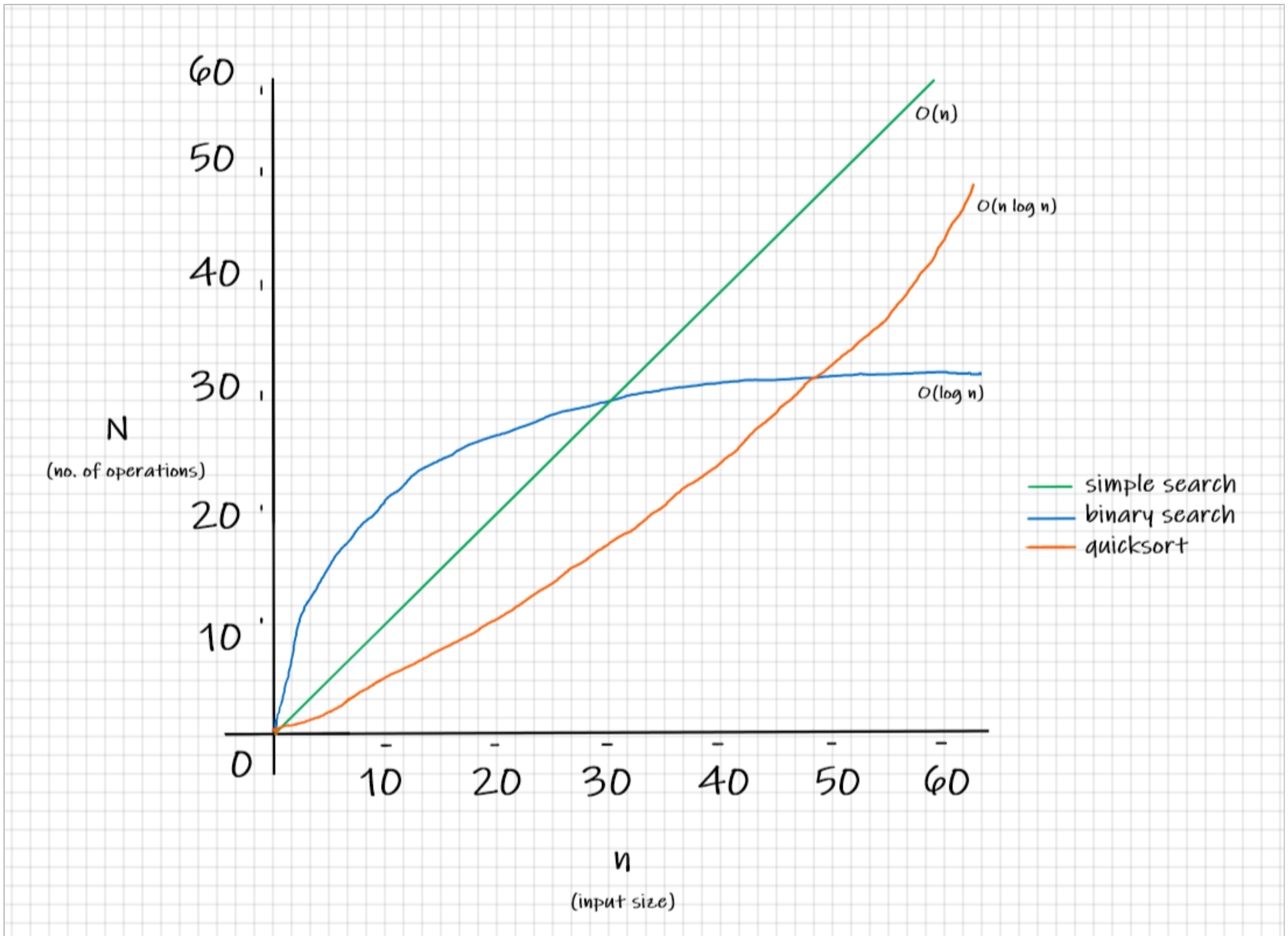
simple search and binary search



Big O Complexity - Simple Search and Binary Search

