General notes and tips

- Python is probably the best language to code in for LeetCode type problems. You can focus more on the algorithm rather than the syntax details of the programming language because Python has simple syntax.
- Take notes of what the interviewer is saying. It's not guaranteed that they'll put the question and test cases up on the screen, and it can be a bad look if you ask questions that you would already know the answer to if you were paying attention.
- Ask the interviewer questions. "What is the recommended time complexity?" "What is the recommended space complexity?" Ask about assumptions about the input (e.g. is it already sorted?). It's okay to ask the interviewer about their thoughts on two different approaches/algorithms to implement a solution.
- At least initially, focus more on writing a correct algorithm rather than an efficient one.
- If you're given time/space complexity recommendations/constraints, use them to give you hints about what data structures or algorithms might be the best for solving the problem.
- You can start off defining variables or functions that you can easily determine or know you will need to define. Examples include the variables that are used in time/space complexity recommendations/constraints and the value you need to return.
- Write test cases, and try to cover edge cases. You can draw pictures to help yourself understand how your code would handle the test cases.
- Give variables descriptive names (e.g. `for carIndex in range(len(cars))` rather than `for i in range(len(c))`)
- Don't worry about defining too many or unnecessary variables. The added readability is worth the negligible inefficiency.
- Don't hesitate to define helper functions. If you do, add comments about what inputs the function requires, what value(s) it returns, if any, and what side-effects it has, if any.
- Write comments explaining what values a variable holds if it would be hard to come up with a decently short, descriptive name for it.
- Practice, practice, practice! Get comfortable with using the various data structures/common techniques, and you should become better at recognizing patterns so that you can solve problems you've never seen before more easily. The NeetCode 150 is a good place to start.

Data structures in Python

`List` (also called array; can use to implement a stack) Initialize with: `l = [], l = list()`		
Append value **v** to the back of **l**	`l.append(v)`	$O(1)$
Access the element at index **i** of **l**	`l[i]`	$O(1)$
Pop from the back of **l**	`l.pop()`	$O(1)$

`Set` Initialize with: `s = set(), s = set(l: List)`		
Add to value **v** to **s**	`s.add(v)`	$O(1)$
Check if value **v** in **s**	`v in s`	$O(1)$

Remove value **v** from **s**	`s.remove(v)`
	`s.discard(v)`
		$O(1)$

`Dictionary` (also called hash map) Keys can be anything that is hashable- strings, ints, user-defined objects. Lists are not hashable because they're mutable, but tuples are immutable and hashable. Initialize with: `d = {}, d = dict()`		
Assign value **v** to key **k**	`d[k] = v`	$O(1)$
Check if key **k** in **d**	`k in s`	$O(1)$

Retrieve the value with key **k**	`d[k]`
	`d.get(k, fallbackValue)`
		$O(1)$

Delete key **k** from **d**	`del d[k]`
	`d.pop(k)`
		$O(1)$

`Queue` (can use to implement a stack. Note deque is an abbreviation for double-ended queue and is pronounced "deck")
Initialize with:
`from collections import deque`
`q = deque()`		

Append value **v** to the back of **q**	`q.append(v)`	$O(1)$
Append value **v** to the front of **q**	`q.appendleft(v)`	$O(1)$
Retrieve the element at the back of **q**	`q[-1]`	$O(1)$
Retrieve the element at the front of **q**	`q[0]`	$O(1)$
Pop from the back of **q**	`q.pop()`	$O(1)$
Pop from the front of **q**	`q.popleft()`	$O(1)$

`Priority Queue` (also called heap; Python's implementation is a min-heap)
Initialize with:
```
import heapq
pq = []
```
Note: rather than using pairs with explicit priorities, you may also use an instance of a class with the __lt__(self, other) method defined to determine which of two different instances has the higher priority.

