2018 SpeedStudy [1]

P1xt

January 25, 2018

[1] An adventure to explore how much of MITOCW's (Computer Science, Physics and Mathematics) Curriculum I can thoroughly learn in a year.

Abstract

I'm tackling my 2018 learning goals in true "Speedrun" style. The following list contains far more than I expect I'll be able to complete in 2018, so it's sufficient that I can speedrun the year without running out of items from the list to learn.

My dual main goals are to A) study algorithms in depth and B) explore Physics. To that end, I am including a variety of Computer Science, Mathematics and Physics courses, with no prescribed order. I'll just select the next course in whatever list I'm most inspired to continue each time I'm ready to progress to a new course.

Additionally, I'm including a variety of "bonus point" opportunities for reading books, completing projects, and completing algorithmic challenges.

My goal is to see how many points I can rack up by December 31, 2018.

Current Course: 18.01
Total Points: 0

Activity	Points
Basic Project	100
Substantial Project	200
Large Project	300
Gigantic Project	400
Duolingo (1 level)	100
Book	200
Tutorial site (Udemy/Pluralsight/Egghead/Treehouse-/etc) course	100
University level course (Coursera/edX/MITOCW/etc)	500
Physical Activity (30 minutes)	10
Analytics Vidhya Competition	50
Crowd Analytix Competition	50
Kaggle Competition	50
Driven Data Competition	50
Blog Post on any topic	15
Blog Post Tutorial	40
Video Tutorial	50
Open Source PR	50
Module to npm	200
Team Up for a project	100
CodeWars (10 problems)	50
CodinGame (1 Tier)	100
CodinGame (1 Bot Competition)	20
HackerRank (10 problems)	50
Google Code Jam (1 round from past contest)	100

Contents

Course: 6.0001

Notes

All knowledge falls into two categories - **Declarative knowledge** (statements of fact) and **Imperative knowledge** (instructions "how to" compute something).

An **Algorithm** is a list of instructions that describe a computation that when executed on a set of inputs will proceed through a set of well-defined states and eventually produce an output.

Fixed-program computers only have the code necessary to do one thing, whereas **stored program computers** are more flexible and execute any computation in their instruction set.

Syntax - which strings of characters and symbols are well formed.

Semantics - associates a meaning with each syntactically correct string that has no static semantic errors.

Static Semantics - which syntactically valid strings have meaning.

Scaler objects are indivisible, wheras **non-scalar objects** such as strings have internal structure. Python's four types of scalar objects are int, float, bool, and None.

Objects and **Operators** are combined to form **Expressions** which each evaluate to an object of some type.

Operations on strings, tuples, ranges and lists - seq[i],len(seq), seq1+seq2, n*seq, seq[start:end], e in seq, e not in seq, for e in seq.

Strings - can be indexed into, sliced, and length can be found with the len function. Contain only characters. Methods include: s.count(s1), s.find(s1), s.rfind(s1), s.index(s1), s.rindex(s1), s.lower(), s.replace(old, new), s.rstrip(), s.split(d).

Tuples - immutable ordered sequences of elements, like strings, but can be ordered sequences of any type, not just characters. Format - ('a', 'b', 1, 1.2, 'a'). Can be concatenated, indexed and sliced like strings.

Ranges - immutable like strings and tuples but contain only integers. The range function returns an object of type range and takes three arguments, start, stop and step. Step may be negative. Step default is 1. Start defaults to 0.

Lists - like tuples and can contain collections composed of any type of object, but use square brackets and are mutable. Use id() to get the id of a list object. Can use append(e), count(e), insert(i,e), extend(L1), remove(e), index(e), pop(i), sort(), reverse().

List Comprehension - creates a new list in which each element is the result of applying a given operation to a value from a sequence (another list). Like .map() in JavaScript. Syntax: L = [x**2 for x in range(1, 7)] will cause L to contain [1, 4, 9, 16, 25, 36].

Dictionaries - objects of type dict, like lists but are indexed as key/value pairs. Like lists, are mutable. Use keys that are **hashable types** which include all immutable data types, but no mutable ones. Methods avaiable on dicts: len(d), d.keys(), d.values(), k in d, d[k], d.get(k, v), d[k]=v, del d[k], for k in d.

