Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 6

Dr Nick Hayward

recursion and the call stack - part 7

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

```python
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))
```

Algorithms and Data Structures

recursion and the call stack - part 8

- pseudocode outline for pattern of execution for tail recursion
- e.g. `3!` - factorial(3)

```
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
```

Video - Algorithms and Data Structures

Recursion, the Call Stack, and Overflow...part II

!!Con 2019- Tail Call Optimization: The Musical!! - UP TO 7:58

Source - !!Con 2019- Tail Call Optimization: The Musical!! - YouTube

Algorithms and Data Structures

recursion and the call stack - part 9

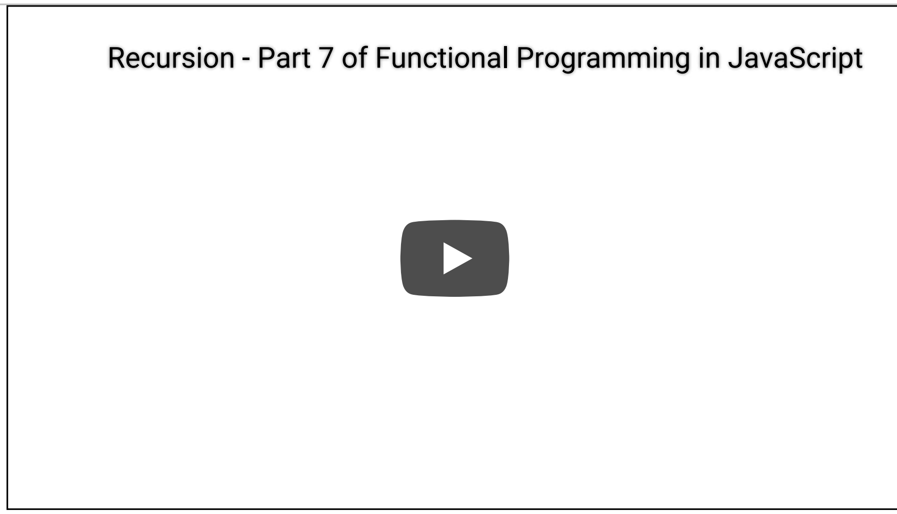
- stack may be used to represent execution logic
  - *e.g. for a recursive function in JavaScript*
- code example uses a *call stack* to ensure expected execution

```
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));
```

recursion and the call stack - part 10

- initial `findSolution()` function is called
  - *passed parameter of* `24` *is the value to check*
- function returns an executed `find()` function
  - *initial test values for current and history*
- part of this function's execution
  - *checks initial values until it reaches* `else` *part of conditional statement*
- returns `find()` function
  - *called recursively*
  - *initially checking against addition of 5*
  - *continues to check possible values with `+ 5`*
  - *either succeeds or moves onto right side of logical OR,* `||`*,* logical OR checks with `* 3`*
- it will either succeed or fail with these recursive checks
- structure that permits this recursion to execute
  - *structure is the call stack*
- *call stack* provides a defined pattern to execution
  - *pattern allows the code to run as expected*

Recursion for Fun - part 2



Recursion and Fun - JavaScript - UP TO 14:46

Source - Recursion and Fun - JavaScript - YouTube

stack operations - part 1

- we may define a stack as a simple list of elements
  - *may be accessed from only one end*
  - *known as the top of the stack*
- i.e. refer to data structure as *last in, first out*
- known limitation of this structure
  - *lack of access to elements not at top of stack*
- a simple difference between structures
  - *i.e. basic list or array and specific stack*
- to access the bottom element
  - *all elements above must first be popped*

stack operations - part 2

- stack operations are simple, e.g.
  - *add elements to the top*
  - *pop elements from the top*
- also means specific restrictions must be in place
  - *ensures only these operations are allowed*
  - *i.e to define usage for the data structure*
  - *if not, it ceases to be a stack*

stack operations - part 3

- complementary operations are commonly available
  - *may vary relative to language implementation*
- e.g. a stack may permit the following
  - *view the top element in the stack*
  - *this is not pop - element is not removed from stack*
  - *operation known as peeking*
  - *clear operation will remove all elements from stack*
  - `length` *property returns number of elements in stack*
  - `empty` *returns whether stack has any values or not*

