Week 3 - Advanced Python

Monday

- Advanced Object-Oriented Programming
  - Creating Multiple Instances through Loops
  - Magic Methods
  - Object Inheritance
  - Method Overriding
  - Inheriting Multiple Classes
- **Exercise:**
  - **Dungeon Monsters**

Tuesday

- Advanced Data structure (extra reading)
  - Tree
  - LinkedList
- Map, Reduce, Filter, and Lambda
- Recursive Functions
- **Extra**: Generator and Iterator
- **Exercise:**
  - **Python Map and Reduce Exercise**
  - **Fibonacci Sequence**

Wednesday

- Intro to RE – Regular Expression &
  - **Demo**: Regular Expression
- In-place algorithm & Two pointers
- Merge two sorted arrays and Merge sort
- **Exercise**:
  - **Regular Expression Exercise**

Thursday

- DFS and BFS (extra reading)
- Binary Search
- Introduction to Dynamic programming/Memoization
- **Exercise**:
  - Binary Search
  - String

Friday

- **Interviews:** 10 minutes per person
- **Project:** BlackJack (OOP)
- **Weekend Challenge**:
  - Finish BlackJack (commit)

Advanced data structure

Python Collections

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set tuple.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

Reference
https://docs.python.org/3.6/library/collections.html

Most frequency used.

- deque
- Counter
- defaultdict

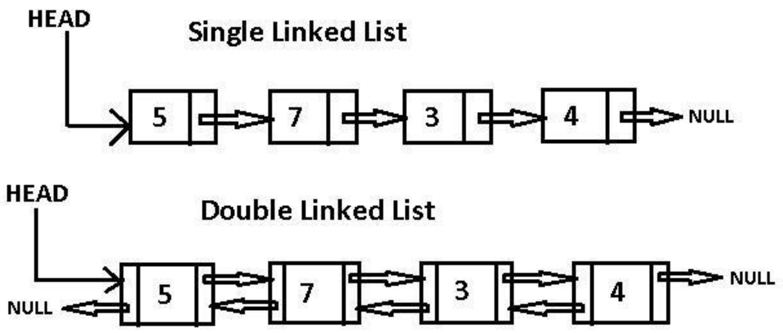
DEMOs

- Moving average Using deque
- Most common words in text using Counter
- Merge tags Using defaultdict with different data structure

LinkedList

- Compare the difference with Array List
- Single Linked List vs Double Linked List

Data Structure image



Python Code

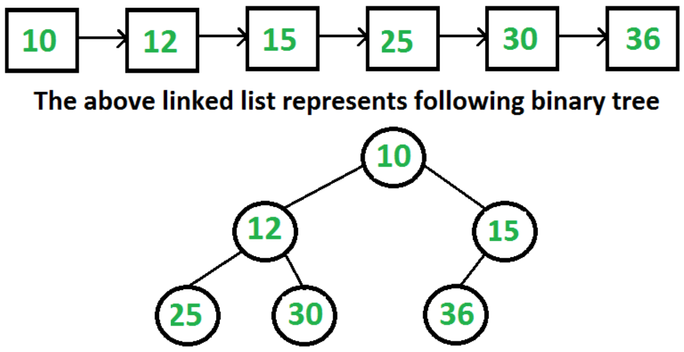
```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None
```

Array vs LinkedList and Coding Practice

- https://leetcode.com/tag/array/
- https://leetcode.com/tag/linked-list/

Tree

Data structure



Python Code

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

Coding Practice

https://leetcode.com/tag/tree/

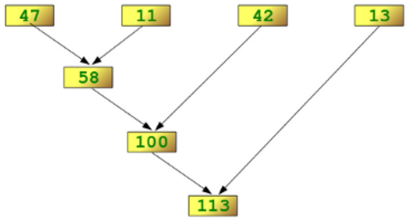
Map, Reduce, Filter, and Lambda

Map
Map applies a function to all the items in an input list.

```
>>> list(map((lambda x: x **2), items))
[1, 4, 9, 16, 25]
```

Reduce
Reduce is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list.

