How to Ace the Data Science Coding

1. Sorting

	Time Complexity	Space Complexity	Stable	In-Place
Bubble Sort	$O(n^2)$	$O(1)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n)$	Yes	No
Quick Sort	$O(n \log n)$	$O(\log n)$	No	Yes
Heap Sort	$O(n \log n)$	$O(1)$	No	Yes
Tim Sort	$O(n \log n)$	$O(n)$	Yes	No
Bucket Sort	$O(n + k)$	$O(n + k)$	Yes	No

1.1. Merge Sort

Divide-and-conquer algorithm; stable and efficient but uses extra memory. Divides the unsorted array into smaller arrays, sorts them, and then merges them back together.

```
def merge_sort(arr):
if len(arr) > 1:
    mid = len(arr) // 2
    L = arr[:mid]
    R = arr[mid:]

    merge_sort(L) #sort left half
    merge_sort(R) #sort right half

    i = j = k = 0

            # merge sorted halfs
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
```

1.2. Quick sort

Fast and in-place but not stable. Partitions the array into smaller arrays around a pivot and sorts them recursively.

```
def quickSort(arr, s, e):
    if e - s + 1 <= 1:
        return

    pivot = arr[e]
    left = s # pointer for left side

    # Partition: elements smaller than pivot on left side
    for i in range(s, e):
        if arr[i] < pivot:
            tmp = arr[left]
            arr[left] = arr[i]
            arr[i] = tmp
            left += 1

    # Move pivot in-between left & right sides
    arr[e] = arr[left]
    arr[left] = pivot

    # Quick sort left side
    quickSort(arr, s, left - 1)

    # Quick sort right side
    quickSort(arr, left + 1, e)

    return arr
```

1.3. Heap Sort

In-place but not stable. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. Repeat for the remaining elements.

```
import heapq

def heap_sort(arr):
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]
```

1.4. Tim Sort

divides the list into small chunks, then sorts the chunks using an optimized version of insertion sort. Finally, it merges the sorted chunks in a manner similar to merge sort

```
arr.sort()
```

1.5. Bucket Sort

applicable if small range (like 0 - 100k), only very rare, create bucket for every value, not stable

```
def bucketSort(arr):
    # Assuming arr only contains 0, 1 or 2
    counts = [0, 0, 0]

    # Count the quantity of each val in arr
    for n in arr:
        counts[n] += 1

    # Fill each bucket in the original array
    i = 0
    for n in range(len(counts)):
        for j in range(counts[n]):
            arr[i] = n
            i += 1
    return arr
```

2. Searching

	Time	Space	Requirements	Best Use-Case
Linear Search	$O(n)$	$O(1)$	None	Unsorted or small data
Binary Search	$O(\log n)$	$O(1)$	Sorted Array	Large, sorted data

2.1. Linear Search

goes through each element in the list sequentially until the desired element is found (or list ends)

2.2. Binary Search

only on sorted arrays, repeatedly divide the sorted list into halves until the target element is found

```python
def binarySearch(arr, target):
    L, R = 0, len(arr) - 1

    while L <= R:
        mid = (L + R) // 2

        if target > arr[mid]:
            L = mid + 1
        elif target < arr[mid]:
            R = mid - 1
        else:
            return mid
    return -1
```

2.2.1. Find hidden number in range

```python
def binarySearch(low, high):
    while low <= high:
        mid = (low + high) // 2

        if isCorrect(mid) > 0:
            high = mid - 1
        elif isCorrect(mid) < 0:
            low = mid + 1
        else:
            return mid
    return -1

# Return 1 if n is too big, -1 if too small, 0 if correct
def isCorrect(n):
    if n > 10: #hidden target value
        return 1
    elif n < 10:
        return -1
    else:
        return 0
```

3. Linked Lists

3.1. Singly Linked Lists

```python
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

class LinkedList:
    def __init__(self):
        # Init the list with a 'dummy' node which makes removing a node
        self.head = ListNode(-1)
        self.tail = self.head

        def insertEnd(self, val):
        self.tail.next = ListNode(val)
        self.tail = self.tail.next

    def remove(self, index):
        i = 0
        curr = self.head
        while i < index and curr:
            i += 1
            curr = curr.next

        # Remove the node ahead of curr
        if curr and curr.next:
            if curr.next == self.tail:
                self.tail = curr
            curr.next = curr.next.next

    def print(self):
        curr = self.head.next
        while curr:
            print(curr.val, " -> ", end="")
            curr = curr.next
        print()

    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNod
        new_list = None
        current = head

        while current:
            next_node = current.next
            current.next = new_list
            new_list = current
            current = next_node

        return new_list
```

