Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 4

Dr Nick Hayward

Project outline & mockup assessment

Course total = 15%

- begin outline and design of an application
  - *built from scratch - languages include*
    - JavaScript
    - Python
    - C
    - …
  - *builds upon examples, technology outlined during first part of semester*
  - *must implement algorithms & data structures*
  - *purpose, scope &c. is group's choice*
  - *NO blogs, to-do lists, note-taking…*
    - chosen topic requires approval
  - *presentation should include mockup designs and concepts*

Project mockup demo

Assessment will include the following:

- brief presentation or demonstration of current project work
  - *~ 5 to 10 minutes per group*
  - *analysis of work conducted so far*
  - *presentation and demonstration*
    - outline current state of app concept and design
    - show prototypes and designs
  - *due Tuesday 11th February 2020 @ 4.15pm*

Fun Exercise

pseudocode game

Consider the following Snake game,

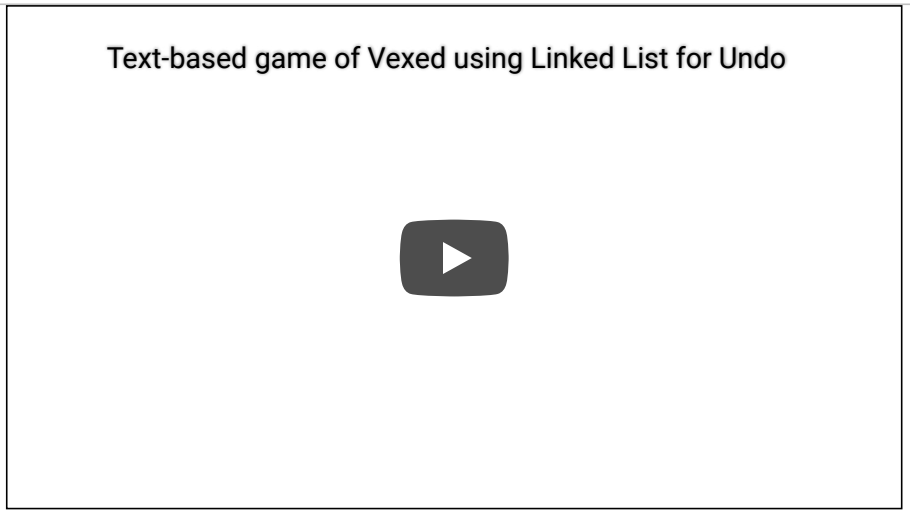
Then, using **pseudocode**

- define logic for this game
  - *use linked list*
- how will the following be used in this game
  - *accessors/selectors*
  - *mutators*

Approx. 10 minutes...

Video - Algorithms and Data Structures

Linked list in games



Text based game of Vexed

Source - Text based game with linked list - YouTube

System and Memory

app to memory

- as we design an appropriate algorithm,
  - *need to consider how an OS and application handle and use memory*
- e.g. an OS's management of memory
  - *closely associated with process requirements and usage*
- memory management relative to a process (e.g. application) may be considered broadly as follows
  - *ensure each process has enough memory to execute*
    - cannot run out of memory or use other processes' memory allocation
  - *different memory types must be organised efficiently*
    - ensures effective management of each process
- we may start by managing memory boundaries for different processes

System and Memory

process and memory usage

- if we consider restrictions and limitations of array implementation and management
  - *need a way to effectively manage this use of memory*
- as a child process is created, it is assigned an address memory space
- each process will see their memory space as a contiguous logical unit
- such memory addresses might actually be separate across the system
- disparate addressed memory spaces for each process
  - *may then be organised together, as needed, by the system's kernel*
- e.g. separate memory stores and addresses organised into a contiguous group per process
- benefit is efficient use of memory space
- no need for pre-assigned large chunks of memory
  - *or reserved memory that is never used by a process*
- kernel controls access for a process to memory addresses
  - *kernel is controlling conversion of assigned virtual addresses*
  - *converts to a physical address in the system's hardware memory*