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```



Filter
As the name suggests, filter creates a list of elements for which a function returns true.

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
 [1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x: x % 2 == 0, fib)
 [0, 2, 8, 34]
```

Lambada
The lambda operator or lambda function is a way to create small anonymous functions (functions without a name). These functions are throw-away functions. Lambda functions are mainly used in combination with the functions filter(), map() and reduce().

```
>>> f = lambda x, y : x + y
>>> f(1,1)
2
```

Reference
- http://book.pythontips.com/en/latest/map_filter.html
- https://www.python-course.eu/lambda.php
- https://www.python-course.eu/lambda.php

Recursive

- A function/method calls itself
- Termination condition
- Problem need to become smaller and smaller

Example I - factorial

n! = n * (n-1) * (n-2) * ... * 2* 1

For example, if n =4 then 4! = 4 * 3 * 2 * 1. It can break down to…

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1

```
def factorial(n):
  if n == 1:
    return 1
  else:
    return n * factorial(n-1)
```

QUESTION: why not just use for loop.

Example II – Fibonacci numbers

The Fibonacci numbers are the numbers of the following sequence of integer values:

The Fibonacci numbers are defined as:

Fn = Fn-1 + Fn-2

when F0 = 0 and F1 = 1

It is: 0,1,1,2,3,5,8,13,21,34,55, ...

Recursive way:

```
def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)
```

For loop way:

```
def fib2(n):
  a, b = 0, 1
  for i in range(n):
    a, b = b, a + b
  return a
```

Example III – Same Tree

RE – Regular Expression

A ***regular expression*** (regex) is, in theoretical computer science and formal language theory, a sequence of characters that define a **search** pattern.
Regex is very useful in computer science and data science. Also, it is top question be asked during the interview. Learn it in Python is the best approach, and you can transfer knowledge to other languages.

Import it

```
import re
```

Regular Expressions

- a, X, 9, < -- ordinary characters just match themselves exactly.
- . (a period) -- matches any single character except newline '\n'
- \w -- matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_].
- \W -- matches any non-word character.
- \b -- matches word boundary (in between a word character and a non word character)
- \s -- matches a single whitespace character -- space, newline, return, tab
- \S -- matches any non-whitespace character.
- \t, \n, \r -- tab, newline, return
- \d -- matches any numeric digit [0-9]
- \D matches any non-numeric character.
- ^ -- matches the beginning of the string
- $ -- matches the end of the string
- \ -- escapes special character.
- (x|y|z) matches exactly one of x, y or z.
- (x) in general is a remembered group. We can get the value of what matched by using the groups() method of the object returned by re.search.
- x? matches an optional x character (in other words, it matches an x zero or one times).
- x* matches x zero or more times.
- x+ matches x one or more times.
- x{m,n} matches an x character at least m times, but not more than n times.
- ?: matches an expression but do not capture it. Non capturing group.
- ?= matches a suffix but exclude it from capture. Positive lookahead.
- a(?=b) will match the "a" in "ab", but not the "a" in "ac"
- In other words, a(?=b) matches the "a" which is followed by the string 'b', without consuming what follows the a.
- ?! matches if suffix is absent. Negative look ahead.
- a(?!b) will match the "a" in "ac", but not the "a" in "ab"
- ?<= positive look behind
- ?<! negative look behind

Methods

findall	**Return all non-overlapping matching patterns in a string as a list**

finditer	Like findall, but returns an iterator
match	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None
search	Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning
split	Break string into pieces at each occurrence of pattern
sub\ subn	Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression

Escape Codes

Code	**Meaning**
\d	a digit
\D	a non-digit
\s	whitespace (tab, space, newline, etc.)
\S	non-whitespace
\w	alphanumeric
\W	non-alphanumeric
\\	\\

Example (Email)

Simple version

```
"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
```

Full version.

