assessment prep notes!

▼ Arrays

Important Techniques:

1. **Sliding Window: O(n), O(1)**
    a. Use two indices to slide over data struct to explore subsets
    b. Useful when you need to find a specific subset that matches a condition
    c. Problems like: Minimum Size Subarray Sum
2. **Two Pointer: O(n), O(1)**
    a. Use two pointers in order to move towards/away/in tandem
    b. Useful when trying to find pairs of elements, compare elements, or minimize/maximize a value based on pairs of elements
    c. Pointer at each end: if smaller than target, move left in | if larger than target, move right in
    d. Problems like: Valid Palindrome

Strings

Important Techniques:

1. **Sliding Window: O(n), O(1)**
   a. Use two indices to slide over data struct to explore subsets
   b. Useful when you need to find a specific subset that matches a condition
   c. Problems like: Substring Problems
2. **Two Pointer: O(n), O(1)**
   a. Use two pointers in order to move towards/away/in tandem
   b. Useful when trying to find pairs of elements, compare elements
   c. Problems like: Valid Palindrome

Linked Lists

Important Problems:

1. **Reverse a Linked List O(n), O(1)**
    a. Base case: two nodes
    b. List

```
curr = head # head node prev = None while curr: tmp = curr # keep head aside curr.next = prev # head points to node before it prev = curr # prev for the next node is the current head curr = tmp # move on to next return prev
```

2. **Linked List Cycle**
    a. Use a visited set if space is not a constraint
    b. If optimizing for space:
        i. use two pointer to do tortoise and hare
            1. one pointer moves one node per iteration
            2. second pointer moves two nodes per iteration
            3. once pointer two hits None, return False

Recursion

Important Ideas:

1. Look for ways to repeat work
2. Helper functions are really key

HashMap/Dictionary/Set

Important Problems:

1. **TwoSum**
    a. Iterate through nums, adding to a set
    b. Before adding, check if complement is in the set: if yes, return true
    c. if no matches, return false
    d. **Can be done with dictionary if indices needed/duplicates exist**

Matrix

Generation Methods:

1. Naive method: `matrix = [[0] * num_cols] * num_rows`
2. List Comprehension: `matrix = [[0 for col in range(cols)] for row in range (rows)]`

Important Questions:

1. Sudoku Solver
2. Contains Duplicate

Searching and Sorting

Key Algorithms

1. **Binary Search O(log *n*)**
   a. Repeatedly divide a search interval in half until the item is found
   b. Problems: Merge Sorted Arrays
2. **Merge Sort**
   a. Divide and Conquer based
   b. combines two already sorted arrays into one sorted array
   c. Can be used on one array - split into two arrays and split down until sorted

Stacks

Important Things To Know:

1. **LIFO Structure:**
   a. Elements are accessed in reverse order of arrival
   b. [].push(), [].pop()
2. Problems to Know
   a. Valid Parentheses
      i. Push on opening parens
      ii. When seeing closing parens, pop
         1. if not correct type, return false
      iii. return true when stack empty
3. **Monotonic Stack**
   a. Designed to process sequences while maintaining order, non increasing/decreasing
   b. Effective for solving problems where you need to find the next greater or next smallest element in an array

Trees

Important Techniques

1. **Traversals**
    a. In-Order - traverse left-root-right
    b. Pre-Order - traverse root-left-right
    c. Post-Order - traverse left-right-root
2. **Searches**
    a. DFS - search as far along a path before going elsewhere, good for checking path existence
    b. BFS - explore all nodes at existing depth, good for shortest path to somewhere or minimum depth in a tree
        i. Can be implemented with a Queue - FIFO structure where you:
            1. add root to queue (in python, queue = deque())
            2. `queue.popleft()` then add nodes from left to right to queue as well
            3. continue to process till all seen!

Graphs

Important Techniques

1. **BFS**
    a. Done with a queue, similar to above
    b. Useful for minimum distance problems
2. **DFS**
    a. Implemented using recursion - use a visited set to track cycles
    b. useful when solving puzzles or navigating mazes
    c. only visits each node once - more efficient
3. **Topological Sort**
    a. Useful when exploring DAGs (directed acyclic graphs)
    b. For every edge from vertex u to vertex v, vertex u comes **before** vertex v in the ordering
    c. Important when doing scheduling and dependency resolution problems
    d. Implementation: using DFS:
        i. Pick univisited node (usually 0) and do dfs on that node
        ii. Going forward, only explore **unvisited nodes**
        iii. after visiting all child nodes, add parent node to front of the list
        iv. Continue process till all nodes visited!
4. **Djikstra's Algorithm**
    a. Weighted Graphs with **positive** weights
    b. Goal: find shortest, cheapest path from one node to another
    c. Implementation: Uses Priority Queue
        i. mark every node as unvisited
        ii. add initial node to prio queue, store nodes sorted by cost
        iii. process nodes in order of seeing them (BFS)
        iv. keep track of cheapest way to get to each node that you run into

▾ Heaps