Classes - used to implement data abstractions, define an interface between the data type and the rest of the program, comprised of both objects and available operations on those objects. Can be instantiated to create a new instance of the class, and uses dot notation to access attributes associated with the class. Special methods __init__ and __str__ initialize an object of the class, and return a string representation of the object. **Inheritance** allows an class of one type to inherit attributes from another class. Subclases extend the behavior of their superclass. The Subclass should respond properly to any request that it's superclass could handle, the converse is not necessarily true.

Encapsulation - bundling together data attributes and the methods for operating on them.

Information Hiding - how methods in a class are implemented are irrelevant to clients of that class so long as they conform to the specification they agree to.

Generators - any function containing a yield statement. Typically used in conjuction with for statements.

Lambda expression - anonymous function, syntax: lambda <variables>: expression

Floats - are approximations of values containing a fractional or decimal amount. May not be exact. Significant digits denote the precision. An approximation is the rounded value.

Type Casting allows for conversion between types. You can convert one type to another by using the name of the result type you want. int('23') converts the string 23 to the integer 23.

Conditionals allow a program to branch depending on some condition (if x do y, else do z)

Iteration allows a program to repeat some series of instructions until a condition is met.

Computational Complexity is the study of the intrinsic difficulty of problems. Typically uses the asymptotic notation "Big O" which is used to give an upper bound on the order of growth of a function.

Constant time implies that the running time of a program does not increase with the size of the input to the program.

Bisection search - approximates by making educated guess at high and low values, checking the midpoint between the two, and then replacing either the high or low value with the mid value and repeating until a "close enough" value is found. Effectively halves the search domain each iteration.

Successive approximation - Newton-Raphston method for finding the square root
$guess - \frac{p(guess)}{p'(guess)}$
is a better approximation for the root of a polynomial than $guess$.

Recursion - when a function calls itself to iteratively solve a smaller subset of the same problem.

Modules - a .py file containing Python definitions and statements. A program may consist of many modules.

Files - can be accessed via open(fh, 'w | r | a'), fh.read(), fh.readline(), fh.readlines(), fh.write(s), fh.writeLines(S), fh.close()

Testing - running a program to determine whether it works as intended. The intention is to show what bugs exist. **Glass box** tests paths through a program whereas **Black box** tests paths through the specification. Unit testing tests individual functions work properly, integration testing tests the program as a whole. Regression testing ensures that what used to work still works. Drivers simulate part of the program that use the unit being tested, stubs simulate parts of the program used by the unit being tested.

Exception - something that does not conform to the norm. Should be handled by the program. Use try: except <exception>: block to handle.

Assertions - confirm that the state of a computation is as expected. Will raise an AssertionError exception if it evaluates to False.

Debugging - attempting to fix a program that you know isn't working properly.

print() - prints to console

input('prompt string') - gets input from the user

type(X) - returns the type of X

while <condition>: - loops through indented statements while the condition is true

for <item in list: - loops through each item, use break to exit the loop early

def <functionName(x, y)>: - defines a function, return exits the function, can pass parameters by position or keyword (like y='something' as a parameter), can set default in parameter list, like y='something' to be used when no y is passed. **Higher-order functions** take another function as an argument.

Read everything through Chapter 10. Currently at 8.2.1

Problem Set 0

Problem Problem Set 0.

Solution. In Python:

```
"""
Problem Set 0

Read in two numbers then print out the first raised to the power of the second, and on the next
    line
print the log base 2 of the first.
"""

import math

X = int(input("Enter number x: "))
Y = int(input("Enter number y: "))
print("x**y = " + str(X**Y))
print("log(x) = " + str(math.log2(X)))
```

■

Problem Set 1

Problem Problem Set 1.