Video - Algorithms and Data Structures

Recursion, the Call Stack, and Overflow...part III

!!Con 2019- Tail Call Optimization: The Musical!! - UP TO END

Source - !!Con 2019- Tail Call Optimization: The Musical!! - YouTube *example implementations*

- choose various existing data structures to define custom stack
- e.g. might use an array or list
- choice will often depend on support in chosen programming language
- e.g. in JS - common option is an `array` object
- define constructor for `stack` object
  - *then extend prototype for custom properties and methods*

stack constructor

- initial constructor is as follows

```
// CONSTRUCTOR = Stack object
function Stack() {
    /* define instance properties for stack
    * - empty array for instantiated stack
    * - options might include max length, restricted data type &c.
    */
    this.store = [];
}
```

- instantiate a basic Stack object
  - *simply defining an empty array*
  - *use array as store for Stack data structure*
- constructor may be updated to include
  - *type checks and restrictions*
  - *initial values for Stack*
  - *required access context*
  - *...*

extend the prototype

- initially extend Prototype for this object
- add required functionality for a basic stack
- e.g. define functions for the following
  - *add data*
  - *delete data*
  - *get size of Stack*

prototype - add data

- *add data* function
  - *add passed data to top of stack*

```
// PROTOTYPE - add method for value pushed to top of stack
Stack.prototype.add = function (value) {
    this.store.push(value);
    console.log(`value added = ${value}`);
}
```

- underlying `store` object is an array for Stack
  - *use default `push()` method to add required data*

prototype - delete data

- *delete data* function defined as follows

```
Stack.prototype.delete = function () {
    const deletedValue = this.store.pop();
    console.log(`last value deleted = ${deletedValue}`);
}
```

- same as *add data*
  - *use default `pop()` method for `store` array*
  - *added to custom Stack data structure*

prototype - size of Stack

- also define `size()` function for Stack
  - *use built-in Array property for `length`*

```
Stack.prototype.size = function () {
    const size = this.store.length;
    console.log(`store size = ${size}`);
}
```

prototype - peek Stack

- useful option is *peeking* at the top of Stack
- e.g.

```
Stack.prototype.peek = function () {
    const peekValue = this.store[(this.store.length-1)]
    console.log(`top value = ${peekValue}`);
}
```

- function will return copy of top value
  - *will not delete item from Stack and underlying `store` array*

prototype - clear stack

- common operation for a Stack is to clear all entries,
  - *yet preserve the Stack itself*
- i.e. resetting store array for instantiated Stack object
- e.g.

```
Stack.prototype.clear = function () {
    // resets Stack's array store - clears all items
    this.store = [];
}
```

prototype - check empty stack - part 1

- check an instantiated Stack object for entries
  - *i.e. determine if stack is empty or not*

```
Stack.prototype.empty = function () {
    if (this.store.length === 0) {
        return true;
    } else {
        return false;
    }
}
```

prototype - check empty stack - part 2

- conditional logic has been placed in this function
  - *i.e. not passed down chain of logic to requesting application call*
  - *means function is self-contained*
- function returns valid response regardless of execution context
- as we develop Stack’s Prototype methods
  - *add further restrictions and controls*
  - *clearly defines how to use this data structure*
- also define what and how may be returned
  - *custom data structure customised to context, usage…*

Prototype in JavaScript - part 1

Prototype in JavaScript - UP TO 1:00

Source - Prototypes in JavaScript - YouTube

Video - Algorithms and Data Structures

Prototype in JavaScript - part 2

Prototype in JavaScript - UP TO 6:41

Source - Prototypes in JavaScript - YouTube

control access to the stack - part 1

- Stack object and methods
  - *now working as expected*
- Stack is still open to mis-use
  - *due to array object in the Stack*
- restrict and control access to this Stack data structure
  - *e.g. using a Proxy*

control access to the stack - part 2

- to use a Proxy with our Stack constructor
  - *define a custom `construct` trap*
- may also use Reflect API to define defaults for handlers
- `construct` trap intercepts calls
  - *i.e. to defined `new` operator for a given constructor*