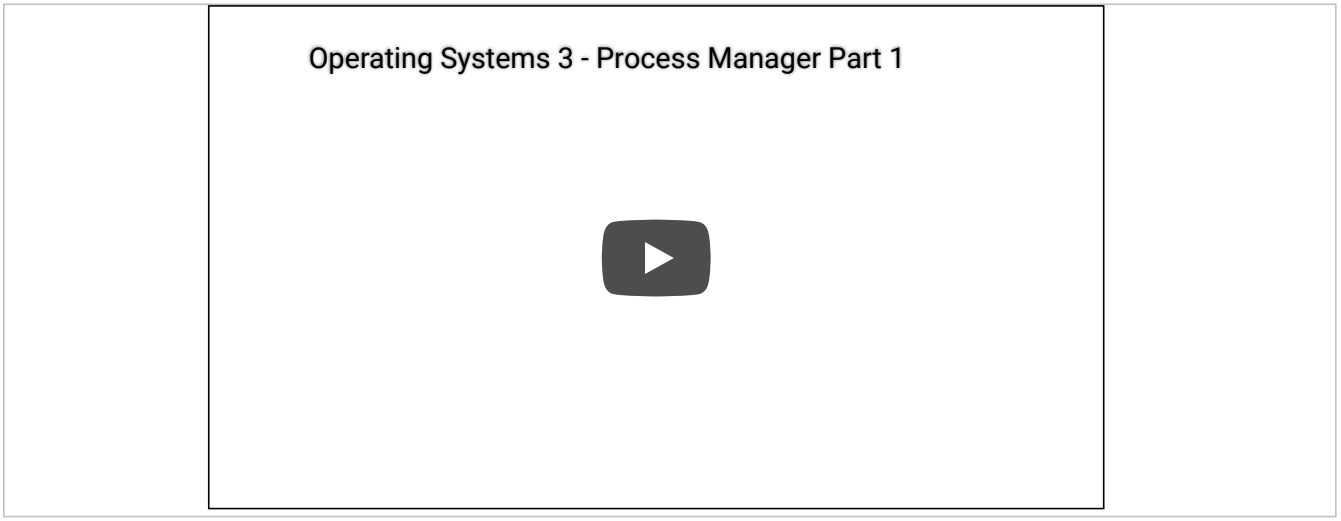
System and Memory

process and state

- each child process may have a related *state*
  - *associated during the lifetime of the process itself*
- state may be monitored by the system's *kernel*
  - *a process will wait until resources are available to allow a change in state*
- kernel may then switch processes relative to an update in a process' state

Video - System and Memory

process manager - part 1



Operating Systems - Process Manager - UPTO 1.41

Source - Process Manager - YouTube

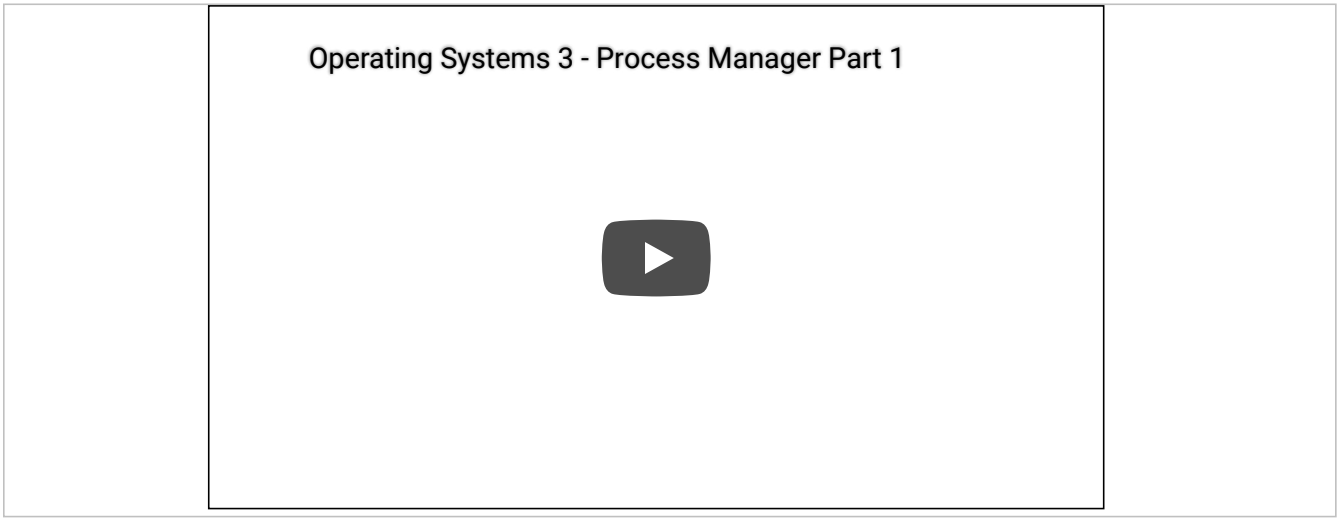
System and Memory

process manager

- *process manager* is responsible for processes in a system
- it is controlled by the system's *kernel*, and manages the following
  - *process creation and termination*
  - *resource allocation and protection*
  - *cooperation with device manager to implement I/O*
  - *implementation of address space*
  - *process scheduling*