Solution. In Python:

```
"""
Problem Set 1A

Calculate the number of months it will take to save up for the down payment
on your dream home.
"""


ANNUAL_SALARY = float(input("Enter your annual salary: "))
PORTION_SAVED = float(
    input("Enter the percent of your salary to save, as a decimal: "))
TOTAL_COST = float(input("Enter the cost of your dream home: "))
PORTION_DOWN_PAYMENT = 0.25
CURRENT_SAVINGS = 0
R = 0.04
MONTHLY_SALARY = ANNUAL_SALARY / 12
TOTAL_DOWN_PAYMENT = TOTAL_COST * PORTION_DOWN_PAYMENT
MONTHS = 0

while CURRENT_SAVINGS < TOTAL_DOWN_PAYMENT:
    CURRENT_SAVINGS = CURRENT_SAVINGS + \
        (CURRENT_SAVINGS * R / 12) + (MONTHLY_SALARY * PORTION_SAVED)
    MONTHS = MONTHS + 1

print("Number of months: " + str(MONTHS))
```

```
"""
Problem Set 1B

Calculate the number of months it will take to save up for the down payment
on your dream home. Account for a semi-annual raise.
"""


ANNUAL_SALARY = float(input("Enter your annual salary: "))
PORTION_SAVED = float(
    input("Enter the percent of your salary to save, as a decimal: "))
TOTAL_COST = float(input("Enter the cost of your dream home: "))
SEMI_ANNUAL_RAISE = float(input("Enter the semi-annual raise, as a decimal: "))
PORTION_DOWN_PAYMENT = 0.25
CURRENT_SAVINGS = 0
R = 0.04
MONTHLY_SALARY = ANNUAL_SALARY / 12
TOTAL_DOWN_PAYMENT = TOTAL_COST * PORTION_DOWN_PAYMENT
MONTHS = 0

while CURRENT_SAVINGS < TOTAL_DOWN_PAYMENT:
    MONTHS = MONTHS + 1
    CURRENT_SAVINGS = CURRENT_SAVINGS + \
        (CURRENT_SAVINGS * R / 12) + (MONTHLY_SALARY * PORTION_SAVED)
    if MONTHS % 6 == 0:
        MONTHLY_SALARY = MONTHLY_SALARY + (MONTHLY_SALARY * SEMI_ANNUAL_RAISE)

print("Number of months: " + str(MONTHS))
```

```
"""
Problem Set 1C

Calculate the right amount to save, print number of steps in
bisection search.
"""
INITIAL_ANNUAL_SALARY = float(input("Enter your annual salary: "))

PORTION_DOWN_PAYMENT = 0.25
RATE_OF_RETURN = 0.04
SEMI_ANNUAL_RAISE = 0.07
MONTHS = 36
ACCURACY = 100
MONTHLY_RATE_OF_RETURN = RATE_OF_RETURN / 12
TOTAL_COST = 1000000
DOWN_PAYMENT = TOTAL_COST * PORTION_DOWN_PAYMENT
CURRENT_SAVINGS = 0.0
SEARCH_STEPS = 0
SEED_HIGH = 10000
HIGH = SEED_HIGH
LOW = 0
PORTION_SAVED = (HIGH + LOW) / 2


while abs(CURRENT_SAVINGS - DOWN_PAYMENT) > ACCURACY:
    ANNUAL_SALARY = INITIAL_ANNUAL_SALARY
    MONTHLY_SALARY = ANNUAL_SALARY / 12
    MONTHLY_DEPOSIT = MONTHLY_SALARY * (PORTION_SAVED / 10000)
    CURRENT_SAVINGS = 0.0

    for month in range(1, MONTHS + 1):
        CURRENT_SAVINGS = CURRENT_SAVINGS * \
            (1 + MONTHLY_RATE_OF_RETURN) + MONTHLY_DEPOSIT

        if month % 6 == 0:
            ANNUAL_SALARY = ANNUAL_SALARY * (1 + SEMI_ANNUAL_RAISE)
            MONTHLY_SALARY = ANNUAL_SALARY / 12
            MONTHLY_DEPOSIT = MONTHLY_SALARY * (PORTION_SAVED / 10000)
    PREV_PORTION_SAVED = PORTION_SAVED
    if CURRENT_SAVINGS > DOWN_PAYMENT:
        HIGH = PORTION_SAVED
    else:
        LOW = PORTION_SAVED
    PORTION_SAVED = int(round((HIGH + LOW) / 2))
    SEARCH_STEPS = SEARCH_STEPS + 1
    if PREV_PORTION_SAVED == PORTION_SAVED:
        break

if PREV_PORTION_SAVED == PORTION_SAVED and PORTION_SAVED == SEED_HIGH:
    print("It is not possible to pay the down payment in three years.")
else:
    print("Best savings rate:" + str(PORTION_SAVED / 10000))
    print("Steps in bisection search:" + str(SEARCH_STEPS))
```

■

Problem Set 2

Problem .

Solution. ■

Problem Set 3

Problem .

Solution. ■

Problem Set 4

Problem .

Solution. ■

Problem Set 5

Problem .

