Algorithms, Data Structures, O(my)

Constraints of Time and Space

Lisa Hatch

a Senior Software Engineer, who uses Python

- for an alternate perspective on programming languages
- as a tool for data exploration and machine learning
- to implement DevOps automation
- for new opportunities

https://www.linkedin.com/in/lhatch https://github.com/laeccan

Why?

Why?

- Upcoming job interview
- Refresh computer science concepts
- Life hacking
- Improve programming skills
- Expand technical vocabulary

O(my!) … Big O Notation

What is Big O Notation?

- Method to describe the growth pattern of an algorithm or data structure
- Limiting behavior for growth of a function
- Only describes upper bound -- nothing about lower bound
- Enables growth comparison between algorithms
- Tool to pick algorithm or data structure based on time or space constraints

https://en.wikipedia.org/wiki/Big_O_notation

O(1)

Constant time (not ‘instant’ time)

O(log n)

Logarithmic

Logarithm Refresher

The logarithm is the inverse function to exponentiation.

The logarithm to base 10 (b=10), the **common logarithm**, has many applications in science and engineering.

The **natural logarithm** (b ≈ 2.718, e) has widespread math and physics use.

The **binary logarithm** (b = 2) is commonly used in computer science.

$\log_b(x) = y \qquad b^y = x$

$\log_{10}1000 = 3 \qquad 1000 = 10^3$

$\log_2 64 = 6 \qquad 64 = 2^6$

$e^{\ln x} = x \qquad \text{if } x > 0$

$\ln(e^x) = x$

O(n)

Linear

O(n log n)

Linearithmic, log linear, or quasilinear

$O(n^2)$

Quadratic

O(n!)

Factorial

http://bigocheatsheet.com/

Data Structures

What are common Python data structures?

What are common Python data structures?

- list
- sets
- dictionary
- collections.deque

list features

```
alist = [1, 4, 9, 16, 25]
alist[:]     # slice outputs full list [1, 4, 9, 16, 25]
alist[:-1]  # negative step reverses list [25, 16, 9, 4, 1]

alist.append(x)      # equivalent to [len(alist):] = [x]
alist.insert(i,x)    # insert x at position i
alist.remove(x)      # remove first occurrence of x
alist.pop(i)         # remove and return item at position i; default, pops last item
```

set features

```
aset = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(aset)      # no dups nor ordering, so like {'orange', 'banana', 'pear', 'apple'}

a = {x for x in 'abracadabra' if x not in 'abc'}    # set comprehension of {'r', 'd'}
set <= other                                        # issubset(other)
set | other                                         # union(other)
set & other                                         # intersection(other)
set - other                                         # set elements not in others
set ^ other                                         # elements in either, but not both
```

dictionary features

names = {‘Katharine Jarmul’, ‘Tarek Ziadé’, ‘Wesley Chun’, ‘Guido Van Rossum’, ‘Jessica McKellar’, ‘Andreas Muller’’, ‘Carol Willing’, ‘Luciano Ramalho’, ‘Jake Vanderplas, ‘Jacqueline Kazil’}
twitter_handles = {‘@kjam’, ‘@tarek_ziade’, ‘@wescpy’, ‘@gvanrossum’, ‘@jessicamckellar’, ‘@t3kcit’, ‘@willingcarol’, ’@ramalhoorg’, ‘@jakevd’, ‘@jackiekazel’}

```
for name, twitter_handle in zip(names, twitter_handles):
    print(f”Name: {name}, Twitter: {twitter_handle}”)       # f-string formatting
```

https://zartis.com/top-10-python-developers-on-twitter-at-christmas/

More **dictionary** features

```
names.items()      # (key, value) pairs view of dictionary
a_dict = { ‘skills’: [‘Python’, ‘AWS’, ‘SQL’], ‘locations’: [‘Tokyo’, ‘Chennai’, ‘Atlanta’]
a_dict.keys()      # [‘skills’, ‘locations’]
a_dict.values()    # [ [‘Python’, ‘AWS’, ‘SQL’], [‘Tokyo’, ‘Chennai’, ‘Atlanta’] ]
a_dict.popitem()   # pops random (key,value) and returns it
dict.fromkeys(‘pyladies’, 42)      # sequence defines keys, value set as default
# {'p': 42, 'y': 42, 'l': 42, 'a': 42, 'd': 42, 'i': 42, 'e': 42, 's': 42}
```