Video - System and Memory

process manager - part 2



Operating Systems - Process Manager - Scheduler - UPTO END

Source - Process Manager - Scheduler - YouTube

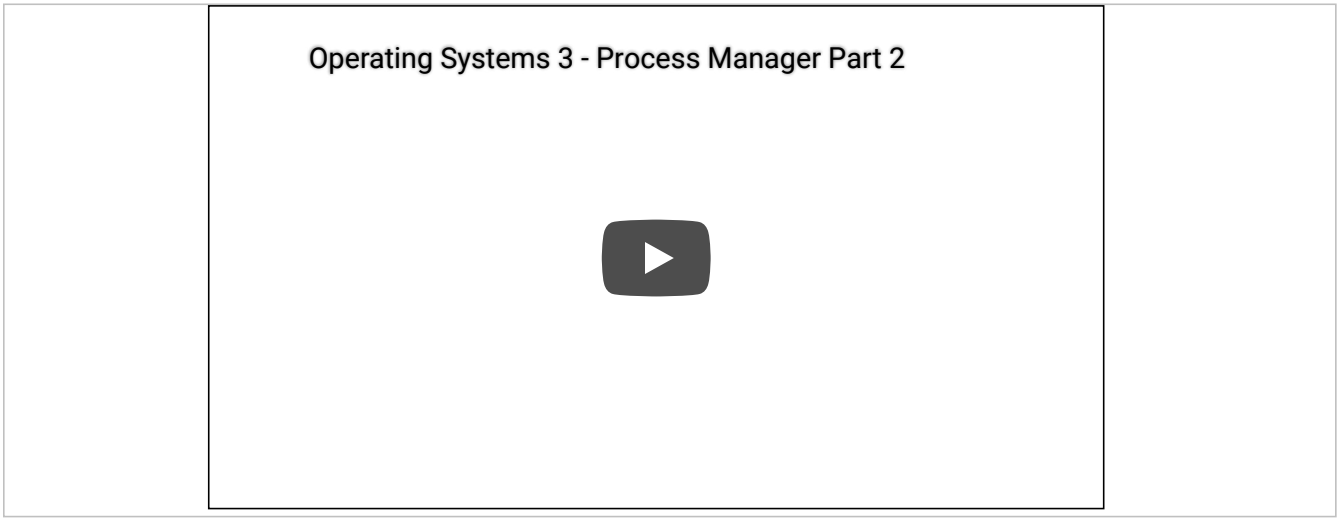
System and Memory

process scheduling

- a key part of managing processes in a system is efficient scheduling
- scheduling is part of the *process manager*
  - *actually maintained by the system's kernel*
- kernel is responsible for switching between processes
  - *checking and migrating available ready state processes to execution in an active state*
- *kernel* is selecting processes to execute in the system on the available CPU
  - *kernel is choosing the next process to run on the CPU*
- context switch is informed by the required and available *process properties*
- selection of process is also determined by the nature of the process itself
  - *i.e. is it I/O bound or CPU bound*
- algorithm helps determine the best process choice
  - *ensures system runs efficiently and without apparent delays*
- example algorithms include,
  - *first-come, first-served*
  - *shortest job next*
  - *round robin*
  - *multi-level priority queue*
- scheduling is meant to provide a fast and efficient system
- kernel chooses processes to allow the system to run fast
- e.g. it is common to assign priority to a *front-facing* process over one running in the *background*

Video - System and Memory

process manager - part 3



Operating Systems - Process Manager - Algorithms and Management

Source - Process Manager - Algorithms and Management - YouTube

Practical usage - JavaScript Arrays

intro

- collections in JS includes arrays
  - *associated built-in array methods, plus ES6 updates for sets and maps &c.*
- arrays in JS are simply objects
- as objects, arrays can access methods…

Practical usage - JavaScript Arrays

create an array

- two fundamental ways to create new arrays:
  - *using the built-in Array constructor*
  - *using array literals [ ]*
- e.g.

```
const readers = ["emma", "yvaine", "daisy"];
const archives = new Array("waldzell", "mariafels");
```

- array literals tend to be the more common option for JS development
- n.b. Writing to indexes outside the array bounds extends the array
- e.g. `readers.length === 5`
- if we try to write to a position outside of array bounds, as in

```
readers[4] = "bea";
```

- array will expand to accommodate the new situation
- may end up creating a hole in the array
  - *the item at index 3 will be* `undefined`
  - *length property will also be updated*

Practical usage - JavaScript Arrays

adding and removing items at either end of an array

- a few simple methods we can use to add items to and remove items from an array:
  - *push - adds an item to the end of the array*
  - *unshift - adds an item to the beginning of the array (existing items are moved forward one index posn)*
  - *pop - removes an item from the end of the array*
  - *shift - removes an item from the beginning of the array (existing items are moved back one index posn)*
- n.b. push and pop are faster than shift and unshift due mods of the index...

