Leetcode cheatsheet

Procedure

1. Ask questions (edge cases, input sorted, negative values, length, memory considerations, is string comparison case sensitive, etc.).
2. Clarify you've understood the problem well (what's the input and what's being asked).
3. Draw the problem.
4. Come up with test cases that ensure 100% your solution works.
    a. **NOTE:** This is crucial, because if you miss important test cases you may code an approach that ends up not working and being rejected by the coding platform.

5. Solve the problem in these steps:
    a. How does your brain do it? This sometimes gives you a good method.
    b. Brute force first, since sometimes you can optimize from there. Check Optimizations section.
    c. Exploit the logic of the problem (p.e. Container with most water problem).

6. Think of at least 3 different ways to solve it. Try to check if you can use any of the approaches mentioned below.
    a. **NOTE:** Avoid being silent for too long and saying "I don't know" without following up with a potential solution. You're showing you can reason and find a way.
    b. **NOTE:** Start **without** premature optimization and optimize from there. Because sometimes an extra O(n) operation is very fast and simplifies the thinking a lot.
    c. **NOTE:** If your gut tells you it's not going to work, try first finding another way.

7. Code the edge cases guards (undefined input, etc.).
    a. **NOTE:** Be extra sure that your code covers all cases! Don't fail the test because of a < instead of the correct <= in an if clause for example.
8. Decide on a solution and code it, while talking about it.

Approaches

While or for loop: O(n)

- When dealing with more than one pointer use while loop.
- If you have to manipulate the iteration pointer, use a for loop, it's way easier to do.

Recursion: O(n) / O(log(n)) / O(2^n)

- Use it when you need to output one solution, if you need multiple solutions probably Backtracking is better.
- Try it when you can identify a recurrent case (usually a mathematical formula). That is a subproblem you can solve over and over again until you reach an end condition, and then resolve bubbling up to the top. You can visualize it like a Tree data structure.
- Useful when you don't see any other way than testing all paths.

- Problems:
    - Fibbonacci (easy): The recurrent case is `fib(i) = fib(i-1) + fib(i-2)`
    - House Robber (medium): The recurrent case is `rob(i) = Math.max(rob(i - 2) + currentHouseValue, rob(i - 1))`

Linear Search: O(n)

- Use to find a value in an unsorted Linked List or Array.

Binary Search: O(log(n))

- Use it when input is sorted and you can choose left or right in all cases.

Two Pointers: O(n)

- Use when you have a solid logic for when to move the left or right pointer.
- If the array is sorted it can be used too.
- Problems:
    - Check palindromes.
    - Container with most water (medium and hard): To maximize water you need to move the lesser value pointer as the taller the walls the more water you can fit.
    - Maximum Product of Two Elements in an Array (easy): To find the biggest product you move the lesser value pointer as you maximize by maximizing both ends.

Floyd's Tortoise and Hare: O(n)

- Use when detecting a cycle in a Linked List or a graph.

Sliding Window: O(n)

- Try it when you want a sequential portion of an array.
- Try it when Two Pointers cannot be used, f.e. when you have negative numbers.
- Problems:
    - Maximum Contiguous Subarray (medium): You can decide when to drop the left part of the subarray (when it sums less than zero).
    - Longest Substring Without Repeating Characters (medium): You drop the left part when you find a repeated character.

Memoization: O(n)

- Use when you can save previous work to avoid repeating it in the future. Usually your `dp` HashMap will have the key equal to the value of function parameters.
- Use when you want to extend the range of possible values before a Stack Overflow error.
- Use when you need to derive the solution from the memoized HashMap (bottom up).
- Problems:
    - Fibbonacci: You calculate the value once for each value.
    - Coin Change (bottom-up and top-down): You calculate the amount value only once and can build from there.

Backtracking: typically O(candidates^n)

- Use it when you need to output more than one solution. If the problem asks for one solution you probably can use recursion + memoization instead.

- Use when you have to test candidates for a solution and then decide if you discard the tested candidate or not before proceeding and returning a valid one.

DFS: O(V + E)

- Use when the input is a graph or tree
- Preorder ➞ 9, 4, 1, 6, 7, 20, 15, 16, 170
    - Use when needing to recreate the tree.

- Inorder ➞ 1, 4, 6, 7, 9, 15, 16, 20, 170
    - Useful to order a tree

- Postorder ➞ 1, 7, 6, 4, 16, 15, 170, 20, 9
    - Useful when wanting to get the smallest value.

- Problems:
    - Find longest/deepest path in a BST.
    - Find the exit of a maze.

BFS: O(V + E)

- Use when the input is a graph or tree.
- Problems:
    - Find shortest path.
    - Scan tree by levels.
    - Find highest values in a heap.

Topological Sort (Graphs)

- Use when you want to sort the nodes from less connection to them to more.
- Use when wanting to detect a cycle in a Directed Graph. Since Topological Sort can only take nodes that have 0 incoming connections left if there's a point where you can't take a node to add to the sorted array it means there's a cycle in there.
- Problems:
    - Course Schedule (medium).

Dijkstra: O(E log V)

- Use when input is weighted graph.

Sorting

Check complexities in Big O Cheatsheet. Lowest possible best case Θ(n), average case is Θ(n log(n)).

- Data is almost sorted ➞ Insertion Sort.
- Need Space complexity O(1) ➞ Insertion Sort.
- Need stable sorting ➞ Merge Sort / Insertion Sort.
- Need consistency in all cases ➞ Merge Sort.
- Need parallelization ➞ Merge Sort.

- Need the fastest sorting AND can choose a good pivot ➞ Quick Sort.
- Input is integers AND you know the min or max value ➞ Radix/Counting sort.

Optimizations

- The output requested isn't a data structure ➞ Try a solution with space complexity O(1) by working on the same inputs instead of storing extra data.
- You do a lot of repeated checks/calculations ➞ Try memoization approach.
- Have you tried iterating from behind? ➞ No? Try it!
- Return early when you can to save iterations.

Tricks and code snippets

- Clean a string only leaving alphanumeric characters (the `^` means not in sequence): `string.replaceAll(/[^A-Za-z0-9]+/g, '')`.
- If you want to pass counters to a recursive function you probably want to pass everything by value to maintain the correct counter at each recursive level. In the `'aabaabab'` Codility problem the correct signature was `tryAddChar('a', Object.assign({}, prevChars), str, A, B)`.