```
"(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x
5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z
0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])"
```

In-place algorithm

In computer science, an in-place algorithm is an algorithm which transforms input using no auxiliary data structure. However, a small amount of extra storage space is allowed for auxiliary variables. The input is usually overwritten by the output as the algorithm executes. In-place algorithm updates input sequence only through replacement or swapping of elements.

An algorithm which is not in-place is sometimes called **not-in-place** or **out-of-place**

Reverse an array in place

```
def reverseInplace(lst):
    left,right= 0, len(lst)-1
    while left<=right:
        lst[left],lst[right]= lst[right],lst[left]
        left= left+1
        right=right-1

a=[1,2,"4",8,0]
reverseInplace(a)
print(a)
>> [0, 8, '4', 2, 1]
```

Reverse an array out place

```
def reverseOutplace(lst):
    return lst[::-1] # create a new o

a=[1,2,"4",8,0]
b = reverseOutplace(a)
print(a)
>>[1,2,"4",8,0]
print(b)
>> [0, 8, '4', 2, 1]
```

Two pointers

Using more than one pointer to point positions of a data structure. It could be left pointer or right pointer. It could be the slow or fast pointer…

Example I - Reverse an array in place

Write a function that takes an array as input and returns the array as reversed.

Example:

Given a = [1,2,4, "e"], return ["e",4,2,1]

```
def reverseList(lst):
  left,right= 0, len(lst)-1
  while left<=right:
    lst[left],lst[right]= lst[right],lst[left]
    left= left+1
    right=right-1
```

Example II - Remove Duplicates from Sorted Array in place

Given a sorted array, remove the duplicates in-place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

Example:

Given *nums* = [1,1,2],

Your function should return length = 2, with the first two elements of *nums* being 1 and 2 respectively.
It doesn't matter what you leave beyond the new length.

```
class Solution(object):
  def removeDuplicates(self, nums):
    if len(nums)<1: return 0