3.2. Doubly Linked Lists

```python
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
        self.prev = None

# Implementation for Doubly Linked List
class LinkedList:
    def __init__(self):
        # Init the list with 'dummy' head and tail nodes which makes
        # edge cases for insert & remove easier.
        self.head = ListNode(-1)
        self.tail = ListNode(-1)
        self.head.next = self.tail
        self.tail.prev = self.head
```

3.3. Queue

```python
from collections import deque
```

Command	Explanation	Use-Case
`append(x)`	Adds x to the right side of the deque.	Appending an element at the end.
`appendleft(x)`	Adds x to the left side of the deque.	Prepending an element at the beginning.
`pop()`	Removes and returns an element from the right side of the deque.	Removing the last element.
`popleft()`	Removes and returns an element from the left side of the deque.	Removing the first element.
`extend(iterable)`	Adds all elements from iterable to the right side of the deque.	Extending the deque with multiple elements at the end.
`extendleft(iterable)`	Adds all elements from iterable to the left side of the deque.	Extending the deque with multiple elements at the beginning.
`rotate(n)`	Rotates the deque n steps to the right.	Rotating all elements n steps to the right.
`count(x)`	Counts the number of deque elements equal to x.	Counting occurrences of a specific element.
`remove(value)`	Removes the first occurrence of value.	Removing a specific element by value.
`reverse()`	Reverses the elements of the deque in-place.	Reversing the order of elements.
`clear()`	Removes all elements from the deque.	Clearing all elements from the deque.
`index(x[, start[, end]])`	Returns the position of x in the deque.	Finding the index of a specific element.

- FIFO append() and popleft()
- LIFO append() and pop()

4. Trees

4.1. Binary Trees

each node two children, no cycles allowed

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

4.1.1. Delete nodes

```
def delete_nodes(root, to_delete):
    """
    Deletes nodes from a binary tree given their values.

    Parameters:
    - root: The root of the binary tree.
    - to_delete: A set of node values to be deleted.

    Returns:
    - The root of the modified tree.
    """
    if root is None:
        return None

    # If the current node should be deleted
    if root.val in to_delete:
        # Perform extra operations on the node to be deleted

        # Delete the node and return None
        return None

    # Recursively delete nodes in the left and right subtrees
    root.left = delete_nodes(root.left, to_delete)
    root.right = delete_nodes(root.right, to_delete)

    return root
```

Recursion:

- passing info downwards – by arguments
- passing info upwards – by return value

4.2. Binary Search Tree

sorted property: every left child must be smaller and every right child greater than its parent, no duplicates

4.2.1. Search

Time: $O(\log n)$

```
def search(root, target):
    if not root: #if root NONE/NULL
        return False

    if target > root.val:
        return search(root.right, target)
    elif target < root.val:
        return search(root.left, target)
    else:
        return True
```

4.2.2. Insert

```
# Insert a new node and return the root of the BST.
def insert(root, val):
    if not root:
        return TreeNode(val)

    if val > root.val: #call insert on right subtree
        root.right = insert(root.right, val)
    elif val < root.val: #call insert on left subtree
        root.left = insert(root.left, val)
    return root
```

4.2.3. Remove

```
# Return the minimum value node of the BST.
def minValueNode(root):
    curr = root
    while curr and curr.left: #while current node and left node is not
        curr = curr.left
    return curr

# Remove a node and return the root of the BST.
def remove(root, val):
    if not root:
        return None

    if val > root.val:
        root.right = remove(root.right, val)
    elif val < root.val:
        root.left = remove(root.left, val)
    else:
        if not root.left: # if no left child, return righ child
            return root.right
        elif not root.right:
            return root.left
        else: # replace with smallest value in subtree and remove small
            minNode = minValueNode(root.right)
            root.val = minNode.val
            root.right = remove(root.right, minNode.val)
    return root
```

4.3. Depth-First Search

visit left deepest node, travers up, ...
Time: $O(n)$

```
def inorder(root):
    if not root:
        return
    inorder(root.left)
    print(root.val)
    inorder(root.right)
```

4.4. Breadth-First Search

Time: $O(n)$

```
from collections import deque

def bfs(root):
    queue = deque() #FIFO queue

    if root:
        queue.append(root)

    level = 0
    while len(queue) > 0:
        print("level:␣", level)
        for i in range(len(queue)):
            curr = queue.popleft()
            print(curr.val)
            if curr.left:
                queue.append(curr.left)
            if curr.right:
                queue.append(curr.right)
        level += 1
```