Practical usage - JavaScript Arrays

adding and removing items at any array location

- if we simply delete an array item
  - *we leave a hole at that index position with `undefined`…*
  - *array length will still include this hole…*
- instead we need to use the `splice` method for insertion and deletion
- e.g.

```
var removedItems = readers.splice(1, 1);
```

- this removes a single item at index posn 1
- `splice` method will also return its own array of deleted items.
- using `splice` method
  - *also insert items into arbitrary positions in an array*
- e.g. consider the following code:

```
removedItems = readers.splice(1, 2, "cat", "rose", "violet");
//readers: ["daisy", "cat", "rose", "violet"]
//removedItems: ["emma", "yvaine"]
```

- starting from index 1
  - *it first removes two items*
  - *then adds three items: “Mochizuki”, “Yoshi”, and “Momochi”*
  - *algorithm defined and working…*

Practical usage - JavaScript Arrays

common operations on arrays

- some common operations on JS arrays include,
  - *iterate - traverse arrays*
  - *map - map existing array items to create a new array based on these items*
  - *test - check array items match certain conditions*
  - *find - find specific array items*
  - *aggregate - compute a single value based on array items, e.g. compute total for array from array items...*

Practical usage - JavaScript Arrays

common operations on arrays - iterate with `forEach`

- all JS arrays have a built-in method for `forEach` loops
- e.g.

```
const archives = ['waldzell', 'mariafels'];

archives.forEach(archive => {
    console.log(`archive name = ${archive}`);
});
```

Practical usage - JavaScript Arrays

common operations on arrays - map arrays

- with array mapping
  - *creating a new array based on the items in an existing array*
  - *become common usage in JavaScript development*
- idea is simple
  - *we map each item from one array to a new item in a new array*
  - *we might extract just names from an array of archives*
- e.g.

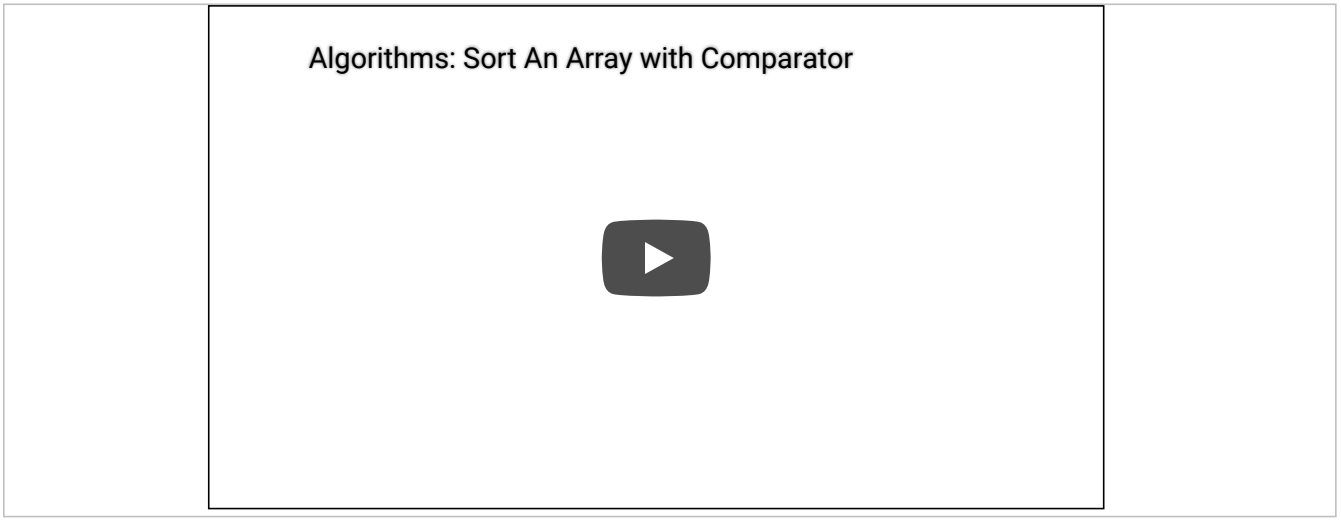
```
// array
const archives = [
  {name: 'waldzell', type: 'game'},
  {name: 'mariafels', type: 'benedictine'}
];

// map array items to new array
const archiveNames = archives.map(archive => archive.name);

// iterate through new array
archiveNames.forEach(archive => {
  console.log(`archive name = ${archive}`);
});
```

