2018 SpeedStudy [1]

P1xt

January 9, 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

Hash tables

Breadth-first search

Dijkstra's algorithm

Greedy algorithms

Exercises

Problem 1.1.

Solution. 7 ■

Problem 1.2.

Solution. 8 ■

Problem 1.3.

Solution. Olog(n) ■

Problem 1.4.

Solution. O(n) ■

Problem 1.5.

Solution. O(n) ■

Problem 1.6.

Solution. O(n) ■

Problem 2.1.

Solution. A linked lists - they'll have fast inserts, plus since every element will need to be read, array advantage on reads won't apply. ■

Problem 2.2.

Solution. A linked list - the operation is insert heavy and only needs to shift the first element off the list ■

Problem 2.3.

Solution. A sorted array - for selection sort it's better to use an array because with it, you can immediately access the mid-point. ■

Problem 2.4.

Solution. The downside is that inserts into an array is slow. So though retrieving data would be quicker, insertiing it would be slower than if a linked list were used. ■

Problem 2.5.

Solution. An array of linked lists would be faster than arrays for inserting, slower than arrays for searching and faster than linked lists for searching. ■

Problem 3.1.

Solution. The greet function is called, it calls greet2, greet2 completes, then greet completes ■

Problem 3.2.

Solution. Eventually the program will fill the call stack and abort. ■

Problem 4.1.

Solution. Python:

```
def sum(list):
  if list == []:
    return 0
  return list[0] + sum(list[1:])
```

■

Problem 4.2.

Solution. Python:

```
def count(list):
  if list == []:
    return 0
  return list[0] + count(list[1:])
```

■

Problem 4.3.

Solution. ■

Problem 4.4.

Solution. ■

Problem 4.5.

Solution. ■

Problem 4.6.

Solution. ■

Problem 4.7.

Solution. ■

Problem 4.8.

Solution. ■

Problem 5.1.

Solution. ■

Problem 5.2.

Solution. ■

Problem 5.3.

Solution. ■

Problem 5.4.

Solution. ■

Problem 5.5.

Solution. ■

Problem 5.6.

Solution. ■

Problem 5.7.

Solution. ■

Problem 6.1.

Solution. ■

Problem 6.2.

Solution. ■

Problem 6.3.

Solution. ■

Problem 6.4.

Solution. ■

Problem 6.5.

Solution. ■

Problem 7.1.

Solution. ■

Problem 8.1.

Solution. ■

Problem 8.2.

Solution. ■

Problem 8.3.

Solution. ■

Problem 8.4.

Solution. ■

Problem 8.5.

Solution. ■

Problem 8.6.

Solution. ■

Problem 8.7.

Solution. ■

Problem 8.8.

Solution. ■

Problem 9.1.

Solution. ■

Problem 9.2.

Solution. ■

Problem 9.3.

Solution. ■

Problem 10.1.

Solution. ■

Problem 10.2.

Solution. ■

Problem 10.3.

Solution. ■

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".