Data Structures & Algorithms Cheat Sheet

Problem Solving Tips

How to solve any problems

1. Think about problem in real life
   a. Read question, digest and then think about it in real life
   b. Rewrite question to make it easier to understand
   c. Think about a solution to a problem, any solution!
      i. Try to optimize it after
2. Rubberducking
   a. Trace through code top to bottom, explain top to bottom and then hear yourself solve the problem
   b. Good way to ask for help indirectly
3. Have an emergency plan
   a. Have a systematic way to go through all the data structures, algorithms and concepts
      i. Create an acronym to help me remember

Patterns for breaking down patterns you haven't seen before - use BUG

- **B**rute force
- **U**se Hash tables
- **G**reedy

General Tips

- Read the problem description & understand problem before jumping in
- Working backwards from a target runtime. At each runtime, visit what could be possible algorithms for that runtime to solve the problem
- Ask about size of inputs since this can affect need for optimization
  - array with length 10 does not necessarily need extra time to optimize vs array with length 1000000
- Check for side effects if doing in-place modifications to any data structure, i.e. arrays, linked lists, etc.
- Check for code that can be refactored, i.e. break code into small pieces and see if they can be reused
- Discuss multiple possible solutions and tradeoffs for them before moving into actual solution

- Be mindful about names of functions and variables! Be sure you are using the correct ones
- Break problems down into small chunks and solve each chunk individually w/o worrying about the other chunks -- assume it'll be correct
- Try to do multiple examples
- Can storing variables be substituted for data structure? Can you just store a couple variables instead of a whole list of items?
- Modularized your code

Python Shortcuts

- Use float('inf') for arbitrarily large/small number
- For heaps:
    - `import heapq`
    - `# create a list, let's call it test_list`
    - `# min heap operations:`
        - `heapq.heapify(test_list)`
        - `heapq.heappop(test_list)`
    - `# max heap operations:`
        - `heapq._heapify_max(test_list)`
        - `heapq._heappop_max(test_list)`
- Arrays:
    - To visit a loop in reverse:
        - for i in range(list_length - 1, -1, -1)
            - range(start, end, increment)
    - To get both index and item at index, use enumerate
        - for idx, val in enumerate(some_list)
            - idx will return the index
            - val will return the element at that index
            - can add a "start = x" to tell where the list should start

Optimization

- Look for repeated work
    - can it be optimized by using %?
    - can you solve it by using greedy?
- Look at hints given in the problem
    - are the inputs limited?
- Look at data structures that you could use
- Look for bounded runtime
    - see if you could try to narrow down the solution by bounding the runtime

Hints

- Look at what the inputs are being described as
    - Is the range of inputs limited?
    - Is the input a complete tree?

General Programming

- Think up and draw out all possible cases (edge cases)
- Use very specific and descriptive variable names
- If too many variables to keep track of, create an object to help make the code more readable
- To save for space, look for:
    - hints in regards to range of the inputs --> can you create a data structure (maybe array) to help optimize for space?
- To save for time, think about:
    - just-in-time vs ahead-of-time calculations - just-in-time is typically good for calculations that does not need to be called often whereas ahead-of-time is good for calculations that will be called many times
- Short-circuiting
    - could it help reduce runtime by not having to perform another check?
    - could help with having to check arrays index out of bound if place another statement that could short-circuit before checking for an index out of bound in array

Big O

Strings

- For strings, the space complexity could be O(k) where k is the number of characters in ASCII or Unicode and thus is constant

Queues

- when looking at runtime analysis for queues operation, the trick is to think of it in terms of cost per item passing through the queue rather than cost per enqueue() or dequeue() operations

Linked List

- "one pass" isn't always fewer steps than "two passes." Always ask yourself, "Have I actually changed the number of steps?"

Greedy

- typically fast and good way to break down a problem
- ask yourself is it possible to go through one pass by simply updating the 'best answer so far' as we went?
- What additional values would we need to keep updated as we looked at each item in our input in order to update the 'best answer so far' in constant time?
- Don't forget to check for double negatives as it could turn into positive, discuss this assumption
- could the greedy algorithm be used twice instead of once?

