Two minute drill

Disclaimer: these are notes scribbled down over the course of a few weeks, scrambling to refresh myself prior to interviews. Definitely not 100% accurate, just general thoughts, notes, and my approach to certain types of problems. **Not meant as a cheatsheet, but an example of what helped drill some things into my brain, might be useful for others to create their own documents for themselves**

- **General things to remember**
  - Check edge cases, e.g. bad input, empty input (at least talk about them, incorporate into test cases)
  - Check for off-by-one problems, i.e. for loops/iteration, counting
  - Walk through code with example(s) afterwards
  - Keep runtime in mind throughout the process, maybe jot notes down while working through problems
- **Subsets**
  - When generating a subset, we have a "choice" of whether a specific character is involved or not, giving us **2^n** subsets
    - If we have a binary choice of yes/no, then **2^n** is appropriate
  - **A substring is contiguous. A subset is not**
  - Consider recursion. Base case and build, start with building subsets of 1 element, remembering that an **empty array is a valid subset**, build from there. Keep old results, but append a new element to each old element to produce new ones, ex:
    - [A] = [ ], [A].
      [A,B] = [ ], [A], [B], [AB]
      [A,B,C] = [], [A], [B], [AB], [C], [AC], [BC], [ABC]
- **Permutations**
  - Consider recursion. Base case and build, start with a permutation of one character, and build from there
    - We can generate permutations by thinking of it as **inserting the new character at every possible position of our previous result**, ex:
      [A] = [A]
      [A,B] = [A,B] [B,A]
      [A,B,C] = [A,B,C] [A,C,B] [C,A,B] [C,B,A] [B,C,A] [B,A,C]
  - Consider that we're picking from n choices for the first spot, and then n-1 choices for the next spot, etc. We then need to accommodate for the fact that each permutation has a length of n, runtime can be considered along the lines of a factorial?
- **Substrings**
  - Brute force generation of all substrings can be done in **O(n^2)** with 2 for loops

- Consider a “sliding window” technique of evaluating substrings - this can be done in a problem such as “Longest Substring w/o duplicate chars” by keeping track of current characters, and adding/comparing the next character
    - **A substring is contiguous. A subset is not**
- **Palindromes**
    - The obvious way to determine a palindrome is comparing end and start and incrementing/decrementing pointers as you go
    - A more efficient method is to consider whether str[left] and str[right] are the same character. If they are, and str[left + 1] and str[right - 1] is a palindrome, then it is a palindrome. Can apply this recursively
- **Linked Lists**
    - On average - search/access is O(n), since we need to traverse down the LL
    - On average - insertion/deletion is O(1), since we simply add a node to the end of the LL
    - Make the clarification between singly linked and doubly linked
    - Reversing a linked list can be done with a prev, curr, and next pointer. Think things through one node at a time, and draw an example to help
    - Finding the middle node of a linked list can be done with a fast and slow pointer, e.g. rabbit/hare
- **Searching**
    - For problems where the order of the collection is not integral to solving it, consider sorting **O(n log n)** and running binary search **O(log n)**
        - This is bounded by sorting, so may be a bottleneck later on when optimizing, but typically works for a brute force solution
    - Binary search works by splitting the search area in half, picking a middle point and comparing the target against the middle point. If the midPoint is greater than the value, search left. If the midPoint is less than the value, search right
- **Sorting**
    - Many different variations, the “best” have average runtimes of **O(n log n)**
        - Quicksort
            - Done in-place, we choose an element to be the pivot value, and sort items less than the pivot to the left, and items greater than to the right
            - Recursively continues on subarrays
        - Mergesort
            - Accomplished with a temp array O(n), also a recursive solution that splits the problem space in half, sorting the left half, sorting the right half, then merging together
            - Merge step handles most of the work, uses two pointers looking at the left and right halves, taking the smaller value and putting it into the results array
            - Be careful to remember to add the rest of the elements, once one of the arrays runs out of elements