4.5. Backtracking

Determine if path exists (e.g. without any zeros), recursively try every path, Time: $O(n)$

```
def leafPath(root, path):
    if not root or root.val == 0:
        return False
    path.append(root.val)

    if not root.left and not root.right:
        return True
    if leafPath(root.left, path):
        return True
    if leafPath(root.right, path):
        return True
    path.pop() # backtrack because path didnt work
    return False
```

5. Heap/Priority Queue

pop values based on priority (min or max)

- structure property: complete binary tree (every single level in the tree is full, except the last level), missing nodes are at the end of level (right side)
- order property: parent is always smaller than its children

```
leftChild of i = heap[2 * i]
rightChild of i = heap[(2 * i) + 1]
parent of i = heap[i // 2]
```

5.1. Implemented Python commands

Command	Explanation	Use-Case
`heapify(iterable)`	Transforms the iterable into a valid heap, in-place.	Creating a heap from an existing list in $O(n)$ time.
`heappush(heap, elem)`	Adds an element to the heap while maintaining the heap property.	Adding a new element to a heap.
`heappop(heap)`	Removes and returns the smallest element from the heap.	Extracting the minimum element from a heap.
`heappushpop(heap, elem)`	Pushes a new element on the heap, then pops and returns the smallest element from the heap.	Efficiently adding an element and then removing the smallest.
`heapreplace(heap, elem)`	Pops and returns the smallest element, and then adds the new element to the heap.	Replacing the smallest element in a heap with a new value.
`heapify(heap)`	Transforms a list into a heap, in-place.	Transforming an unsorted list into a heap.
`nsmallest(n, iterable[, key])`	Returns the n smallest elements from the iterable, in ascending order.	Finding the n smallest elements in a collection.
`nlargest(n, iterable[, key])`	Returns the n largest elements from the iterable, in descending order.	Finding the n largest elements in a collection.

6. Graphs

$Edges \leq Vertices(nodes)^2$ → every pointer can go to everywhere (cycles allowed)
Represent as

- Matrix: grid graph
- Adjacency Matrix (less common)
- Adjacency List: neighbors as list (no cycles)

6.1. Graphs as Adjacency List

neighbors as list (no cycles)

```python
from collections import deque

# GraphNode used for adjacency list
class GraphNode:
    def __init__(self, val):
        self.val = val
        self.neighbors = []

# Or use a HashMap
adjList = { "A": [], "B": [] }

# Given directed edges, build an adjacency list
edges = [["A", "B"], ["B", "C"], ["B", "E"], ["C", "E"], ["E", "D"]]

adjList = {}

for src, dst in edges:
    if src not in adjList:
        adjList[src] = []
    if dst not in adjList:
        adjList[dst] = []
    adjList[src].append(dst)


# Count paths (backtracking)
def dfs(node, target, adjList, visit):
    if node in visit:
        return 0
    if node == target:
        return 1

    count = 0
    visit.add(node)
    for neighbor in adjList[node]:
        count += dfs(neighbor, target, adjList, visit)
    visit.remove(node)

    return count

# Shortest path from node to target
def bfs(node, target, adjList):
    length = 0
    visit = set()
    visit.add(node)
    queue = deque()
    queue.append(node)

    while queue:
        for i in range(len(queue)):
            curr = queue.popleft()
            if curr == target: #reached target
                return length

            for neighbor in adjList[curr]:
                if neighbor not in visit:
                    visit.add(neighbor)
                    queue.append(neighbor)
        length += 1
    return length
```

6.2. Matrix as Graph

6.2.1. Matrix Breadth-First Search (BFS)

explores all neighbor nodes before moving on to nodes at the next depth level ⇒ can find shortest paths between two nodes if unweighted

- Time: $O(nm)$
- Space: $O(V)$

```python
# Shortest path from top left to bottom right
def bfs(grid):
    ROWS, COLS = len(grid), len(grid[0])
    visit = set()
    queue = deque()
    queue.append((0, 0))
    visit.add((0, 0))

    length = 0
    while queue:
        for i in range(len(queue)):
            r, c = queue.popleft()
            if r == ROWS - 1 and c == COLS - 1:
                return length #reached goal

            neighbors = [[0, 1], [0, -1], [1, 0], [-1, 0]]
            for dr, dc in neighbors:
                if (min(r + dr, c + dc) < 0 or #not out of bounds
                    r + dr == ROWS or c + dc == COLS or
                    (r + dr, c + dc) in visit or grid[r + dr][c + dc] ==
                    continue
                queue.append((r + dr, c + dc))
                visit.add((r + dr, c + dc))
        length += 1
```

