**DS&A topics and plan**

**Resources:**

* [Introduction to algorithms](file:////Users/cl16977/Documents/Practice_Interview_Problems/%5bALGORITHMS%5d%5bIntroduction%20to%20Algorithms.%20Third%20Edition%5d.pdf) book
* DS&A FCC YouTube [course](https://www.youtube.com/watch?v=pkYVOmU3MgA)
* [Take U Forward](https://www.youtube.com/c/takeUforward/playlists) YouTube courses: Recursion, More Recursion, Trees, Graphs, Stack and Queue, Binary Search, Greedy Algorithm, Linked Lists, 2-pointer Qs, D Qs, Sorting/Searching/Maths
* Cracking the coding interview

**Plan**:

This plan should be stuck to, and all goals SMART.

Current links to have open:

FCC DS&A course: <https://www.youtube.com/watch?v=pkYVOmU3MgA> (4.02.51)

<https://jovian.ai/aakashns/python-binary-search-trees>

<https://jovian.ai/learn/data-structures-and-algorithms-in-python/lesson/lesson-2-binary-search-trees-traversals-and-balancing>

<https://www.youtube.com/watch?v=Yt50Jfbd8Po&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=16> (TUF BST Q’s)

<https://www.youtube.com/watch?v=f-sj7I5oXEI&list=PLgUwDviBIf0q8Hkd7bK2Bpryj2xVJk8Vk&index=48> (TUF BST Q’s)

<https://www.youtube.com/watch?v=IJDJ0kBx2LM> (Recursion course FCC)

<https://www.youtube.com/watch?v=G1fRTGRxXU8&list=PLgUwDviBIf0rGlzIn_7rsaR2FQ5e6ZOL9&index=9> (TUF Recursion Course)

Leetcode Comb Sum 1-3

**Very short term TODO (so I don’t forget where I was one day to the next):**

* Do validate BST question
* **Do TUF (I’m on L8 – do it – read leetcode discussion to understand it and maybe other leetcode python vids for that solution; then make l7 patterns more pythonic once having looked at leetcode l8 answers; (read about recursion from that book? Ch1-4 probs useful (probs not 5) as an intro (I’m on 2.3) – esp for recursion 4.3, 4.4 and 4.5 – or leave it til later); then do some leetcode easy qs on recursion and backtracking (do combination sum 3 aka L9 TUF – I FINALLY DID THIS - NEED TO NOW WRITE WHAT I LEARNT (ESP ABOUT THE RES BIT AND FUNCTION SHOULD BE WITHIN THE METHOD FOR IT TO WORK (OR MAYBE IT CAN BE A DIFFERENT FUNCTION BUT NEED TO WORK OUT HOW TO GET IT TO WORK IN THAT CASE)…CLEAN IT UP IF IT NEEDS TO BE…TRY WITH APPEND/POP AND SEE IF QUICKER/LESS MEMORY USED…TRY WITH A CLEANER APPROACH – SEE IF WE CAN IMPROVE SUCH AS JUST PASSING K AND N AND DECREASING K EACH TIME AND N EACH TIME…), then look at TUF solution and discussion on leetcode; do combination sum 2) – note backtracking tag is more appropriate on leetcode; then continue with that course or do fcc course first?) then FCC recursion course? Or should I do it with DP too? Or do now anyway?**
* Do up to BFS and DFS (L4) in Take U Forward Series…
* Do other traversals (pre, in, post, level) (L13) in Take U Forward Series…
* **I got to 4.02.50 in the FCC DS&A vid – carry on from there!**
* **Tidy .py file with all options in there and add to function docstrings and also add to notes in this doc (e.g. add why we need self-balancing trees. – think about naïve insert into BST…it unbalances it; and also types of binary trees and how to check if each is as is (esp balanced!))**
* Then do TUF up to L18/19 (don’t need to do all Q’s) – done L14, L16 (kind of!)
* Carry on with FCC DS&A vid…
* Then do TUF up to L33/34 (don’t need to do all Q’s)
* Do foobar L3
* Read ch 12 BSTs in book

**Binary trees**:

* [by 2021/12/31] finish binary tree vid and qs from long vid – leaving some Q’s til I know BFS and DFS
* [by 2022/01/10] do the take u forward course on binary trees – do BFS and DFS for binary trees at least! And a few Q’s
* [by 2022/01/12] read Ch12 of Intro to Algorithms book
* [by 2022/01/15] do some extra leetcode binary tree q's
* [by 2022/01/18] Make sure I’ve covered the following problems
  + create a binary tree using tree nodes
  + do in/pre/postorder traversals - doing both iterative and recrusive - make sure we can write both from scratch…and that we really understand the iterative solutions and why they work for all 3 (postorder is hardest)
  + height/depth
  + num nodes
  + min/max depth
  + diameter
  + is the binary tree a bst
  + min and max vals too of bst
  + insert, update, list all, is balanced for bst
  + create bst from sorted list
  + balance unbalanced bst
  + do some leetcode problems!!! E.g. invert a binary tree
* [by 2022/01/19] do a summary of Binary trees in a notes document (create this doc) for revision later
* [by 2022/01/22] do **Q3 of foobar** challenge
* (do **Q4 of foobar** challenge)
* (do **Q5 of foobar** challenge)

**Recursion:**

* [by 2022/01/27] do Take U forward YouTube video
* [by 2022/01/31] do FCC DP & Recursion YouTube video
* do ?