Video - Fun example

Java - comparator array sort



Array Usage - Sort an Array with Comparator - Java

Source - Sort an Array with Comparator - YouTube

Practical usage - JavaScript Arrays

common operations on arrays - test array items

- check one or more array items to see if they match certain conditions
- help with this requirement, JS provides some useful built-in functions, `every` and `some`
  - *`every` method - pass a callback, which is called for each specified property in the array*
    - e.g. check if all properties have a specified value &c.
    - returns a boolean for the check - `true` for *all* properties matching specified value, otherwise `false`
  - *`some` method - pass a callback, which is called for each specified property in the array*
    - e.g. check at least one property matches a specified value
    - returns a boolean - `true` for at least one match, `false` for zero matches
  - *e.g.*

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine'}
];

// check archives - `every` returns true for all match, `false` for a single error/omission
const everyName = archives.every(archive => 'name' in archive);
// check boolean return for `every` method in everyName
everyName === true ? console.log(`each archive has a name`) : console.log(`at least one
         archive is unnamed...`);

// check archives - `some` return true for a single match, `false` for no matches
const singleLocation = archives.some(archive => 'location' in archive);
// check boolean return value
singleLocation === true ? console.log(`at least one archive has a location`) :
         console.log(`no archive has a location...`);
```

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 1

- also search and find items in JS arrays
- JS provides another built-in function, `find`
- e.g.

```
// array
const archives = [
  {name: 'waldzell', type: 'game', location: 'castalia'},
  {name: 'mariafels', type: 'benedictine', location: 'czech'}
];

// find object in array
const locations = archives.find(archive => {
  // return object - not found simply returns undefined
  return archive.location === 'castalia';
});

// check search - check undefined or log archive name to console
locations !== undefined ? console.log(`archive in castalia = ${locations.name}`) :
         console.log(`location and archive not found...`);
```

- if the requested item can be found
  - *matching object will be returned*
- otherwise, the `find` method will simply return `undefined`

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 2

- `find` will return first matching item
  - *regardless of the number of matches*
- to search an array for all matches we can use the `filter` method instead
- e.g.

```
// filter array and return multiple matches
const filterTypes = archives.filter(archive => 'type' in archive);
```

- returns all matching items
  - *simply check length of return object,*
  - *and iterate through the results*
- e.g.

```
// check filter returns
if (filterTypes.length >= 1 ) {
  for(let archive of filterTypes) {
    console.log(`archive name = ${archive.name} and type = ${archive.type}`);
  }
} else {
  console.log(`archive types are not available...`);
}
```

Practical usage - JavaScript Arrays

common operations on arrays - searching arrays - part 3

- also possible to filter an array by index using following methods
  - *`indexOf` = find the index of a given item*
  - *e.g.*

```
const waldzellIndex = archives.indexOf('waldzell');
```

- `lastIndexOf` = find last index of multiple matched items
- e.g.

```
const waldzellIndex = archives.lastIndexOf('waldzell');
```

- `findIndex` = effectively works the same as `find` but returns an index value
- e.g.