control access to the stack - part 3

- define initial Proxy wrapper for passed constructor

```
/*
* PROXY
*/
function proxyConstruct(constructor) {

    const handler = {
        construct(constructor, args) {
            console.log('proxy constructor...');
            // const stack = Reflect.construct(constructor, args);
            return new constructor(...args);
        }
    };
    return new Proxy(constructor, handler);
}
```

control access to the stack - part 4

- then pass basic `Stack` constructor to the proxy

```
// proxy wrapper for Stack constructor
const proxiedStack = new proxyConstruct(Stack);
// instantiate proxied Stack & check store...
console.log(new proxiedStack().store);
```

control access to the stack - part 5

- instantiation of a proxied Stack object
  - *allows us to wrap constructor for Stack*
- may still use prototype methods for instantiated Stack object
- benefit of using a proxy for the constructor
  - *control of initial object instantiation*
- i.e. if object cannot be instantiated
  - *access to Prototype methods becomes irrelevant*

Video - Algorithms and Data Structures

Proxy in JavaScript

Using a JavaScript Proxy - UP TO 3:28

Source - Proxy in JavaScript - YouTube

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));
```

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
```

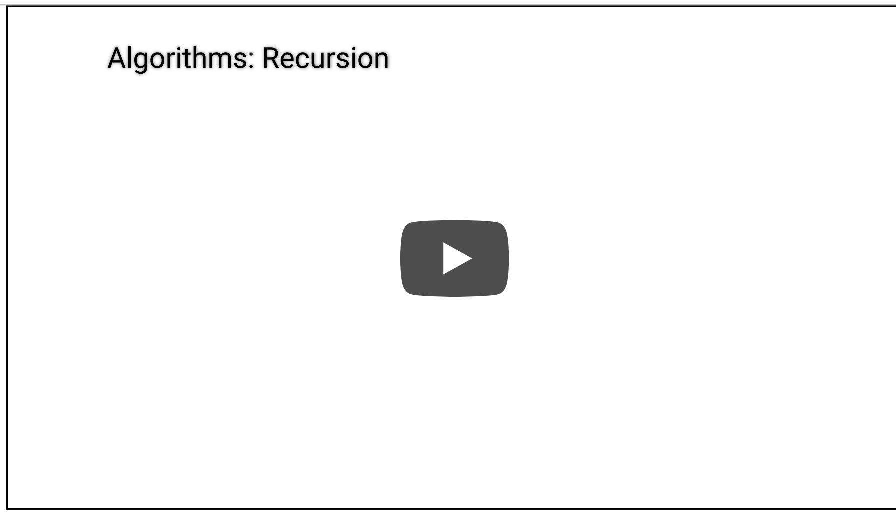
```
        fib(3-2)
            n = 1
            return 1 // base returned

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

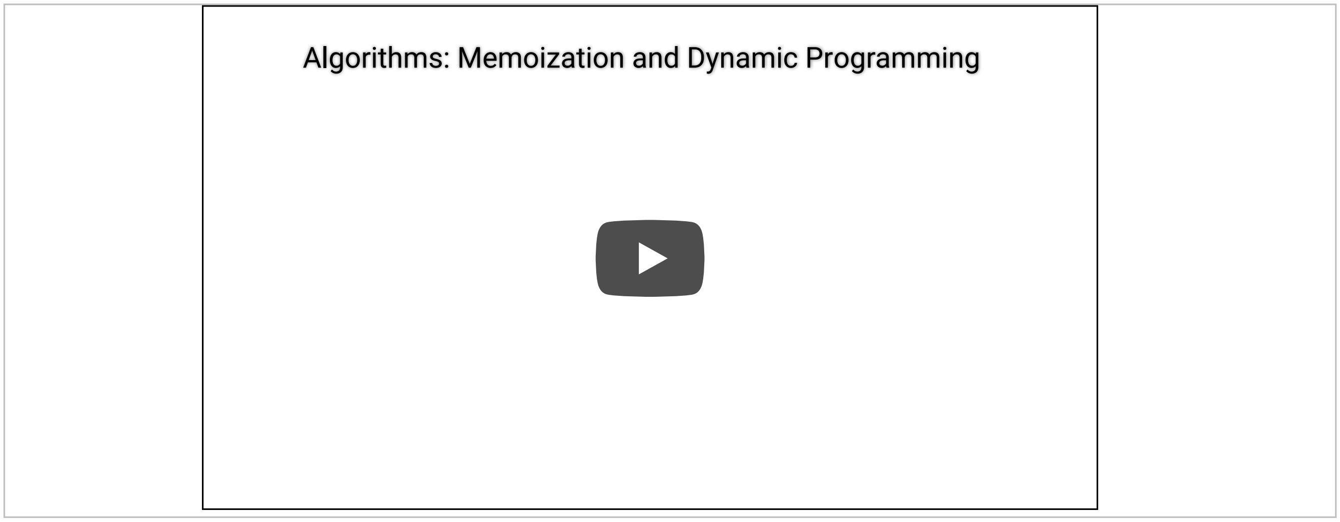