Make **pq** heap-ordered	**`heapq.heapify(pq)`**	$O(n)$
Push value **v** with priority **p** into **pq**	**`heapq.heappush(pq, (p, v))`**	$O(log(n))$
Retrieve the **`(p, v)`** pair with the minimum **p** in **pq**	**`pq[0]`**	$O(1)$
Pop the **`(p, v)`** pair with the minimum **p** from **pq**	**`heapq.heappop(pq)`**	$O(log(n))$

`Disjoint Set` (also called union-find)
Initialize with:
```
from scipy.cluster.hierarchy import DisjointSet
ds = DisjointSet()
```

Add value **v** to **ds**	**`ds.add(v)`**	$O(1)$
Merge (*union*) **u** and **v**'s subsets	**`ds.merge(u, v)`**	$O(1)$
Check whether **u** and **v** are in the same subset	**`ds.connected(u, v)`**	$O(1)$
Get (*find*) the subset containing **v**	**`ds.subset(v)`**	$O(1)$
Get all subsets	**`ds.subsets()`**	$O(1)$

User-defined data structures

`Singly linked list`
Definition:
```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next   # ListNode
```
Initialize with:
```
head = ListNode()
```

`Doubly linked list`
Definition:
```
class DoublyListNode:
    def __init__(self, val=0, prev=None, next=None):
        self.val = val
        self.prev = prev   # DoublyListNode
        self.next = next   # DoublyListNode
```
Initialize with:
```
head = tail = DoublyListNode()
```

`Binary tree`
Definition:
```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left   # TreeNode
        self.right = right   # TreeNode
```
Initialize with:
```
root = TreeNode()
```

`Prefix tree` (also called trie)
Definition:
```
class TrieNode:
    def __init__(self):
        self.children = {}   # Dictionary mapping values to TrieNodes
        self.is_end = False
```
Initialize with:
```
root = TrieNode()
```

```
Graph
Definition:
class GraphNode:
    def __init__(self, val=0, neighbors=None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []   # List of GraphNodes
Initialize with:
node = GraphNode()
```

Built-in functions

- **`range(n: num)`** returns a **`List`** from to to **`n`** - 1, mainly used in for loops
- **`len(container)`** returns the length of the container. For **`List`** and **`Set`**, the number of items in it; for **`Dictionary`**, the number of keys in it.
- **`l.sort(key?)`** sorts **`l`** (which is a **`List`**) in-place, meaning it does not create a copy and modifies **`l`**.
  - key can be a lambda such as **`key=lambda item: item[1]`** or **`key=lambda x: -x`** or **`key=lambda c: -ord(c)`**. The lambda is applied to every value in **`l`** and sorts based on the return values of the lambda.
- **`sorted(i, key?)`** returns a sorted copy of **`i`** (which is an iterable container such as a **`List`**, **`Set`**, or **`Dictionary`**), leaving **`i`** unchanged.
  - If **`i`** is a **`Set`**, it returns a sorted **`List`** of **`i`**'s values. If **`i`** is a **`Dictionary`**, it returns a sorted **`List`** of the **`i`**'s keys.
- **`max(i, key?)`** returns the maximum element in **`i`** (which is an iterable container such as a **`List`**, **`Set`**, or **`Dictionary`**).
- **`max(arg1, arg2, …, key?)`** returns the maximum argument among the provided arguments.
- **`min(...)`** works similarly to **`max()`**.
- **`ord(c: char)`** returns the unicode code of **`c`** (which is a **`string`** with only one character).

Time complexity

- $O(1)$ - index into list, retrieve from/insert into set, retrieve from/insert key into dictionary
- $O(log(n))$ - binary search (or otherwise halving search space each iteration), insert into/pop from priority queue
- $O(n)$ - iterating through list or linked list
- $O(n * log(n))$ - sorting, n insertions into a priority queue
- $O(n^2)$ - nested for loop, compare each element to every other element (i.e examine all pairs of elements)
- $O(2^n)$ - generating all subsets of n elements
- $O(n!)$ - generating all permutations of n elements

Space complexity

- $O(1)$ - two pointers, sliding window, constant number of variables
- $O(log(n))$ - recursion on balanced binary tree
- $O(n)$ - list
- $O(n * log(n))$ - mergesort
- $O(n^2)$ - matrices (list of lists)
- $O(2^n)$ - storing all subsets
- $O(n!)$ - storing all permutations