Image - Big O revisited

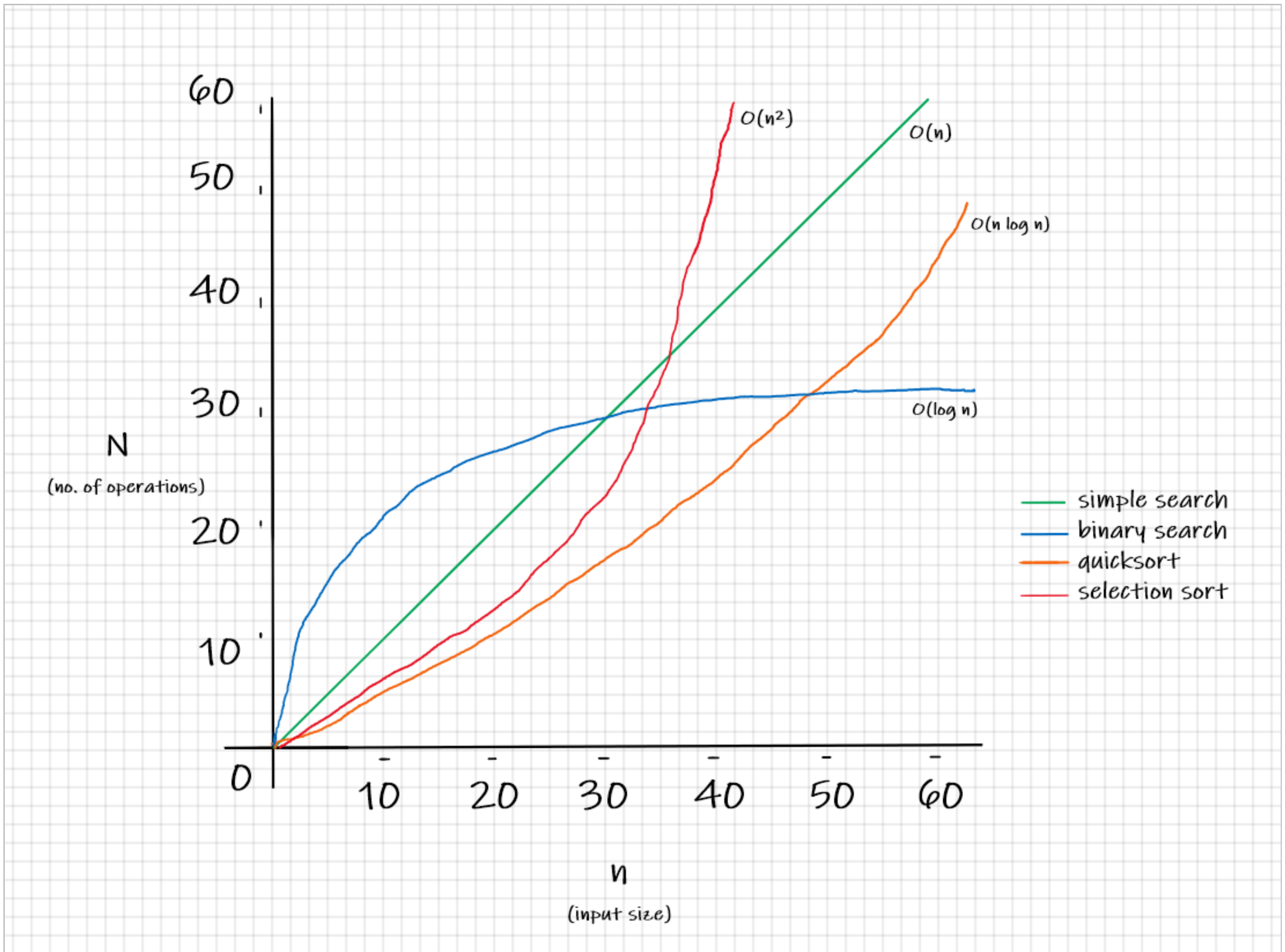
simple search, binary search, and quicksort