```
const waldzellIndex = archives.findIndex(archive => archive === 'waldzell');
```

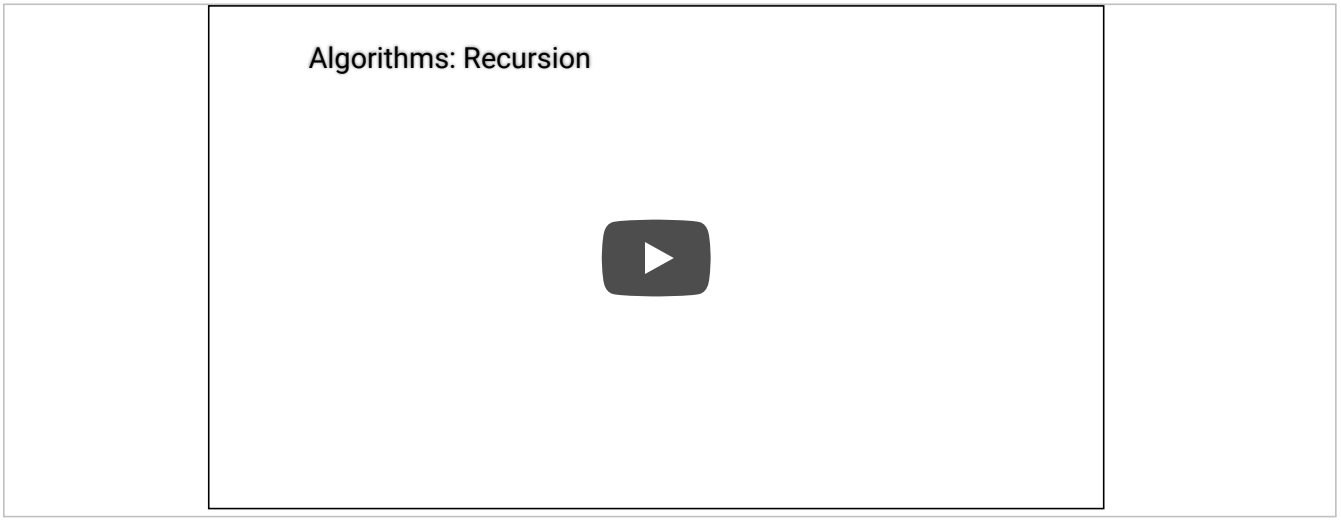
Algorithms and Data Structures

recursion and patterns

- as seen with custom linked list data structure
  - *access and iteration is a key consideration*
  - *e.g. general use, effective re-use…*
- e.g. no existing index for each item in the linked list
  - *choose to use a pattern such as recursion to check and access the list*
- recursion is a common technique used in design of many algorithms
  - *and app development in general*
- key benefit of recursion is option to define a *base* case and *recursive* case
  - *to help solve a given problem*
- recursion commonly provides an elegant way to solve complex problems
- its usage may also be seen as somewhat divisive
  - *i.e. controversial depending upon context*

Video - Algorithms and Data Structures

Recursion



Recursion - UP TO 2:27

Source - Recursion - YouTube

Algorithms and Data Structures

recursion for fun - part 1

- consider a jar of *10,000* sweets with various colours
  - *only a single winning gold sweet*
- we might design an algorithm with recursion
- initially two procedures we may define
  - *pick a sweet*
  - *check the sweet's colour*
- second procedure may also be used to check sweet's colour
  - *i.e. does it match prize gold sweet*
- defines whether we need a recursive call to first procedure
  - *or process has finished*

Algorithms and Data Structures

recursion for fun - part 2

- outline required steps to achieve overall process of finding gold sweet
- procedures will include various tasks to help resolve overall process
- e.g.

```
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...
```

Algorithms and Data Structures

recursion for fun - part 3

- 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...
```

- we can see difference between these initial approaches to solving same problem
- 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...`*
- second example uses *recursion*
  - *i.e. keep calling first step, or function, until a break is achieved*

Algorithms and Data Structures

recursion for fun - part 4

- consider this problem using two sample implementations
  - *reflect sample outlines*
- first example uses a `while` loop
  - *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
```

- while main sweet jar still contains sweets
  - *`pick_a_sweet()` function will choose a sweet*
  - *function will need to return chosen sweet*
  - *check its colour and remove it from the main jar*
- then check current sweet's colour
  - *add it to the second jar if it's not gold*
- if sweet is gold, you win and the loop will exit

Algorithms and Data Structures

recursion for fun - part 5

- 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);
```

- able to loop through passed jar of sweets
  - *check each one until we find winning gold sweet*
- we make a number of assumptions
  - *i.e. passed jar as an array, value of sweet's colour as a string....*
- also a slow search
  - *only have information for required colour of winning sweet*
  - *need to iterate through whole jar*

Algorithms and Data Structures

recursion for fun - part 6

- second implementation
  - *consider a solution using recursion*
- initially consider 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
```

- recursive example follows same underlying pattern as `while` option
- instead of loop we may now call `search_sweets()` method
  - *for all sweets in the jar*
  - *or until we find the gold sweet*

Algorithms and Data Structures

recursion for fun - part 7

- implement this algorithm using recursion with 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!`);
    }l,

}

// 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);
```

- need to check availability of sweets in jar
  - *allows us to check for winning gold sweet*
- conditional statements follow same pattern as previous JavaScript example
- may now recursively call `searchSweets()` function

Video - Algorithms and Data Structures

Recursion for Fun

Recursion and Fun - JavaScript - UP TO 7:32

Source - Recursion and Fun - JavaScript - YouTube

recursion - linked list

- consider earlier linked list
  - *may define an algorithm for implementing procedures on a list using recursion*
- e.g. following algorithm may be outlined to find last element of 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 *complexity* of this algorithm
  - *the procedure has **linear time complexity***
  - *i.e. if length of list is increased, execution time will likewise increase by same factor*
- performance does not mean that lists are always inferior to arrays
- lists are not an ideal data structure for certain uses
  - *regardless of applied common algorithms*
- i.e. when an application needs to access last element of a longer list

Algorithms and Data Structures

stacks and the call stack

- a brief segue into *stacks*
  - *in particular a consideration of call stack used with program execution*
- key to understanding execution of many algorithms in code
  - *e.g. a better understanding of recursion for development*
- if we don't understand how order of execution is tallied and reconciled by applications
  - *we'll struggle to clearly understand nature of algorithms*
  - *and their general usage*