Solution. ■

Book: Grokking Algorithms

Notes

Introduction to Algorithms

Binary search

- requires a sorted list of elements
- returns the position within the list of the element being searched for
- returns null if the element being searched for isn't present in the list
- mechanic: eliminate half the list by comparing the search input to the element at the mid point of the list to determine whether the element is in the first half (is less than mid) or second half (is greater than mid). Continue eliminating half repeatedly until the search input equals the value at mid, or there are no more elements in the list to check (value does not exist in list).

Logarithms

- the flip side of exponents
- $\log_2 8 == 3$ because $2^3 == 8$

Big O notation

- examines how fast an algorithm is
- compares the number of operations to determine how fast the algorithm grows
- establishes a worst case run time
- efficiency, fastest to slowest
  - $O(logn)$ - log time (Binary search)
  - $O(n)$ - linear time (Simple search)
  - $O(n * logn)$ - (quicksort)
  - $O(n^2)$ - (selection sort)
  - $O(n!)$ - (travelling salesman)

Selection sort

- $O(n^2)$
- find biggest item in list, add to new list
- repeat until all items transferred to new list

Arrays and linked lists

- Arrays
  - Allow fast reads
- Linked lists
- All elements must be the same type
- All elements are stored contiguously in memory
  - Allow fast inserts
  - Allow fast deletes
  - Each item is stored independently in memory

Recursion

A recursive function calls itself, passing the new instance a smaller subset of the same problem. The function must be supplied a "base case" to indicate that some value be returned rather than a new recursion so as to avoid an infinite loop.

Quicksort

- Divide and Conquer
- Base case - simplest case
- Divide problem until it becomes the base case
- Quicksort - pick a pivot, create two sub-arrays, elements smaller and larger than pivot,
  quicksort recursively on the sub-arrays.

Inductive proof has base case and inductive case. (if it works for base case, it will work for one larger than base, if it works for that, it will work for all.

Hash tables

Maps strings to numbers. Is a dictionary in Python. Is good for lookups. Useful for caching. Needs a good hash function to ensure keys map evenly over the hash. Take constant O(1) time. Avoid collisions with a low load factor (items/slots) and a good hash function.

Breadth-first search

Allows you to find the shortest path. A graph models a set of connections as a set of nodes with edges that can be connected to other nodes (neighbors). Good for answering two types of questions:

- Is there a path between two items.
- What is the shortest path between two items.

Queue is FIFO. Stack is LIFO. A tree is a unidirectional graph.

Dijkstra's algorithm

Finds fastest path (as opposed to the shortest path from breadth-first). Find cheapest node, check if there is cheaper path to neighbors (if so update their costs), repeat for every node, calculate final path. Only works on directed acyclic graphs (DAGs)

```
node = find_lowest_cost_node(costs)
while node is not None:
  cost = costs[node]
  neighbors = graph[node]
  for n in neighbors.keys():
    new_cost = cost + neighbors[n]
    if costs[n] > new_cost:
      costs[n] = new_cost
      parents[n[ = node
  processed.append(node)
  node = find_lowest_cost_node(costs)
```

Greedy algorithms

Used to tackle problems that don't have a fast algorithmic solution (np-complete problems). Simple to write and get "pretty close". Optimize locally, hoping to end up with an overall optimal solution. If you have an NP-complete problem, your best bet is to use an approximation algorithm. Greedy algorithms run fast and are easy to write so they are good options for approximation algorithms.

Dynamic programming

Solve sub-problem which leads to solution to entire problem. Doesn't operate on fractions, either takes the whole or nothing. Each subproblem must be discrete. Sometimes best solution doesn't fill knapsack completely. Use a grid to solve, each cell is a sub-problem.

K-nearest neighbors

used for classification (categorization in a group) and regression (predicting a response). Feature extraction means converting an item into a list of numbers that can be compared. Pick good features for success.

Where to go next

The Fourier transform, given a thing, it will give you the ingredients (sub parts) of that thing. Distributed algorithms, can run across multiple machines.

Note: I WAS doing all of the problems. But they were trivial. So I stopped. This was a decent overview book but it pretty much stayed in beginner-land, I'll do exercises when I hit up Cormen or Sedgewick.

Book: The Pragmatic Programmer

Notes

A Pragmatic Philosophy

Take responsibility. Be honest and direct. Provide solutions, provide options, not excuses. Protect against entropy. Don't leave a mess. Startup fatigue cure: build minimal, then add. Remember the big picture. Write software that is good enough. Know when to stop.

Knowledge Portfolio - invest regularly, diversify, keep a balance between conservative and high risk/rewards, buy low sell high, review and rebalance.