Big O Complexity - Simple Search, Binary Search, and Quicksort

Image - Big O revisited

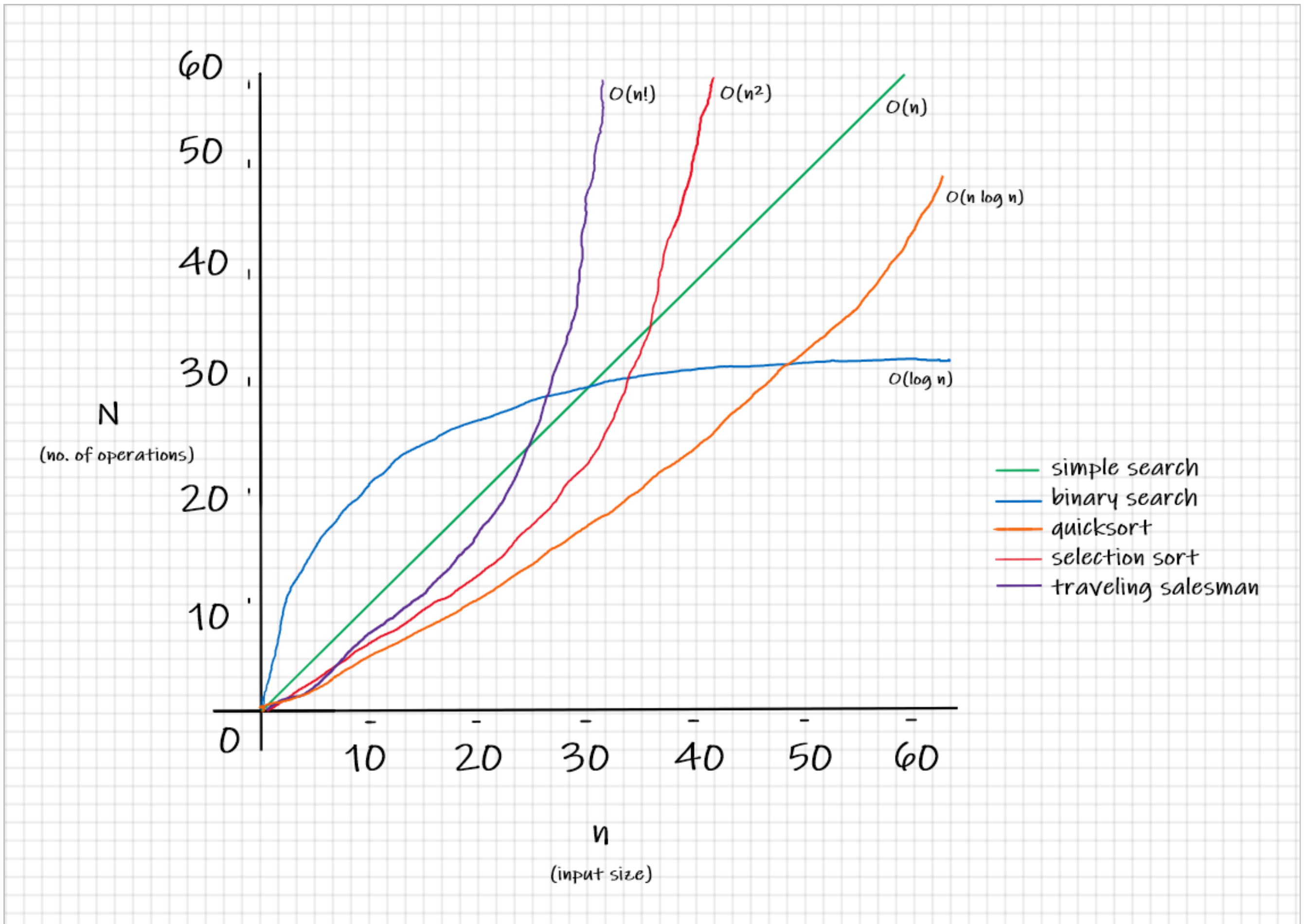
simple search, binary search, quicksort, and selection sort



Big O Complexity - Simple Search, Binary Search, Quicksort, and Selection Sort

Image - Big O revisited

simple search, binary search, quicksort, selection sort, and traveling salesman



[Big O Complexity - Simple Search, Binary Search, Quicksort, Selection Sort, and Traveling Salesman]()

Big O revisited - part 2

- consider the following comparative run times
  - *computer capable of a basic 10 operations per second*
  - *(such a slow computer helps visualise the comparative performance)*

data size	quicksort	selection sort	traveling salesman
10 items	3.3 seconds	10 seconds	4.2 days
100 items	66.4 seconds	16.6 minutes	2.9×10^{149} years
1000 items	996 seconds	27.7 hours	1.27×10^{2559} years

- previous graphs indicative of expected performance
  - *not accurate reflections of performance times*
- show difference in expected performance for each algorithm
  - *relative to scale...*
- e.g quickly see that *selection sort*, `O(n²)`, is a slow algorithm
  - *in particular compared with quicksort...*

Big O revisited - part 3

- compare with another sorting algorithm
  - *e.g merge sort - a time of* $O(n \log n)$
  - *much faster than selection sort*
- current algorithm *quicksort* is a tad harder to pin down
- for worst case
  - *time is* $O(n^2)$
  - *potentially as slow as selection sort*
- for average case
  - *define a time of* $O(n \log n)$
  - *comparable with faster algorithm merge sort*
- if *merge sort* is considered faster with a time of $O(n \log n)$
  - *why not use this algorithm all the time instead of quicksort?*