6.2.2. Matrix Depth-First Search (DFS)

first as deep as possible before backtracking, useful for scenarios where you want to go as deep as possible into the tree/graph, like solving mazes

- Time: $O(4^{nm})$
- Space: $O(n + m)$

```python
# Matrix (2D Grid)
grid = [[0, 0, 0, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 1, 0, 0]]

# Count paths (backtracking)
def dfs(grid, r, c, visit): #r, c: starting row and col,
    ROWS, COLS = len(grid), len(grid[0])
    if (min(r, c) < 0 or
        r == ROWS or c == COLS or #dont move out of bounds
        (r, c) in visit or grid[r][c] == 1): #reach visited or blocked po
        return 0 # no valid path
    if r == ROWS - 1 and c == COLS - 1:
        return 1 #reach last row and col

    visit.add((r, c))

    count = 0
    count += dfs(grid, r + 1, c, visit)
    count += dfs(grid, r - 1, c, visit)
    count += dfs(grid, r, c + 1, visit)
    count += dfs(grid, r, c - 1, visit)

    visit.remove((r, c))
    return count

print(dfs(grid, 0, 0, set()))
```

7. Common problems

7.1. Detect cyles in a list

keeping track of visited nodes results in $O(n^2$ time, improve by using two pointers moving at different speeds to detect a cycle. This approach has $O(1)$ space complexity and $O(n)$ time complexity.

```
def hasCycle(self, head: Optional[ListNode]) -> bool:
    if not head:
        return False

    slow, fast = head, head.next

    while fast is not None and fast.next is not None:
        if slow == fast:
            return True

        slow = slow.next
        fast = fast.next.next

    return False
```

7.2. Dynamic Programming: Fibonacci

if recursively $O(2^n)$, but with DP $O(n)$

```
# Brute Force
def bruteForce(n):
    if n <= 1:
        return n
    return bruteForce(n - 1) + bruteForce(n - 2)

# Memoization
def memoization(n, cache):
    if n <= 1:
        return n
    if n in cache:
        return cache[n]

    cache[n] = memoization(n - 1) + memoization(n - 2)
    return cache[n]

# Dynamic Programming
def dp(n):
    if n < 2:
        return n

    dp = [0, 1]
    i = 2
    while i <= n:
        tmp = dp[1]
        dp[1] = dp[0] + dp[1]
        dp[0] = tmp
        i += 1
    return dp[1]
```

7.3. Dynamic Programming: Longest Common Subsequence

naive recursive has exponential time complexity, but reduced to polynomial

```
# Time: O(2^(n + m)), Space: O(n + m)
def dfs(s1, s2):
    return dfsHelper(s1, s2, 0, 0)

def dfsHelper(s1, s2, i1, i2):
    if i1 == len(s1) or i2 == len(s2):
        return 0

    if s1[i1] == s2[i2]:
        return 1 + dfsHelper(s1, s2, i1 + 1, i2 + 1)
    else:
        return max(dfsHelper(s1, s2, i1 + 1, i2),
                   dfsHelper(s1, s2, i1, i2 + 1))


# Time: O(n * m), Space: O(n + m)
def memoization(s1, s2):
    N, M = len(s1), len(s2)
    cache = [[-1] * M for _ in range(N)]
    return memoHelper(s1, s2, 0, 0, cache)

def memoHelper(s1, s2, i1, i2, cache):
    if i1 == len(s1) or i2 == len(s2):
        return 0
    if cache[i1][i2] != -1:
        return cache[i1][i2]

    if s1[i1] == s2[i2]:
        cache[i1][i2] = 1 + memoHelper(s1, s2, i1 + 1, i2 + 1, cache)
    else:
        cache[i1][i2] = max(memoHelper(s1, s2, i1 + 1, i2, cache),
                   memoHelper(s1, s2, i1, i2 + 1, cache))
    return cache[i1][i2]


# Time: O(n * m), Space: O(n + m)
def dp(s1, s2):
    N, M = len(s1), len(s2)
    dp = [[0] * (M+1) for _ in range(N+1)]

    for i in range(N):
        for j in range(M):
            if s1[i] == s2[j]:
                dp[i+1][j+1] = 1 + dp[i][j]
            else:
                dp[i+1][j+1] = max(dp[i][j+1], dp[i+1][j])
    return dp[N][M]


# Time: O(n * m), Space: O(m)
def optimizedDp(s1, s2):
    N, M = len(s1), len(s2)
    dp = [0] * (M + 1)

    for i in range(N):
        curRow = [0] * (M + 1)
        for j in range(M):
            if s1[i] == s2[j]:
                curRow[j+1] = 1 + dp[j]
            else:
                curRow[j+1] = max(dp[j + 1], curRow[j])
        dp = curRow
    return dp[M]
```

