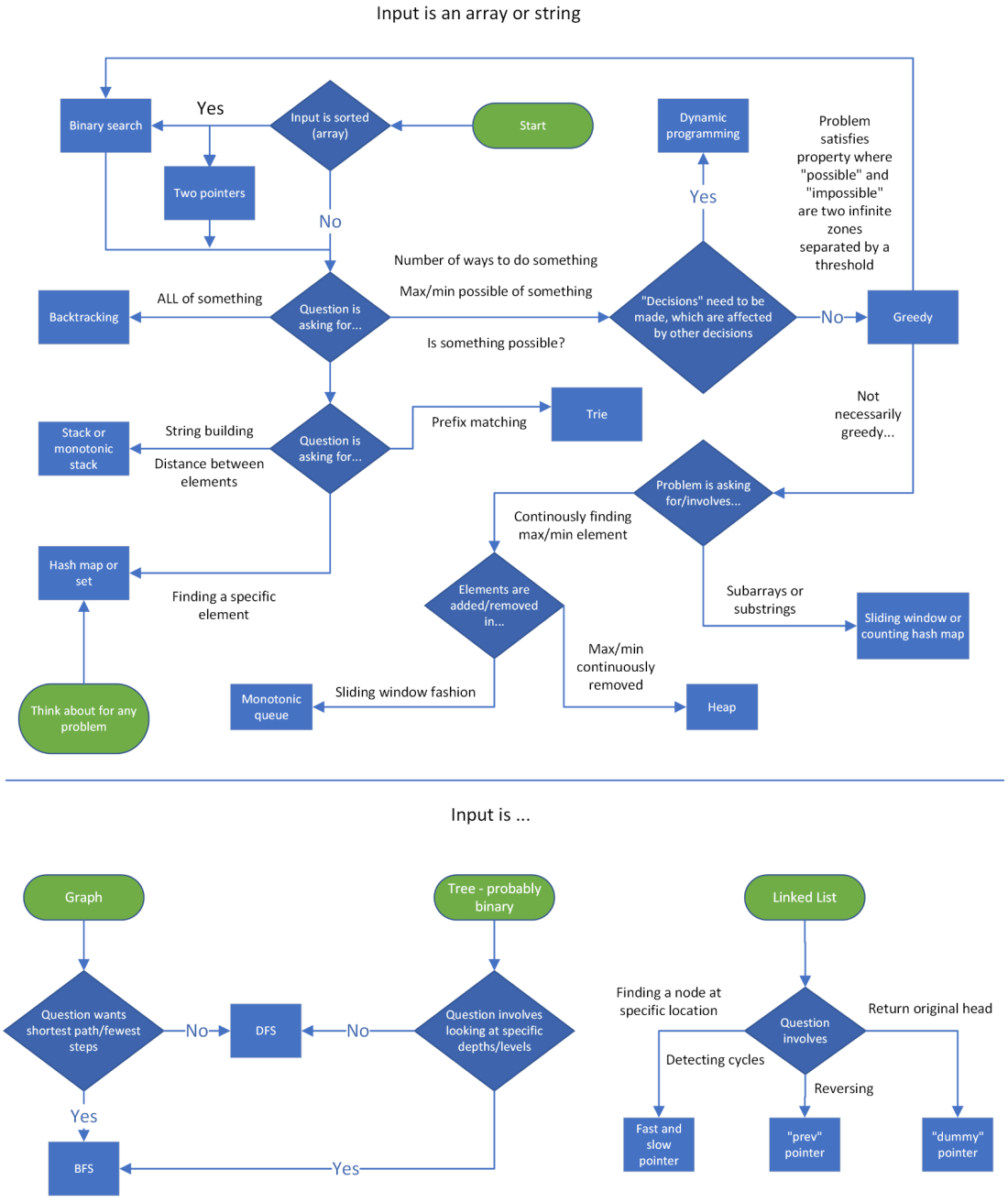
Basics **3**
a. Flow chart 3
b. Big O 4
c. Recursion 4
Arrays & Strings **5**
a. Basics 5
b. Two Pointers 6
c. Sliding Window 8
d. Prefix Sum 9
Hashing **11**
a. Basics 11
b. Checking for existence 12
c. Counting 12
Linked Lists **13**
a. Basics 13
b. Fast and slow pointers 14
c. Reversing a linked list 16
Stacks and queues **17**
a. Stacks 18
b. String problems 19
c. Queues 20
d. Monotonic 22
Trees and graphs **22**
a. Binary trees 22
b. Binary trees - DFS 23
c. Binary trees - BFS 26
d. Binary search trees 27
e. Graphs 28
f. Graphs - DFS 29
g. Graphs - BFS 31
h. Implicit Graphs 33
Heaps **34**
a. Basics 34
b. Top k 35
Greedy **36**
a. Basics 36
Binary Search **37**
a. Basics 37
b. On Arrays 42