Algorithms and Data Structures

Big O revisited - part 4

- consider a comparison of *quicksort* and *merge sort*
  - *should helps choose a preferred algorithm to use...*
- start with the following simple usage
  - *a Python function to iterate a list*

```
def print_list(data):
  for val in list:
    print val
```

- as this iteration loops the whole list
  - *runs with a time of* `O(n)`
- what happens if we need to introduce a pause per iteration...
  - *e.g. perhaps to check an external data store, API &c.*
  - *add a test pause of one second per iteration*
- both use cases need to loop through data
  - *each may be defined with a time of* `O(n)`
- even though both functions return same time using Big O notation
  - *first iteration without pause will return faster real-world performance and time...*

Big O revisited - part 5

- consider apparent contradiction for a moment
  - *start to understand actual meaning of Big O notation*
- consider a time of `O(n)` as follows

```
`constant` x `n`
```

- or

```
`c` x `n`
```

- `c` is a fixed amount of time algorithm will take
  - *or the constant*
- comparative times for basic iteration and iteration with a pause
  - *e.g.* `10ms * n` *vs* `1 sec * n`

Big O revisited - part 6

- usually ignore such constants
  - *if comparative algorithms have different Big O time*
  - *i.e. for most instances - constant doesn't matter*
- e.g. compare *simple search* to *binary search* for previous usage

```
simple search = 10ms * n
binary search = 1 sec * log n
```

- simple search initially seems faster
- if we scale query to **four billion** elements
  - *disparity in performance becomes clear...*

```
simple search = 10ms * 4 billion = 463 days
binary search = 1sec * 32 = 32 seconds
```

- clear improvement in times with binary search
  - *the **constant** did not make a difference*

Big O revisited - part 7

- still exceptions to this rule
- i.e. constant may sometimes make a difference
- *Quicksort* versus *merge sort* is one example where this holds true
- *Quicksort* has a smaller constant than *merge sort*
- if they're both `O(n log n)` time
  - *quicksort is faster...*
- *quicksort* is faster in practice
  - *it hits average case more frequently than worst case*

Big O revisited - part 8

- *average case* and *worst case*
- performance for *quicksort* predicated on chosen *pivot*
- e.g. if we choose a pivot and array is already sorted
  - *quicksort does not check if input array is already sorted*
  - *i.e. it will try to sort the passed array*
- if we compared two possible scenarios for an array
  - *1. first element is always chosen as the pivot*
  - *2. middle element is always chosen as the pivot*
- starting at middle element
  - *will not need to make as many recursive calls for this example*
  - *i.e. hits the base case more quickly, and required call stack will also be shorter...*

Big O revisited - part 9

- first example, choosing first element, is *worst case*
- second example, middle element selection, is *best case*
- for *worst case*
  - *stack size is* `O(n)`
- *best case* has a stack size of `O(log n)`
- e.g. we may see how best case is partitioning the array

```
[1, 2, 3, 4, 5, 6, 7, 8]

[1, 2, 3] 4 [5, 6, 7, 8]
[1] 2 [3]   [5] 6 [7, 8]
                [] 7 [8]
```

Big O revisited - part 10

- for *worst case*
  - *checking each element in array*
  - *e.g. eight in this example*
- first operation takes `O(n)`
  - *we actually check* `O(n)` *elements on every level of call stack*
- even if we partition array in a different manner
  - *e.g. with a different pivot*
- still checking `O(n)` elements every time
- i.e. each level of the stack currently takes `O(n)` time to complete

Big O revisited - part 11

- difference between worst case and best case
  - *seen when we consider height of call stack*
- e.g. best case will check `O(log n)` levels
  - *height of its call stack*
- each level takes `O(n)` time
  - *algorithm will take `O(n) * O(log n)`*
  - *i.e. `O(n log n)` time*
  - *best case for this algorithm*
- see difference when we calculate comparative worst case
  - *a time of `O(n)` for each level*
  - *but also `O(n)` levels*
  - *algorithm will take `O(n) * O(n)`*
  - *i.e. $O(n^2)$ time*
- also define best case as average case
  - *if we always choose a random element in array as defined pivot*
  - *quicksort algorithm will have average time of `O(n log n)`*

Resources

- Algorithms - YouTube
- Divide and Conquer - YouTube
- Memoisation - YouTube
- Quicksort - Java - YouTube
- Recursion & Fibonacci - YouTube
- Recursion and Fun - JavaScript - YouTube
- Recursion and the Call Stack - Java - YouTube