Note as of Python 3.7: Dictionary order is guaranteed to be insertion order.

collections.deque features

```
from collections import deque   # generalization of stacks (LIFO), queues (FIFO)
deck = deque('ghi')
deck.append('j')                # add to right side (end)
deck.appendleft('f')            # add to left side (start)

deck.rotate(1)                  # deck.appendleft(deck.pop()), -n allowed also
deck.reverse()                  # inplace reversal, returning None
```

Other **collections** datatypes

```
Point = namedtuple('Point', ['x', 'y'])
p = Point(11, y=22)             # instantiate with positional or keyword arguments
dp = {‘x’: 5, ‘y’: 2}
Point(**dp)                                     # unpack a dictionary to a namedtuple

v  = ChainMap(music_art, art_opera)       # faster than new dict() + updates()

words = re.findall(r'\w+', open('hamlet.txt').read().lower())
Counter(words).most_common(10)        # top 10 listing of (word: count)
```

Data structure terminology

Arrays have indexed storage location; mid-array insert or delete is O(n).

Linked lists provide fast insert and delete operations, but item traversal is O(n).

Queues are list-based: FIFO, **stack** (LIFO), priority or **deque** (double-ended).

Graphs and **trees** are linked lists where a node may link to zero or more nodes. A **heap** is a specialized tree structure, which may be used as a priority queue.

Hash table uses a hash function to compute an index into an array of slots.

Algorithms

Some algorithmic terms

Brute force is a trial and error method to solve a problem.

Recursion provides base and general cases; general case repeatedly used.

Divide and Conquer decompose a problem into smaller subproblems.

Greedy makes the locally optimal choice at each problem stage.

Dynamic programming refers to simplifying a decision by breaking it down into a sequence of decision steps over time by a series of value functions.

Fibonacci series - recursion

Identify base case.

Here there are two: position 0 returns 0, position 1 returns 1

The base case is the key to avoiding infinite recursion or … stack overflow.

Identify recursive case.

How does the function call itself? How are the values returned?

http://bit.ly/2GZw6co
[https://www.cs.usfca.edu/~galles/visualization/RecFact.html]

```
def get_fib(position):
    if (position <= 0):
        return 0
    elif (position == 1):
        return 1
    else:
        return (get_fib(position - 2) +
                get_fib(position - 1))

print(f"Fibonacci 9 = {get_fib(9)}")
```

Quick Sort - divide and conquer approach

1. pick a value in the array at random, then partition based on pivot position
2. move all values larger than pivot above it (left side to right side, after value)
3. move all values lower than pivot below it
4. continue recursively through lower and upper sections of the array, sorting those segments, until the whole array is sorted

http://bit.ly/2GFiTWZ
[https://interactivepython.org/courselib/static/pythonds/SortSearch/TheQuickSort.html]
http://bit.ly/2GJyaWt
[https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html]

Tree Traversal - BFS vs DFS

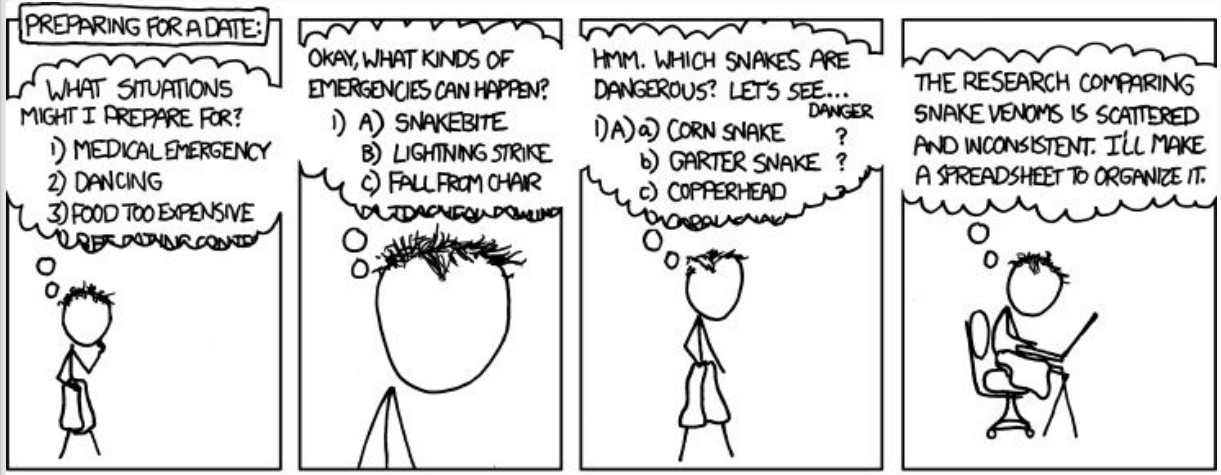
Bread-first looks at all nodes adjacent to node, then moves to next level.