Algorithms and Data Structures

stacks - intro - part 1

- *stack* data structure commonly represented as
  - *a modified, restricted array or list*
  - *used in various programming and scripting languages*
- *stack* is an efficient data structure
  - *used for many development purposes*
- Data may only be added and removed from top of structure
  - *affords ease of implementation and speed*
- commonly refer to a stack as *last in, first out*
  - *or LIFO*
- stack of plates in a restaurant kitchen
  - *a good analogy of this structure's usage*
  - *dirty plates are added to top of stack*
  - *a dishwasher will wash these plates from the top down*
  - *so last plate on the stack will be washed first*

Video - Algorithms and Data Structures

Python Stacks

Python Stacks

Source - Stack - YouTube

Algorithms and Data Structures

stacks - intro - part 2

- *push* method may add a value to the end of an array
- *pop* method may be used to remove last value in array
- stack is a data structure
  - *allows values to be pushed into it*
  - *and popped from it as needed*
- last item added is now the first removed
- Stacks may be used for many different purposes in development
  - *e.g. execution requests to function calls*
- a stack is a common structure
  - *e.g. storing lists of items for ordered usage*

Algorithms and Data Structures

stacks - intro - part 3

- similar in nature to a linked list
- restricted use of a *Stack* normally defines alternative names for primitive operators
- conceptually - commonly define construction and access as follows
  - *i.e. for a custom implementation of a Stack data structure*
- constructor to enable instantiation of Stack data structure
  - *e.g.* `EmptyStack` *or simply* `Stack`
- basic selectors for required default functionality
  - `top(stack)` *- return top element from stack*
  - `pop(stack)` *- returns stack without top element*

stacks - intro - part 4

- specific implementation of such selectors may vary from language to language
- fundamental concept of the data structure remains consistent
- also see similar `true` and expected relationships for a stack
- similar to those seen for a linked list
- e.g.
  - `isEmpty(EmptyStack)`
  - `not isEmpty(push(x, s))` *(for any* `x` *and* `s`*)*
  - `top(push(x, s)) = x`
  - `pop(push(x, s)) = s`

Algorithms and Data Structures

stacks - intro - part 5

- conceptually define following as useful comparison
  - *a* `list` *and* `stack`

structure	constructors	selectors	condition
list	EmptyList, MakeList	first, rest	isEmpty
stack	EmptyStack, push	top, pop	isEmpty

Video - Algorithms and Data Structures

Stacks and the Call Stack - part 1

Call Stack - UP TO 4:50

Source - Call Stack - YouTube

Algorithms and Data Structures

stacks and the call stack - part 1

- as a computer executes code, commands, and various logic...
- uses an internal *stack*
  - *the call stack*
  - *records and checks order of execution*
- e.g. consider the following Python code

```
def greetings(name):
    print "hello, " + name + "!"
    more_greetings(name)
    print "ready to leave..."
    goodbye()
```

- as we execute `greetings()` function
  - *also execute other defined custom functions*
  - *i.e.* `more_greetings()` *and* `goodbye()`
- also execute `print` function - internal to Python
- initially consider custom functions relative to call stack

```
def more_greetings(name):
    print "how are you, " + name + "?"

def goodbye():
    print "take care, goodbye!"
```

Algorithms and Data Structures

stacks and the call stack - part 2

- as we execute function `greetings()`
  - *system will allow memory*
  - *i.e. a container or box specific to that function call*
- memory will contain function, variable `name` with passed value
- e.g.

```
----------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
----------------------
```

- every time we make a function call
  - *system will save values for all of the variables for that call*
- our code printed the inital greeting

```
"hello, Daisy!"
```

stacks and the call stack - part 3

- then execute another function call `more_greetings()`
- system will again allocate memory for this specific function call

```
----------------------
|   more_greetings   |
|--------------------|
| name:    |  daisy  |
----------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
----------------------
```