Goals

- learn at least one new language every year
- read a technical book each quarter
- read nontechnical books too
- take classes
- stay current
- get wired (newsgroups, online papers, etc)

A Pragmatic Approach

Don't repeat yourself. Make it easy to reuse. Eliminate effects between unrelated things. There are no final decisions. Use Tracer bullets to find the target. Prototype to learn. Estimate to avoid surprises. Iterate the schedule with the code.

The Basic Tools

Keep knowledge in plain text. Use the command line. Use a single editor well. Always use source code control. Fix the problem, not the blame. Don't Panic. Don't assume it, prove it. Write code that writes code.

Pragmatic Paranoia

You can't write perfect software. Design by contract. Crash early. If it can't happen, use assertions to ensure it wont. Use exceptions for exceptional problems. Finish what you start.

Bend, or Break

Minimize coupling between disparate portions of code. Configure, don't integrate. Put abstractions in code, details in metadata. Analyze workflow to improve concurrency. Design using services. Always design for concurrency.

While You Are Coding

Don't program by coincidence. Program deliberately. Estimate the efficiency of your algorithms. Refactor early, refactor often. Design to test. Test your software. Don't use Wizard code you don't understand.

Before The Project

Don't gather requirements, dig for them. Work with a user to think like a user. Abstractions live longer than details. Use a project glossary. Don't think outside the box, find the box. Listen to nagging doubts, start when you're ready. Some things are better done than described.

Pragmatic Projects

No broken windows, boil some frogs, communicate, DRY, use automation, make it good enough. Automate builds. Test early. Test often. Test automatically. Build documentation in, don't bolt it on. Gently exceed your users' expectations. Sign your work. Take responsibility for it.

Book: Incidents in the Life of a Slave Girl by Harriet Jacobs

> "That poor ignorant woman thought that America was governed by a Queen, to whom the President was subordinate. I wish the President was subordinate to Queen Justice."

> "Yet few slaveholders seem to be aware of the widespread moral ruin occasioned by this wicked system. Their talk is of blighted cotton crops – not the blight on their children's souls."

I could ramble on, give a book report, if you will – but I won't. I'll let those two quotes, written over 160 years ago, carry that for me.

I selected this as my first book to read in 2018 because 2017 troubled me. I wanted to look back as a means of looking forward. I read it in one sitting and am preparing to head to bed, reflecting on white supremacists and tiki torches and a President who cares more about monuments to slaveholders than he does about living breathing human beings – and I wonder how many generations must pass before they rise above the "moral ruin".

Book: YDKJS: Up & Going

Notes

Into Programming

- **Code** - a program telling the computer what tasks to perform.
- **Statements** - groups of words, numbers and operators that performs a specific task.
- **Expressions** - any reference to some number of variables or values combined with operators.
- **Output** - console.log / alert / etc
- **Input** - prompt
- **Operators** - assignment(=),
  mathematics (*/+-%),
  compound assignment (+=, -=, *=, /=),
  increment/decrement (++, –),
  object property access(.),
  equality (==, ===, !=, !==),
  comparison (>, <. >=, <=),
  logical(&&, ||)
- **Values & Types** - number(math), string(to print), boolean(decision), JavaScript coerces between types automatically, but you can explicitly convert using Number(), etc.
- **Code Comments** - with /* */ or //
- **Variables** - symbolic container for a value
- **Conditionals** - if / else
- **Loops** - while, do..while, for
- **Functions** - named sections of code, can be passed parameters
- **Scope** - each function gets it's own scope, can be nested. A scope can access it's own variables, plus the variables of any scope it is nested inside, plus the variables of any scopes those scopes are nested inside, all the way up to the global scope.

Into JavaScript

- **Values & Types** - string, number, boolean, null, undefined, object, symbol. Can determine a value's type with typeof. Note - typeof null returns "object".
- **Object** - a compound value you can set properties on.
- **Arrays** - an object that hold a set of values in numerically indexed positions. The first index is at position 0.
- **Built-in Type Methods** - each type has some number of built-in methods, such as toUpperCase() for strings.
- **Comparing Values** - a comparison is falsy ("", 0, -0, NaN, null, undefined, false) or truthy. You can explicitly coerce a value to a specific type, or JavaScript will implicitly coerce before doing comparisons. You can compare for equality (==, ===, !=, !==) or relationship (>, <, >=, <=).
- **Variables** - An identifier must start with a-z, A-Z, $, or _. It can then contain any of those characters plus the numerals 0-9.
- **Function Scopes** - use var to ensure variable is set in the current scope, omitting it will declare the variable on the global scope. Variables belonging to a particular scope are available to all scopes nested within that scope.
- **Conditionals** - in addition to if / else, if / else if / else, and switch /case are also available. Use break to end a case within a switch.
- **Strict Mode** - opt in with "use strict" for additional rules implemented by the JavaScript compiler, generally makes code safer and more easily optimizable.
- **Functions as Values** - Functions are just variables, they can be passed to and returned from other functions.
- **IIFEs** - Immediately Invoked Function Expressions.