    j, pre = 0, None
    for i in range(len(nums)):
      #print i,nums[i],
      if nums[i]!=pre:
        pre=nums[j]=nums[i]
        j=j+1
    return j
```

Test your code at:
https://leetcode.com/problems/remove-duplicates-from-sorted-array/description/

Coding Practice

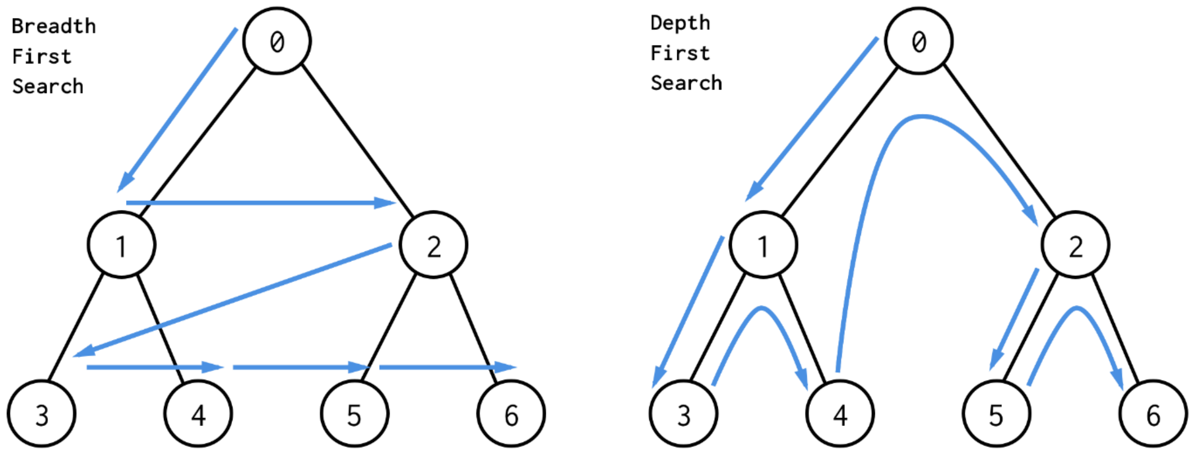
https://leetcode.com/tag/two-pointers/

DFS and BFS

Definitions

- Depth-First Search (DFS)
- Breadth-First Search (BFS)

Reference
http://mishadoff.com/blog/dfs-on-binary-tree-array/



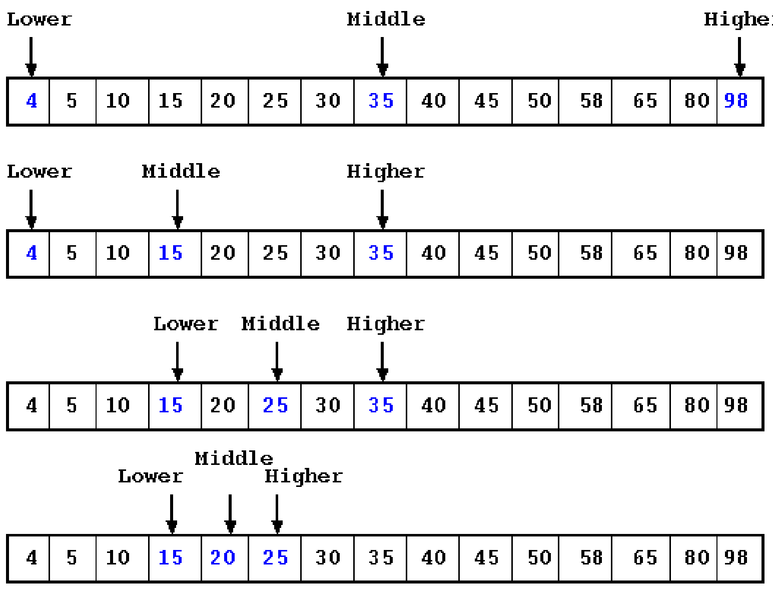
Coding Practice

- https://leetcode.com/tag/depth-first-search/
- https://leetcode.com/tag/breadth-first-search/

Binary Search

In computer science, binary search is a search algorithm that finds the position of a target value within a **sorted** array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array.

How it works



Code

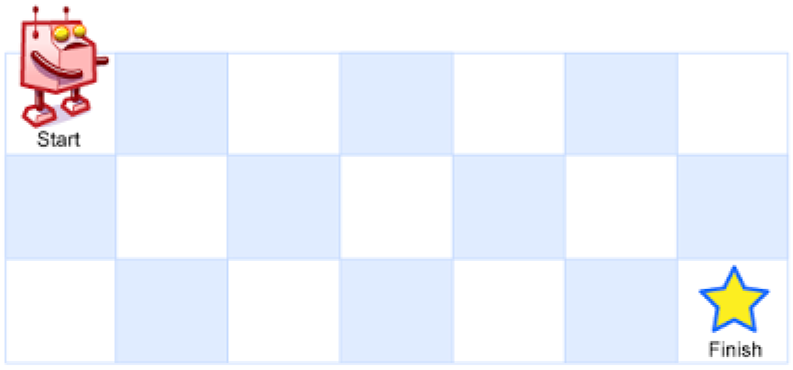
```
def binarysearch(sequence, value):
  lo, hi = 0, len(sequence) - 1
  while lo <= hi:
    mid = lo + (hi-lo) // 2
    if sequence[mid] < value:
      lo = mid + 1
    elif value < sequence[mid]:
      hi = mid - 1
    else:
      return mid
  return None
```

Coding Practice

https://leetcode.com/tag/binary-search/

Introduction to Dynamic programming

Definition
In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (DP) is a method for solving a complex problem by **breaking it down** into a collection of simpler subproblems, **solving each of those subproblems just once**, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the **previously computed solution**, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. The technique of storing solutions to subproblems instead of recomputing them is called "**memoization**".



NOT EASY TAKE TIME TO UNDERSTAND!