Strings

- Inputs
    - Ask if string is an ASCII or unicode string
        - Unicode is a superset of ASCII
        - ASCII maps only to 128 numbers - uses 7 bits
        - Unicode maps to 2^21 numbers
            - Needs to be encoded, i.e. UTF-8, UTF-16, UTF-32
- Could you use tries?
- Use built-in string functions to help when you can
    - string.lower(), string.upper(), string.capitalize(), char.isalpha()

Arrays & Hash Tables

- Check the indices to make sure you're not out of range
- for 2D matrix, try to solve each element within the row instead of approaching the whole
- Could sorting help?
- Could save time by using a dictionary?
    - Could a set be less complicated than a dictionary? If don't need to store the values of the key, set may be a better answer

Linked List

- Look out for side effects since each node typically has a pointer, when modifying in-place, this could cause side effects such as "dangling" node
- Could using the slow and fast runner help solve the problem?

- If stuck, draw out the linked list and perform step by step
- if reversing a linked list, remember to create a prev_node and next_node prior to loop

Trees & Graphs

- Ask if it is a binary search tree, balanced or unbalanced
- Ask if values in the tree are unique
- When you meet a tree problem, ask yourself two questions: Can you determine some parameters to help the node know its answer? Can you use these parameters and the value of the node itself to determine what should be the parameters passed to its children? If the answers are both yes, try to solve this problem using a "top-down" recursive solution.
- Or, you can think of the problem in this way: for a node in a tree, if you know the answer of its children, can you calculate the answer of that node? If the answer is yes, solving the problem recursively using a bottom up approach might be a good idea.

Tree Solving Approach
- Simplify the question if needed
- Decide if the problem be solved using BFS or DFS with modification
- Make a case for which one to choose
- Add in restrictions needed to solve the problem

Tree Edge cases
- Linked list trees (i.e. trees with only left or right children)
- Tree with only one node
- Root is none

DFS - O(n) runtime
- Check if node has been visited in implementation to avoid infinite loop
- Uses stack w/ iterative approach
- Recursive or use iterative approach if cannot solve using recursion
- Can help with short circuiting

BFS - O(n) runtime
- Uses queue
- Don’t use recursion

Graphs

Representation	**Size**	**Pros**	**Cons**
Edge List	O(E)	- simple to implement - typically the input	- linear search to find if an edge exists (can be tweaked to be O(lg E) but tricky)
Adjacency Matrix	O(V^2)	- constant lookup time O(1)	- space is expensive O(V^2) - takes O(V) time to look up adjacent vertices
Adjancency List	O(V + E)	- constant time to obtain adjacency list O(d), d = vertex degree - less space consuming	

- template to solving a graphing problem: (most, not always, remember CHECK reqs!!)
    - create an adjacency list/matrix if you are not given one as an input
    - initialize some kind of queue/stack to store the nodes to visit and initialize a visited array to keep track of nodes already visited so that you don't go in a cycle
        - sometimes, in lieu of the visited array, you could also "sink" an island
    - loop until the queue/stack is empty
        - pop an element from the queue/stack
        - add popped element to visited
        - do some operations with the element for what you're trying to find
        - loop through the neighbors of that element
            - check if it's already visited & do some operation for what you're trying to find and then add that neighbor to the queue
- general graph edge cases are :
    - node w/ no edges - typically an issue for graph traversal
    - cycles - typically an issue for graph traversal causing an infinite loop
    - loops - detect loops by checking if the node is in its own set of neighbors
    - does node exist in graph?
    - if finding route, what if route doesn't exists?
- for time complexity, try to think of things in terms of total number of edges whenever you can
- techniques: modified BFS, DFS and/or backtracking
    - Backtracking --> break the problem up into small pieces and think in terms of choice, constraints and goal
        - Think about how to solve each cell?
- special graph algorithms
    - topological sort - good for putting the vertices in a special order based on incoming edges --> think indegrees

    - Dijkstra's algorithm - finding shortest path for positive weighted graphs (directed & undirected) --> think priority queue (min heap) and map data structure
    - A* algorithm