**Dynamic Programming:**

* do <https://www.youtube.com/watch?v=oBt53YbR9Kk> – FCC course on recursion and DP
* do ?
* do ?

**Graphs:**

* do ?
* do ?
* do ?

**Sorting:**

* do ?
* do ?
* do ?
* Probably know more or less: insertion sort/quicksort/merge sort (potentially be able to code them – they’re easy enough) and know more or less the other main ones too (not how to code, but concept and run times and positives and negatives)

**Hash Tables:**

* do ?
* do ?
* do ?

**Linked Lists:**

* do ?
* do ?
* do ?

**Divide and Conquer:**

* do ?
* do ?
* do ?

**Greedy algorithms:**

* do ?
* do ?
* do ?

**Stacks and Queues:**

* do ?
* do ?
* do ?

**Heaps:**

* do ?
* do ?
* do ?

**Other things – bits/operating systems etc?:**

* do ?
* do ?
* do ?

**??:**

* do ?
* do ?
* do ?

**Direct Interview Prep:**

* do all qs from here: <https://takeuforward.org/interviews/strivers-sde-sheet-top-coding-interview-problems/>
* leetcode q’s and recommended ones from other sites (watch vids/read articles about it at this point)
* read cracking the coding interview
* look at example interviews online e.g. <https://www.youtube.com/watch?v=2dxE83jwGEI>

**Notes:**

**Binary Search Trees:**

* two conditions:
  + the left subtree of any node only contains nodes with keys less than the node's key
  + the right subtree of any node only contains nodes with keys greater than the node's key
* DFS – pre-(root, left, right), in-(left, root, right), and post-order (left, right, root) – time O(n = num of nodes); space O(h = height) which if not balanced is O(n = num\_nodes)
* BFS – aka level-order – uses a Queue
* Inorder traversal returns sorted list
* Can be searched in O(logn)

**Recursion:**

**TUF course** <https://www.youtube.com/watch?v=yVdKa8dnKiE&list=PLgUwDviBIf0rGlzIn_7rsaR2FQ5e6ZOL9> notes at **~/Documents/python\_courses/Recursion**

* Recursion is when a fn calls itself *until* a specific condition is met (base case(s)/condition(s) – i.e. where there is no more recursion) – if no condition met/specified then stack overflow/infinite recursion occurs
* Instead of writing each function call and where they wait in the stack while a function within it is being called, you can use a recursion tree to show what is happening
* When doing memory complexity – it is the stack space used that you need to find the order of
* Backtracking- when you return from the base case upwards. See l2.py for example
* *Parametrised* (the parameter is doing the work and the updated answer is passed to parameter each new recursive fn call) vs *Functional* (you just need to recall the function with a smaller problem each time until reaching base case and the function takes care of things) Recursion – see l3.py for example
* Multiple recursion causes a tree like structure to be formed – so O(2\*\*N) order (if two recursion calls each time) unless memo used then linear i.e. O(N) – see l5.py (except the memo bit – I added that!). And of course if multiple recursion calls, the first call has to finish before the next one etc (I know this, it’s obvious)
* Subsequences of e.g. arrays do not have to be contiguous (like sub-arrays), but the elements in them do need to be in the same order as they appear in the original array - l6.py
* L7 – this shows various patterns, extending what we saw in the l6 about how to get subsequences of an array, the pseudocode for which was:

NOTE: FOR ALL THE BELOW, IT IS DONE WITH C++ IN MIND – INSTEAD OF THE .POP LINE, YOU CAN JUST PASS A COPY OF THE ARRAY TO THE RECURSIVE CALL AND THEN YOU DON’T NEED IT.

(OTHER THINGS TO REMEMBER: PYTHON PASSES BY OBJECT REFERENCE; SLICING AN ARRAY RETURNS A COPY; …)

def fn(arr, ind=0, subsequence=None):

"""

This is what he does in video

"""

if subsequence is None:

subsequence = []

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

print(subsequence)

return

subsequence.append(arr[ind])

fn(arr, ind+1, subsequence) # option 1: take element

subsequence.remove(arr[ind]) # or subsequence.pop()

fn(arr, ind+1, subsequence) # option 2: don't take element

then you can also add a condition to the base case to just print subsequences that are == k (or <= k or >= k or whatever you want):

def fn(arr, k, ind=0, subsequence=None):

"""

print all sub-sequences whose sum is k

TIME: O(2\*\*N) - think of recursion tree and extra N factor due to sum each time

SPACE: O(N) - depth of recursion tree - stack only ever has a max of N calls on it

JUST PRINTS EACH SUBSEQUENCE WHEN IT FINDS IT

"""

if subsequence is None:

subsequence = []