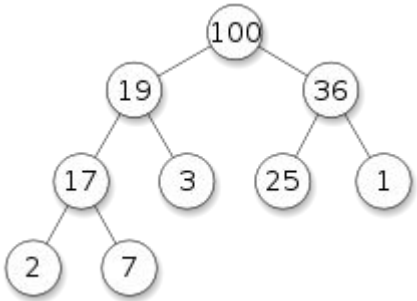
Depth-first follows one path as far as it goes, then repeats.

BFS http://bit.ly/2Sl4RLm

DFS http://bit.ly/2Tir3tW

Xkcd DFS https://xkcd.com/761 or https://www.explainxkcd.com/wiki/index.php/761:_DFS (license https://creativecommons.org/licenses/by-nc/2.5) (https://en.wikipedia.org/wiki/Heap_(data_structure) image)





Knapsack problem - greedy approach

Problem is to maximize the value of the items stuffed in a knapsack, which holds a limited weight.

Greedy Approach:
1. Pick the most expensive thing that will fit in knapsack
2. Pick next most expensive thing that will fit in knapsack

But… this won’t work well if only the most expensive fits, yet the knapsack could hold the second, third, and fourth most expensive items for more $$$.

Knapsack problem - dynamic programming

Dynamic programming divides a hard problem into subproblems to solve.

	1	2	3	4	knapsack subproblems from 1 to 4 pounds
guitar	g	g	g	**g**	$2000, 1 pound
stereo	g	g	g	**s**	$3500, 4 pounds
laptop	g	g	l	**g+l**	$2500, 3 pounds

Consider the problem using only an item from current row and prior rows.
Row 1 pick guitar, $2000. Row 2 pick stereo, $3500.
Row 3 starts off with guitar, then laptop, ends with **guitar and laptop, $4500**.

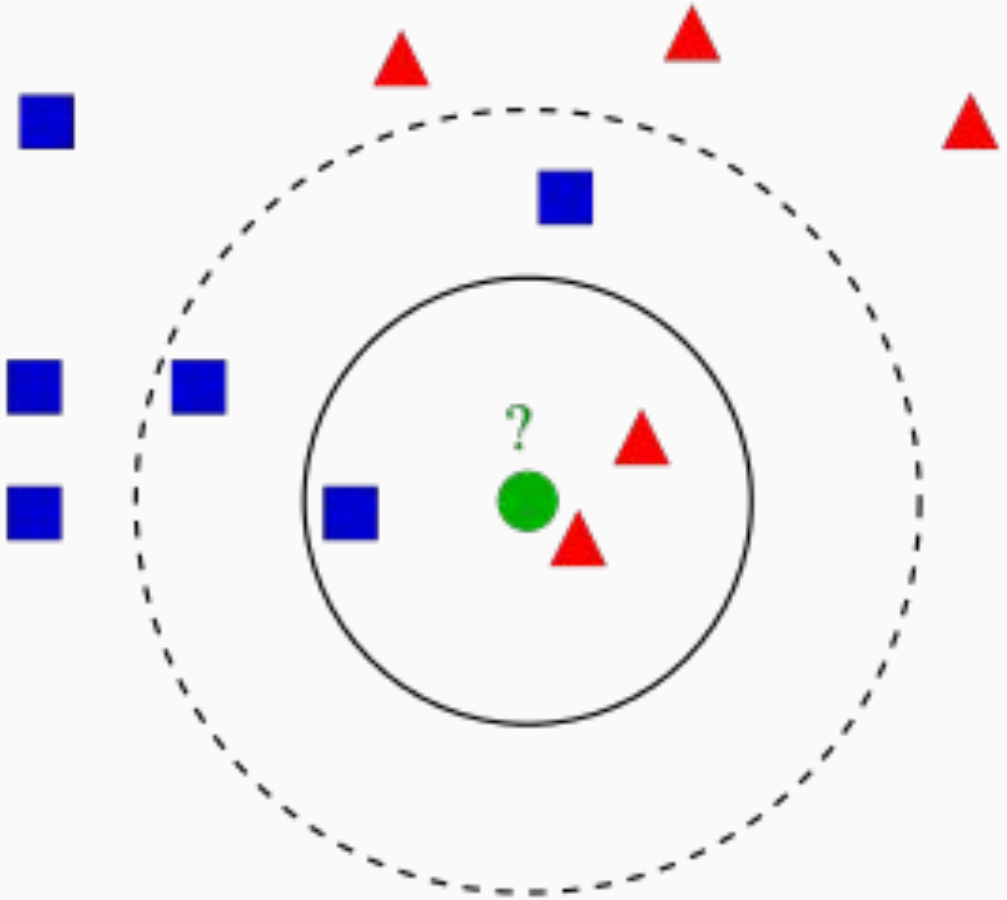
K-nearest neighbor (KNN) - machine learning algorithm

