**Tips for find solutions to coding interview problems**

1. Visualize the problem by drawing it out

2. Think about how you would solve the problem by hand

3. Come up with more examples

4. Break the question down into smaller independent parts

5. Apply common data structures and algorithms at the problem

**Common data structures.** These are the data structures to keep in mind and try, in order of frequency they appear in coding interview questions:

* **Hash Maps**: Useful for making lookup efficient. This is the most common data structure used in interviews and you are guaranteed to have to use it.
* **Graphs**: If the data is presented to you as associations between entities, you might be able to model the question as a graph and use some common graph algorithm to solve the problem.
* **Stack and Queue**: If you need to parse a string with nested properties (such as a mathematical equation), you will almost definitely need to use stacks.
* **Heap**: Question involves scheduling/ordering based on some priority. Also useful for finding the max K/min K/median elements in a set.
* **Tree/Trie**: Do you need to store strings in a space-efficient manner and look for the existence of strings (or at least part of them) very quickly?

**Routines:**

* Sorting
* Binary search: Useful if the input array is sorted and you need to do faster than O(n) searches
* Sliding window
* Two pointers
* Union find
* BFS/DFS
* Traverse from the back
* Topological Sorting

**How to optimize time complexity**

1. Identify the Best Theoretical Time Complexity of the solution

2. Identify overlapping and repeated computation

3. Try different data structures

4. Identify redundant work:

* Don't check conditions unnecessarily
* Mind the order of checks
* Don't invoke methods unnecessarily
* Early termination
* Minimize work inside loops
* Lazy evaluation - is an evaluation strategy which delays the evaluation of an expression until its value is needed

**How to optimize space complexity**

1. Changing data in-place/overwriting input data

2. Change a data structure

**Array**

**Common terms.** Common terms you see when doing problems involving arrays:

* Subarray - A range of contiguous values within an array.
  + Example: given an array [2, 3, 6, 1, 5, 4], [3, 6, 1] is a subarray while [3, 1, 5] is not a subarray.
* Subsequence - A sequence that can be derived from the given sequence by deleting some or no elements without changing the order of the remaining elements.
  + Example: given an array [2, 3, 6, 1, 5, 4], [3, 1, 5] is a subsequence but [3, 5, 1] is not a subsequence.

**Time complexity**

| **Operation** | **Big-O** | **Note** |
| --- | --- | --- |
| Access | O(1) |  |
| Search | O(n) |  |
| Search (sorted array) | O(log(n)) |  |
| Insert | O(n) | Insertion would require shifting all the subsequent elements to the right by one and that takes O(n) |
| Insert (at the end) | O(1) | Special case of insertion where no other element needs to be shifted |
| Remove | O(n) | Removal would require shifting all the subsequent elements to the left by one and that takes O(n) |
| Remove (at the end) | O(1) | Special case of removal where no other element needs to be shifted |

**Things to look out for during interviews**

* Clarify if there are duplicate values in the array. Would the presence of duplicate values affect the answer? Does it make the question simpler or harder?
* When using an index to iterate through array elements, be careful not to go out of bounds.
* Be mindful about slicing or concatenating arrays in your code. Typically, slicing and concatenating arrays would take O(n) time. Use start and end indices to demarcate a subarray/range where possible.

**Corner cases**

* Empty sequence
* Sequence with 1 or 2 elements
* Sequence with repeated elements
* Duplicated values in the sequence

**Techniques**

Sliding window || Two pointers || Traversing from the right || Sorting the array || Precomputation (pre-computation using hashing or a prefix/suffix sum/product) || Index as a hash key ||Traversing the array more than once || Dynamic Programming || Binary Search || Monotonic Queues (Deacreasing & Increasing)

**String**

A string is a sequence of characters. Many tips that apply to arrays also apply to strings. You're recommended to read the page on [Arrays](https://www.techinterviewhandbook.org/algorithms/array/) before reading this page.

**Common data structures** for looking up strings:

* Trie/Prefix Tree
* Suffix Tree

**Common string algorithms**:

* Rabin Karp for efficient searching of substring using a rolling hash
* KMP for efficient searching of substring

**Time complexity**

A strings is an array of characters, so the time complexities of basic string operations will closely resemble that of array operations.

| **Operation** | **Big-O** |
| --- | --- |
| Access | O(1) |
| Search | O(n) |
| Insert | O(n) |
| Remove | O(n) |

Operations involving another string.

Here we assume the other string is of length m.

| **Operation** | **Big-O** | **Note** |
| --- | --- | --- |
| Find substring | O(n + m) | Efficient algorithm for string searching such as the [KMP algorithm](https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm) |
| Concatenating strings | O(n + m) |  |
| Slice | O(m) |  |
| Split (by token) | O(n + m) |  |
| Strip (remove leading and trailing whitespaces) | O(n) |  |

**Corner cases**

* Empty string
* String with 1 or 2 characters
* String with repeated characters
* Strings with only distinct characters

**Techniques**

Counting characters || String of unique characters || Anagram || Palindrome