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

if sum(subsequence) == k:

print(subsequence)

return

subsequence.append(arr[ind])

fn(arr, k, ind+1, subsequence) # option 1: take element

subsequence.remove(arr[ind]) # or subsequence.pop()

fn(arr, k, ind+1, subsequence) # option 2: don't take element

then you can just improve the above code by passing a running sum to optimise it

def fn(arr, k, running\_sum=0, ind=0, subsequence=None):

"""

print all sub-sequences whose sum is k

TIME: O(2\*\*N) - think of recursion tree

SPACE: O(N) - depth of recursion tree - stack only ever has a max of N calls on it

JUST PRINTS EACH SUBSEQUENCE WHEN IT FINDS IT

NOW WITH RUNNING SUM OPTIMISATION

"""

if subsequence is None:

subsequence = []

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

if running\_sum == k:

print(subsequence)

return

subsequence.append(arr[ind])

fn(arr, k, running\_sum+arr[ind], ind+1, subsequence) # option 1: take element

subsequence.remove(arr[ind]) # or subsequence.pop()

fn(arr, k, running\_sum, ind+1, subsequence) # option 2: don't take element

then you can also return early in the case that you want to just return the first subsequence where sum of subsequence == k, by returning a Boolean to signify whether we have found it or not

def fn(arr, k, running\_sum=0, ind=0, subsequence=None):

"""

print one sub-sequence (if one exists) whose sum is k

TIME: O(2\*\*N) - think of recursion tree

SPACE: O(N) - depth of recursion tree - stack only ever has a max of N calls on it

NOW JUST PRINTS FIRST SUBSEQUENCE IT FINDS!

USES A NON-HACKY WAY - GOOD PRACTICE!

"""

if subsequence is None:

subsequence = []

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

if running\_sum == k:

print(subsequence)

return True

return False

subsequence.append(arr[ind])

if fn(arr, k, running\_sum+arr[ind], ind+1, subsequence): # option 1: take element

return True

subsequence.remove(arr[ind]) # or subsequence.pop()

if fn(arr, k, running\_sum, ind+1, subsequence): # option 2: don't take element

return True

return False

then you can also return the count of subsequences that ==k (24.30 in vid) as follows:

def fn(arr, k, running\_sum=0, ind=0, subsequence=None, l=0, r=0):

"""

count all sub-sequences whose sum is k

TIME: O(2\*\*N) - think of recursion tree

SPACE: O(N) - depth of recursion tree - stack only ever has a max of N calls on it

VERY SIMILAR TO ABOVE, BUT ADAPTED TO BE COUNT, NOT RETURN FIRST ONE THAT EQUALS K

"""

if subsequence is None:

subsequence = []

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

if running\_sum == k:

print(subsequence)

return 1

return 0

subsequence.append(arr[ind])

l = fn(arr, k, running\_sum+arr[ind], ind+1, subsequence, l, r) # option 1: take element

subsequence.remove(arr[ind]) # or subsequence.pop()

r = fn(arr, k, running\_sum, ind+1, subsequence, l, r) # option 2: don't take element

return l + r

and can also remove the subsequence printing if we want and therefore all references to subsequence if all we care about is the running sum

def fn(arr, k, running\_sum=0, ind=0, l=0, r=0):

"""

count all sub-sequences whose sum is k

TIME: O(2\*\*N) - think of recursion tree

SPACE: O(N) - depth of recursion tree - stack only ever has a max of N calls on it

VERY SIMILAR TO ABOVE, BUT ADAPTED TO BE COUNT, NOT RETURN FIRST ONE THAT EQUALS K

NOTE: I HAVE NOW ALSO REMOVED THE SUBSEQUENCE HERE TOO

NOTE: I COULD ALSO ADD ANOTHER BASE CASE TO RETURN EARLY IF WE GO ABOVE THE

K - THIS ONLY WORKS IF ALL NUMBERS IN ARR ARE POSITIVE, AND WE'D JUST ADD:

if running\_sum > k:

return 0

AS A BASE CASE

"""

# print(f's={subsequence}, {id(subsequence)}')

if ind >= len(arr):

if running\_sum == k:

return 1

return 0

l = fn(arr, k, running\_sum+arr[ind], ind+1, l, r) # option 1: take element

r = fn(arr, k, running\_sum, ind+1, l, r) # option 2: don't take element

return l + r

Note: when writing some recursive functions, it is v important to always be aware that python passes args/params by object.

Note: I should remember the above patterns of by heart, they are summarised here:

**Print all** (with a specific condition)

def fn():

if base condition is met

if the specific condition is met:

do\_something – e.g. print the subsequence

return

fn(option\_1\_args)

fn(option\_2\_args)

return

**Print one** (with a specific condition)

def fn():

if base condition is met

if the specific condition is met:

return true

return false

if fn(option\_1\_args):

return True

if fn(option\_2\_args):

return True

return False

**Print the count** of subsequences (with a specific condition)

def fn():

if base condition is met

if the specific condition is met:

return 1

return 0

l = fn(option\_1\_args)

r = fn(option\_2\_args)

return l + r

* L8.py – a leetcode problem

**FCC course** <https://www.youtube.com/watch?v=IJDJ0kBx2LM>

* ?
* ?