7.4. Knapsack Problem

given a set of items, each with a weight and a value. You have a knapsack with a fixed carrying capacity (maximum weight it can hold). The goal is to select a subset of the items in such a way that their combined weight is less than or equal to the knapsack's capacity, and their combined value is maximized.

```
def knapsack(W, wt, val, n):
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif wt[i-1] <= w:
                dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][W]
```

7.5. Detonate the Maximum Bombs

Convert list to graph based on condition and perform bfs on graph.

```
def maximumDetonation(self, bombs: List[List[int]]) -> int:
    graph = collections.defaultdict(list)
    n = len(bombs)

    # Build the graph
    for i in range(n):
        for j in range(n):
            if i == j:
                continue
            xi, yi, ri = bombs[i]
            xj, yj, _ = bombs[j]

            # Create a path from node i to node j, if bomb i detonates bo
            if ri ** 2 >= (xi - xj) ** 2 + (yi - yj) ** 2:
                graph[i].append(j)

    def bfs(i):
        queue = collections.deque([i])
        visited = set([i])
        while queue:
            cur = queue.popleft()
            for neib in graph[cur]:
                if neib not in visited:
                    visited.add(neib)
                    queue.append(neib)
        return len(visited)

    answer = 0
    for i in range(n):
        answer = max(answer, bfs(i))

    return answer
```

7.6. Parallel Course

Convert relation list into graph, with additional counter that prerequisites are fulfilled

```
def minimumSemesters(self, N: int, relations: List[List[int]]) -> int:
    graph = {i: [] for i in range(1, N + 1)}
    in_count = {i: 0 for i in range(1, N + 1)}  # or in-degree
    for start_node, end_node in relations:
        graph[start_node].append(end_node)
        in_count[end_node] += 1

    queue = []
    # we use list here since we are not
    # poping from front the this code
    for node in graph:
        if in_count[node] == 0:
            queue.append(node)

    step = 0
    studied_count = 0
    # start learning with BFS
    while queue:
        # start new semester
        step += 1
        next_queue = []
        for node in queue:
            studied_count += 1
            end_nodes = graph[node]
            for end_node in end_nodes:
                in_count[end_node] -= 1
                # if all prerequisite courses learned
                if in_count[end_node] == 0:
                    next_queue.append(end_node)
        queue = next_queue

    return step if studied_count == N else -1
```

7.7. Height of Binary Tree After Subtree Removal Queries

Convert relation list into graph, with additional counter that prerequisites are fulfilled

```
def treeQueries(self, root: Optional[TreeNode], queries: List[int]) -> List[int]:

    #each node stores (value, max_height to left + own height, max_height to right + own height)
    def get_height(root, current):
        if not root: return [0, 0]
        else:
            left = get_height(root.left, current + 1)
            right = get_height(root.right, current + 1)
            root.val = [root.val, current + max(left), current + max(right)]
            return [max(left) + 1, max(right) + 1]

    #traverse the tree and store the solution for all subtrees
    #carry stores the maximum height so far
    def gen_sol(root, carry, dicts):
        if root.left:
            dicts[root.left.val[0]] = max(carry, root.val[2])
            gen_sol(root.left, max(carry, root.val[2]), dicts)
        if root.right:
            dicts[root.right.val[0]] = max(carry, root.val[1])
            gen_sol(root.right, max(carry, root.val[1]), dicts)

    dicts = {}
    get_height(root, 0)
    gen_sol(root, -1, dicts)

    res = []

    #get solutions from the dictionary
    for element in queries:
        res.append(dicts[element])

    return res
```

7.8. Water and Jug Problem

BFS with always possible actions

```
def canMeasureWater(self, jug1Capacity: int, jug2Capacity: int,
targetCapacity: int) -> bool:

    queue = deque([0])
    visit = set()
    steps = [jug1Capacity, -jug1Capacity, jug2Capacity, -jug2Capacity]

    while queue:
        cur = queue.popleft()
        for step in steps:
            total = cur + step
            if total == targetCapacity:
                return True
            if total not in visit and \
            0 <= total <= jug1Capacity + jug2Capacity:
                visit.add(total)
                queue.append(total)
    return False
```