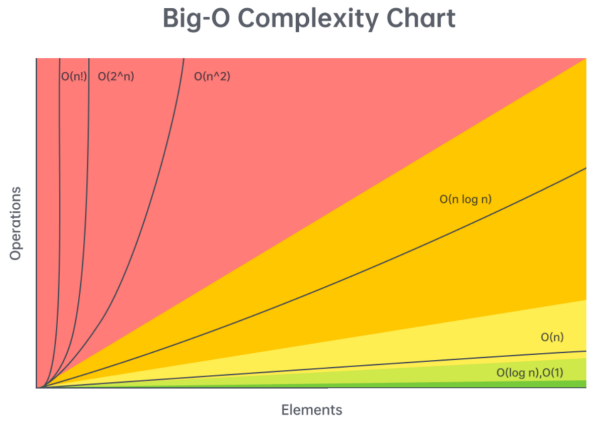
c. On solution spaces.....43
Backtracking.....44
a. Basics.....44
b. Generation.....45
c. More constrained backtracking.....46
Dynamic programming.....47
a. Basics.....47
b. Framework for DP.....48
c. 1D Problems.....49
d. Matrix DP.....49
Advanced.....51
a. Difference array.....51
b. Tries.....52
c. Bit manipulation.....53
d. Modular arithmetic.....54
e. Dijkstra's.....55
More time complexity.....56

Basics

a. Flow chart



b. Big O



Sorting Algorithm	Time Complexity			Space Complexity	Stable
	Best	Average	Worst	Worst	
Quicksort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Heapsort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Timsort	O(n)	O(n log n)	O(n log n)	O(n)	Yes
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No *
Shellsort	O(n log n)	-- (depends on gap sequence)	O(n^2)	O(1)	No
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(nk)	Yes
Radix Sort	O(nk)	O(nk)	O(nk)	O(nk)	Yes
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Yes

*Selection sort can be implemented as a stable sort if, rather than swapping the minimum value with its current value, the minimum value is inserted into the first position and the intervening values shifted up. However, this modification either requires a data structure that supports efficient insertions or deletions, such as a linked list, or it leads to O(n^2) writes.

Miscellaneous

- Sorting: $O(n \cdot \log n)$, where n is the size of the data being sorted
- DFS and BFS on a graph: $O(n \cdot k+e)$, where n is the number of nodes, e is the number of edges, if each node is handled in O(1) other than iterating over edges
- DFS and BFS space complexity: typically O(n), but if it's in a graph, might be O(n+e) to store the graph
- Dynamic programming time complexity: $O(n \cdot k)$, where n is the number of states and k is the work done at each state
- Dynamic programming space complexity: O(n), where n is the number of states

c. Recursion

```
def factorial(n):
    if n == 0 or n == 1:
```

```
        return 1
    else:
        return n * factorial(n-1)

# Example usage:
result = factorial(5)
print("Factorial of 5:", result)
```

This function calculates the factorial of a number using recursion. If the input is 0 or 1, it returns 1. Otherwise, it multiplies the current number by the factorial of the number one less than the current number.

Arrays & Strings

a. Basics

Efficient string building

Efficient string building in Python is often done using the join method rather than repeatedly concatenating strings with the + operator. The join method is more efficient because it joins a list of strings into a single string, and it is particularly useful when you have a large number of strings to concatenate.

```
words = ["Hello", "World", "!"]

# Inefficient way using +
result_inefficient = ""
for word in words:
    result_inefficient += word

# Efficient way using join
result_efficient = "".join(words)

print("Inefficient Result:", result_inefficient)
print("Efficient Result:", result_efficient)
```

In this example, result_inefficient is built by repeatedly concatenating strings using +=, while result_efficient uses the join method. The join method is more efficient, especially when dealing with a large number of strings, because it avoids creating new string objects at each concatenation.
Additionally, if you're building a string from an iterable (like a list comprehension or generator expression), you can directly use join:

```
words = ["Hello", "World", "!"]

# Efficient way using join with list comprehension
result_efficient = "".join(word for word in words)

print("Efficient Result:", result_efficient)
```

Arrays time complexity
Given n = arr.length,

- Add or remove element at the end: O(1)
- Add or remove element from arbitrary index: O(n)
- Access or modify element at arbitrary index: O(1)
- Check if element exists: O(n)
- Two pointers: O(n · k) where k is the work done at each iteration, includes sliding window
- Building a prefix sum: O(n)
- Finding the sum of a subarray given a prefix sum: O(1)