- why does this JavaScript recursive solution actually work as expected?
- as function is called recursively
  - *only returns value for base case*
  - *i.e. either 0 or 1*
- as it continues down from value of passed `n-th` position in series,
  - *it is storing each return*
  - *then returns total for 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

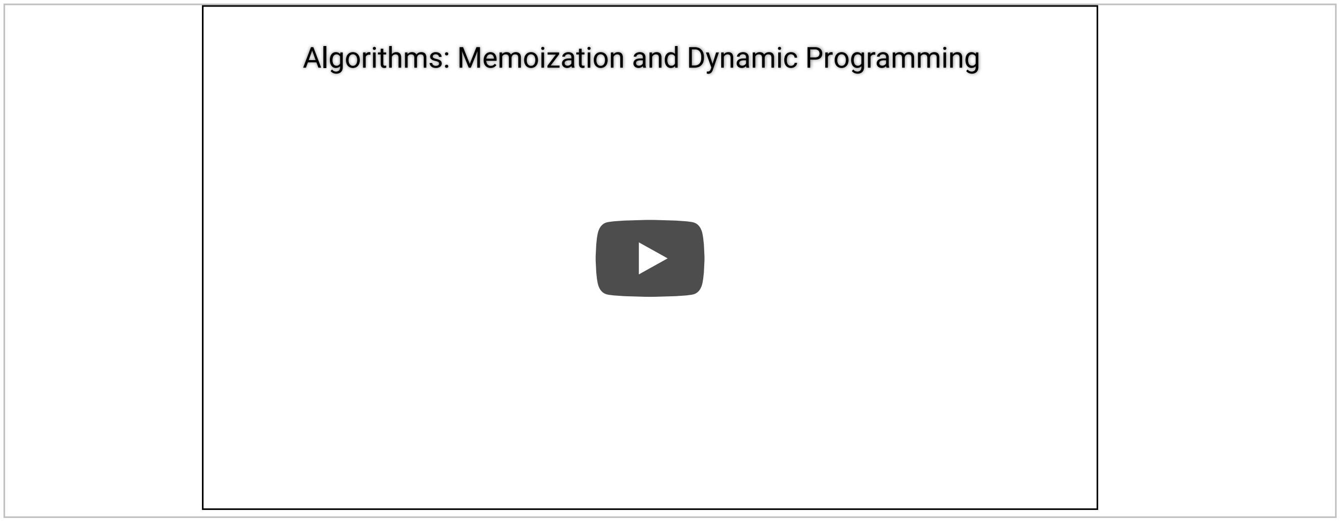
- store arguments of a given function call along with computed result
- e.g. when `fib(4)` is first called
  - *computed value will be stored in memory*
  - *a temporary cache in effect*
- then call stored return
  - *e.g. each and every subsequent call to `fib(4)`*

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

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*

Algorithms and Data Structures

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...*

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`

Algorithms and Data Structures

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

```python
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])}`);
```

Resources

JavaScript

- MDN - Array
- MDN - Prototype
- MDN - Proxy
- Prototypes in JavaScript - YouTube
- Proxy in JavaScript - YouTube

Python

- Stacks - YouTube

Various

- !!Con 2019- Tail Call Optimization: The Musical!! - YouTube
- Algorithms - YouTube
- Divide and Conquer - YouTube
- Event Driven Architecture - YouTube
- Memoisation - YouTube
- Memory Manager - YouTube
- Recursion & Fibonacci - YouTube
- Recursion and Fun - JavaScript - YouTube
- Recursion and the Call Stack - Java - YouTube