Searching & Sorting

- Draw out what you're trying to sort/search and then reason it out before jumping into the the actual algorithm
- If binary tree is involved, think about if BFS or DFS could help solve the problem
    - if implementing a solution using recursion, be sure to talk about possible stack overflows
- If the input is a sorted list, think about possibly using a binary search tree
    - can you use a modified version of binary search, BFS or DFS to solve the problem?
        - remember that DFS is typically recursive and BFS is typically iterative
- Step through the brute force first and then try to optimize from there
- If runtime of log n is involved, try to see if a modified approach of a binary search will help solve the problem
- Edge case - check if input is a possible edge case
- If runtime is O(n^2), can it be improved by sorting to O(nlogn)?
- Discuss sorting in-place tradeoffs -- save space, but could have weird side effects
- Are inputs within a limited range?
    - If bounded --> try bucket sort / radix sort
    - If less than 3 keys to keep track of --> try counting sort

Recursion

- Start with a small base case, then build up from there
- Does memoization help reduce runtime complexity?
- Typically, the space complexity is the height of the tree
- Possible stack overflow using recursion b/c of the call stack
- Backtracking template:

```
def backtrack(candidate):
    if find_solution(candidate):
        output(candidate)
        return

    # iterate all possible candidates.
    for next_candidate in list_of_candidates:
        if is_valid(next_candidate):
            # try this partial candidate solution
            place(next_candidate)
            # given the candidate, explore further.
```

```
                backtrack(next_candidate)
                # backtrack
                remove(next_candidate)
```

Dynamic Programming

- Start with top-down approach and then work to bottom-up approach
  - top-down (recursive): build something from the top down --> memoization
    - typically requires checking a list for solution to a problem
  - bottom-up (iterative): build something from the bottom up --> tabulation
    - typically end up returning dp[input_value]
- Use the FAST technique:
  - **F**irst Solution
  - **A**nalyze first solution
    - does it have an optimal substructure?
    - are there overlapping subproblems? (might need to try medium-sized examples to see pattern)
  - Identify **S**ubproblems
  - **T**urn the solution around

Bits

- Remember to use XOR if you want to find the difference of bits between numbers
- Use XOR for:
  - "cancel out" numbers
  - find difference of bits between numbers
- Use left shift to multiply by 2 and right shift to divide by 2
- 0xaaaaaaa = 10101010101010
- 0x555555 = 01010101010101
- Logical right shift ensures that the shifted bit added on the left side is 0
  - i.e. 1001 >>> 2 = 0010

Mistakes

- Forgetting to modularize the code
- Not considering big cases, i.e. in implementing a minimum function for stack, did not consider creating a second stack to store minimum values
- Not talking about trade offs enough

- Over optimizing too early
- Start coding too soon
- Fluster when getting nervous due to time constraints and pressure
- Forgetting about edge cases!! Always remember to check:
    - length of inputs

Emergency Plan

- Simplify, solve, adapt --> remember to break down a problem and draw examples
    - Split the problem into small solvable chunks, then combine and optimize at the end

Interview Items List

- Dry-erase markers and eraser in ziplock bag
- Extra battery for phone
- Pen & notebook

Google Doc Tips

Set google docs up before phone screen:

1. Uncheck everything under Tools > Preferences
2. Change font to Consolas
3. Use 2 spaces instead of tabs

https://www.codefellows.org/blog/setting-up-google-docs-for-technical-interview-happiness/

Resources

Learn:

- https://www.coursera.org/learn/learning-how-to-learn/home/welcome

Review:

- General Courses:
  - https://www.educative.io/courses/grokking-the-coding-interview
  - https://www.educative.io/courses/grokking-the-system-design-interview
  - https://www.educative.io/courses/grokking-the-object-oriented-design-interview
  - https://www.interviewcake.com/
- Dynamic Programming
  - https://leetcode.com/discuss/general-discussion/458695/Dynamic-Programming-Patterns

Practice:

- interviewing.io
- leetcode.com
- pramp