        - Is a **stable sort**. Ensures elements are always in the same relative order
    - Bubble sort
        - Can be terrible with a runtime of **O(n^2)** in average case
        - Works by comparing two numbers and swapping places if one is greater than the other. Continually does this until no more swaps are required
        - "Bubbles" up elements on repeated iterations
- **Array range analysis**
    - Consider precomputation/calculate values. This can be expanded to multi dimensional arrays, e.g. matrices
    - Basically, DP and recursion
- **Recursion/DP**
    - When considering problems of max/min size, or counting the number of ways… think recursion with memoization, or consider bottom up DP
        - E.g. robot in a grid. To find the number of ways to matrix[row][col], we can generate this by finding how many ways to the adjacent squares. E.g. reaching the point of [x][y] as summing [x-1][y] + [x][y-1], since the robot can only move right or down
        - Finding the maximum path to a square is similar, find the max of the adjacent squares and use that
    - Recursion has the overhead of space used in the call stack. We can consider bottom up DP to get around this
    - Given a recursive problem, there are (typically) a polynomial number of combinations of subproblems. Memoization and DP can reduce the runtime from a massive exponential/factorial problem to polynomial
    - Useful to think of as solving a problem using the optimal solution of a subproblem
        - Ex: from A -> C, if paths go through B, then the optimal solution for B => C must be used. Similarly, if the knapsack problem is solved without using item N, then it must be an optimal solution for a problem without item N
    - Divide and conquer! Always consider what happens if we chop the problem space in half ala mergesort
    - **Recursion follows the tree call path of DFS, think of drawing that when considering using a recursive method to estimate running time**
    - Steps to take to solve a DP problem
        - Identify recurrence relation
        - Identify that the number of parameters is bounded by polynomial (e.g. there are N*M different pair combinations between two strings)
        - Specify an order of recurrence such that partial results exist when we need it, i.e. make sure n-1 and n-2 exist if we're using it to generate n

  - When considering combinatoric problems, think in terms of reducing the number of choices after making a decision, e.g. factorial, there are n-1 fewer choices after the first decision, and n-2 fewer choices after the next, etc.
- **Matrices**
  - When traversing through a matrix, consider “flagging” visited squares
    - Ex: Game of Life, for changes that will occur on the next turn, assign a specific number, e.g. 3 == will die, 4 == rebirth
    - Ex: Counting Islands, DFS traverse through the matrix, setting any islands to water
  - Draw an example and walk through it, refer to coordinates as we write code
  - Consider traversal by layer, e.g. outer layer, then inner layer(s)
- **Graphs**
  - When traversing, remember we may need to check for cycles. Do this with an unvisited state, a finished state, and an intermediate state
    - This is because we set the state to visiting when we first visit a node, then as we unwind out of recursion, we finish the visit. If we continually traverse and start seeing the state of visiting, we know we’re trapped in a cycle
  - Consider DFS when looking for any path between two nodes
    - DFS is typically done recursively
  - Consider BFS when looking for shortest path between two nodes
    - BFS can be done with a queue
- **Trees**
  - Basic traversal:
    - In order - visit the nodes in order of values (if BST), visiting the left child, root, then right child
    - Pre order - visit the root, then left, then right. Root is always first node
    - Post order - visit left, visit right, visit root. Root is always last node
  - We can generate a tree using a combination of pre and inorder traversals
    - Preorder will tell us which nodes are the roots
    - Inorder will tell us which side children fall
  - BST validation
    - We know that left <= root < right, and this applies for all children
    - To validate a BST, we need to ensure that all children adhere to this rule, so we can set a min/max to check as we traverse downwards
- **Heaps**
  - A DS that keeps things sorted. Min heap keeps the smallest at the top, and larger values at the leaves
  - Removing an item from a heap involves swapping the root with a leaf, and then fixing the tree by bubbling down, swapping elements until the heap property is restored
  - Similarly, adding an item is done by inserting a node at the leaf level, and bubbling up until the heap property is fixed

    - We can use a min and max heap in conjunction to be clever, when calculating a median in a dynamic situation
- **Bit Manipulation**
    - Two's complement - flip all bits and add one (remembering we need a signficant digit!). This is useful in subtraction, take the two's comp. of one number and add
    - We can check if the nth bit is set by shifting n - 1 times, and ANDing with 1. & is only true if both are 1, so we know based on the result