$$(function IIFE()console.log("Hello!");)();$$

causes the function to be executed immediately, instead of the normal function behavior (wait for some other part of the program to explicitly call the function). Important to note - the scope of the IIFE is set when it's executed, so it retains a snapshot of that scope for the life of its execution, even if the scope outside it changes before it completes.

- **Closure** - a way to remember and continue to access a function's scope even after the function has finished running.
- **Modules** - let you define private implementation, as well as a public API.
- **this** - IS NOT the same as this in OO patterns. In non-strict mode, this within a function points to the global object. In strict mode, this within a function is undefined. ((Revisit in later book))
- **Prototypes** - prototype linkage for property lookup on an object is setup at the time the object is created. If you attempt to access a property that is not available on an object, it will follow the prototype chain to see if the property exists higher up the chain. IS NOT inheritance, is behavior delegation.
- **Transpiling / Polyfills** - allows you to use new language features by converting them to old syntax executable within your runtime environment (node / browser).

Into YDKJS

- **Scope & Closures** - hoisting / lexical scope / closure / module pattern
- **this & Object Prototypes** - thorough review of "this" - prototypal delegation
- **Types & Grammar** - Type coersion
- **Async & Performance**- Promises / Generators / parallelism with Web Workers and SIMD / Optimization techniques
- **ES6 & Beyond** - both the short- and Yomid-term visions of where the language is headed, not just the known stuff like ES6 but the likely stuff beyond

Frontend Frameworks: React

Basics

- **A React component class** - extends React.component. Inside a React component class, access props with this.props and state with this.state.
- **Functional components** only consist of a render method. They do not define a class extending React.Component, they are simply a function that takes props and returns what should be rendered. Inside a functional component, access props with props not this.props.
- **A component** - takes in parameters, called props, and returns a hierarchy of views to display via the render method.
- **The render method** - returns a description of what you want to render, and then React takes that description and renders it to the screen. In particular, render returns a React element, which is a lightweight description of what to render.
- **React element** - React element. Created with JSX or React.createElement().
- You can put any JavaScript expression within braces inside JSX. Each React element is a real JavaScript object that you can store in a variable or pass around your program.
- In JavaScript classes, you need to explicitly call super(); when defining the constructor of a subclass.
- **this.setState** -Whenever this.setState is called, an update to the component is scheduled, causing React to merge in the passed state update and rerender the component along with its descendants.
- When you want to aggregate data from multiple children or to have two child components communicate with each other, move the state upwards so that it lives in the parent component. The parent can then pass the state back down to the children via props, so that the child components are always in sync with each other and with the parent.
- **Controlled Components** - receive their values from their parent and do not keep their own state.
- The biggest benefit of immutability in React comes when you build simple pure components. Since immutable data can more easily determine if changes have been made, it also helps to determine when a component requires being re-rendered.
- props.onClick() will not work because it would call props.onClick immediately instead of passing it down. Use props.onClick.

Flexbox

Why Flexbox

- Vertically centering a block of content inside its parent.
- Making all the children of a container take up an equal amount of the available width/height, regardless of how much width/height is available.
- Making all columns in a multiple column layout adopt the same height even if they contain a different amount of content.

How To - parent container

- **display: flex;** - on the parent element to create a flex container.
- **flex-direction: column | row | column-reverse | row-reverse;** - on the parent element to specify the direction of the main flex axis.
- **flex-wrap: wrap;** - on the parent element to cause the rows or columns to wrap.
- **flex-flow: row wrap;** - shorthand to combine flex-direction and flex-wrap.

How To - children

- **flex: 1;** - controls what percentage of the space the child consumes. Shorthand for flex-grow, flex-shrink, and flex-basis.

- **align-items** - controls where the flex items sit on the cross axis.
- **align-self** - overrides align-items for an individual element.
- **order** - determines the order in which children will appear.