Applying for classification (though KNN may be used for regression).

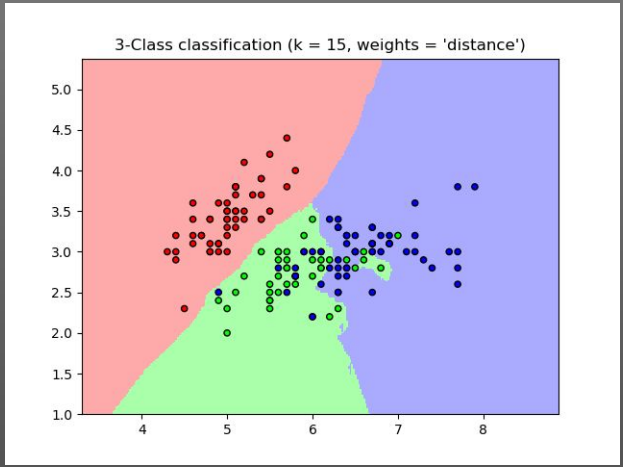
For the **green circle**, what are the nearest **2** items? Blue squares or red triangles?

Distance between items is calculated using Pythagorean formula,
sqrt$((x_1 - x_2)^2 + (y_1 - y_2)^2)$

(https://en.wikipedia.org/wiki/K-nearest_neighbor)



K-nearest neighbor (KNN) - scikit-learn example, K = 15



https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#sphx-glr-auto-examples-neighbors-plot-classification-py

```
from sklearn import neighbors, datasets

n_neighbors = 15
iris = datasets.load_iris() # import some data to play with

# we only take first 2 features. (Avoid ugly slicing by using 2-dim dataset
X = iris.data[:, :2]
y = iris.target
h = .02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...
```

Why? Uses for computational thinking

- **algorithmic thinking** -- developing a set of instructions or sequence of steps to solve a problem
- **evaluation** -- ensuring a solution is appropriate for the need
- **decomposition** -- breaking a problem down into its component parts
- **abstraction** -- hiding detail or removing complexity without losing the data needed to solve a problem
- **generalization** -- finding a general approach to a set of problems

What's next for you?

- Codewars challenges https://www.codewars.com
- Khan Academy Computer Science https://www.khanacademy.org/computing/computer-science
- Python tutorial deep dive https://docs.python.org/3/tutorial/index.html
- Algorithms, by Sedgewick, Wayne https://algs4.cs.princeton.edu/home/
- Recent 'Must-Watch' Talks https://realpython.com/must-watch-pycon-talks
- Kaggle https://www.kaggle.com/learn/overview

Shout outs to

- Grokking Algorithms: An illustrated guide for programmers and other curious people by Aditya Y. Bhargava (Manning, 2016)
- Data Structures and Algorithms in Python, free Udacity course
- Dan Bader Python blog https://dbader.org/blog/
- Think Python https://greenteapress.com/wp/think-python/
- Problem Solving with Algorithms and Data Structures https://interactivepython.org/runestone/static/pythonds/index.html

Slides at https://github.com/laeccan/Salt-Lake-Pyladies/

Understanding ‘why’
is your superpower.