stacks and the call stack - part 4

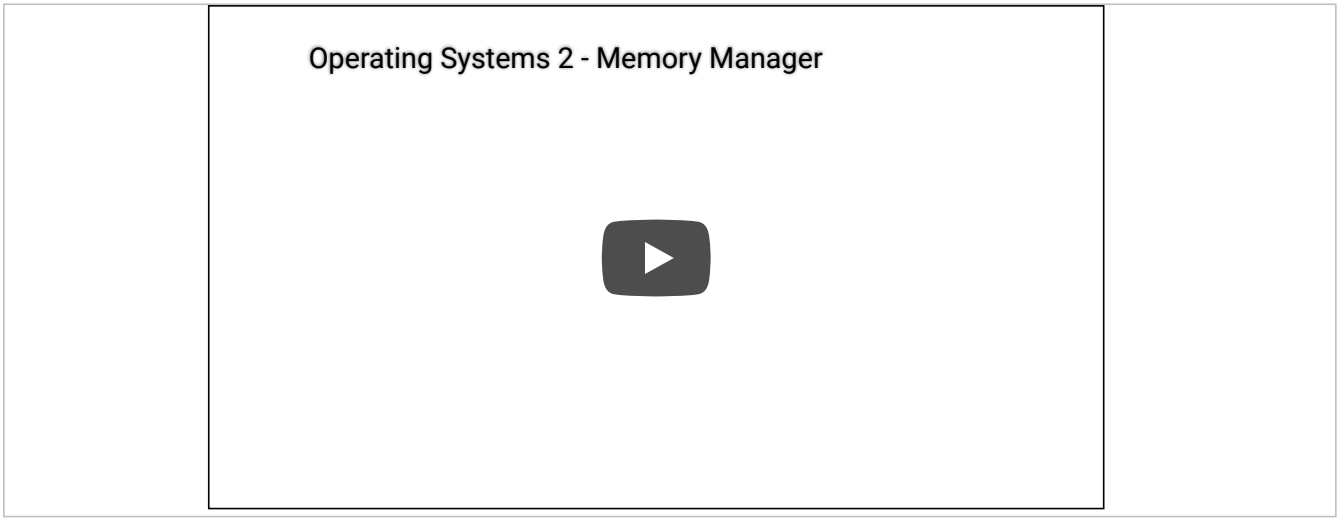
- system is using a *stack* for this memory storage
- i.e. its stacking boxes of memory for current function calls
- then print the second greeting

```
"how are you, Daisy?"
```

- return to function call
- top of call stack
  - *i.e. box of memory for* `more_greetings`
  - *now popped off stack*

Video - Algorithms and Data Structures

Stacks and the Call Stack



Memory Manager - UP TO 7:11

Source - Memory Manager - YouTube

Algorithms and Data Structures

stacks and the call stack - part 5

- stack returns to the following

```
----------------------
|      greetings     |
|--------------------|
| name:    |  daisy  |
----------------------
```

- as we called `more_greetings()` function
  - *initial function for `greetings()` only partially completed*
- i.e. call a function from another function
  - *current function calling execution is paused*
  - *paused in partially completed state*
- values of defined variables for function still stored in memory

Algorithms and Data Structures

stacks and the call stack - part 6

- now back to `greetings()` function
- may continue to execute that function
- e.g. continue by printing

```
"ready to leave..."
```

- then call `goodbye()` functon
- function will be added to top of stack
  - *then executed*
  - *then exit from function back to `greetings()` function*
- at end of initial function call for `greetings()`
  - *exit that function and stack is now clear*
- call stack may store required variables for multiple functions
  - *required order of execution for defined code*

Algorithms and Data Structures

stacks and the call stack - part 7

- order of execution for functions and application code in JavaScript
  - *defined by call stack*
- *call stack* provides context for ordered execution of code
- e.g. a conceptual stack of ordered execution

```
not in function
    in greetings function
        in console.log
    in greetings function
        in moreGreetings function
    in greetings function
        in console.log
    in greetings function
        in goodbye function
not in function
    in console.log
not in function
```

Algorithms and Data Structures

stacks and the call stack - part 8

- this stack represents the following sample JavaScript code

```
function greetings(name) {
  console.log("hello " + name + "!");
  moreGreetings(name);
  console.log("ready to leave...");
  goodbye();
}

function moreGreetings(name) {
  console.log("how are you, " + name + "?");
}

function goodbye() {
  console.log("take care, goodbye!")
}

greetings("Daisy");
console.log("now finished...");
```

- context for this code's execution stored in *call stack*

Video - Algorithms and Data Structures

Stacks and the Call Stack - part 2

Call Stack

Source - Call Stack - YouTube

Resources

JavaScript

- MDN - Arrays
- MDN - Classes
- MDN - Loops and Iteration
- MDN - Prototype
- MDN - Proxy
- MDN - Symbol
- MDN - Symbol.iterator
- Recursion and Fun - YouTube

Java

- Recursion - YouTube
- Sort an Array with Comparator - YouTube

Python

- Stacks - YouTube

Various

- Call Stack - YouTube
- Process Manager - YouTube
- Process Manager - Algorithms and Management - YouTube
- Process Manager - Scheduler - YouTube
- Text based game Vexed with linked list - YouTube