Strings time complexity
Given n = s.length,

- Add or remove character: O(n)
- Access element at arbitrary index: O(1)
- Concatenation between two strings: O(n+m) where m is the length of the other string
- Create substring: O(m), where m is the length of the substring
- Two pointers: O(n * k), where k is the work done at each iteration, includes sliding window
- Building a string from joining an array, stringbuilder, etc: O(n)

b. Two Pointers

Two pointers: One input, opposite ends

The "Two Pointers" technique involves using two pointers that traverse a sequence in opposite directions or maintain a specific relationship. One common use case is when the sequence is sorted, and you want to find a pair of elements that satisfy a certain condition. Let's look at an example where we use two pointers to find a pair in a sorted array that sums up to a target value:

```
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1

    while left < right:
```

```
        current_sum = nums[left] + nums[right]

        if current_sum == target:
            return [nums[left], nums[right]]
        elif current_sum < target:
            left += 1
        else:
            right -= 1

    return None

# Example usage:
sorted_array = [-2, 1, 2, 4, 7, 11]
target_sum = 9

result = two_sum_sorted(sorted_array, target_sum)

if result:
    print(f"Pair with sum {target_sum}: {result}")
else:
    print("No such pair found.")
```

In this example, the two_sum_sorted function uses two pointers (left and right) to iterate through the sorted array from opposite ends. The pointers move toward each other until a pair with the target sum is found or until they meet. This algorithm works because the array is sorted, and it has a time complexity of O(n) since each pointer moves at most once.
This technique is versatile and can be applied to various scenarios, such as finding triplets, checking for a palindrome, or solving problems with specific constraints.

Two pointers: two inputs, exhaust both

```
def merge_sorted_arrays(arr1, arr2):
    result = []
    i, j = 0, 0

    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            result.append(arr1[i])
            i += 1
        else:
            result.append(arr2[j])
            j += 1

    # If there are remaining elements in arr1, append them
```

```
    while i < len(arr1):
        result.append(arr1[i])
        i += 1

    # If there are remaining elements in arr2, append them
    while j < len(arr2):
        result.append(arr2[j])
        j += 1

    return result

# Example usage:
array1 = [1, 3, 5, 7]
array2 = [2, 4, 6, 8]

merged_array = merge_sorted_arrays(array1, array2)
print("Merged Sorted Array:", merged_array)
```

In this example, the merge_sorted_arrays function uses two pointers (i and j) to traverse through the sorted arrays arr1 and arr2. It compares elements at the current positions and appends the smaller one to the result array. This process continues until one of the arrays is exhausted. Then, the remaining elements from the non-empty array are appended to the result.
This approach is efficient and has a time complexity of O(m + n), where m and n are the lengths of the two arrays. It's a common technique used in merging, intersecting, or comparing two sorted arrays.

c. Sliding Window

The sliding window technique involves maintaining a subset of elements within a "window" that slides through the given sequence. It is commonly used for solving problems where you need to find a subarray, substring, or a contiguous sequence of elements that satisfies a certain condition.
Let's consider an example where we find the maximum sum of a subarray of a fixed size k in an array:

```
def max_sum_subarray(nums, k):
    max_sum = float('-inf')
    current_sum = 0

    # Calculate the sum of the first k elements
    for i in range(k):
        current_sum += nums[i]
```

```
    # Slide the window through the array
    for i in range(k, len(nums)):
        max_sum = max(max_sum, current_sum)
        current_sum += nums[i] - nums[i - k]

    # Check the sum of the last window
    max_sum = max(max_sum, current_sum)

    return max_sum

# Example usage:
nums = [1, 4, 2, 10, 2, 3, 1, 0, 20]
k = 3

result = max_sum_subarray(nums, k)
print(f"The maximum sum of a subarray of size {k}: {result}")
```

This code defines a function to find the maximum sum of a subarray of a fixed size (k) using the sliding window technique. The window is slid through the array, and the maximum sum is updated accordingly.

d. Prefix Sum

The prefix sum technique involves precomputing the cumulative sum of elements up to each index in an array. This precomputation allows for quick calculation of the sum of any subarray by using the precomputed prefix sums. Let's consider an example where we use prefix sum to find the subarray with a target sum:

```
def subarray_with_target_sum(nums, target):
    prefix_sum = [0] * (len(nums) + 1)

    # Calculate prefix sum
    for i in range(1, len(nums) + 1):
        prefix_sum[i] = prefix_sum[i - 1] + nums[i - 1]

    # Find subarray with target sum
    for start in range(len(nums)):
        for end in range(start + 1, len(nums) + 1):
            current_sum = prefix_sum[end] - prefix_sum[start]
            if current_sum == target:
                return nums[start:end]

    return None
```

```
# Example usage:
nums = [1, 4, 2, 10, 2, 3, 1, 0, 20]
target_sum = 12

result = subarray_with_target_sum(nums, target_sum)
print(f"Subarray with target sum {target_sum}: {result}")
```

In this example, the subarray_with_target_sum function first calculates the prefix sum array, and then it iterates through all subarrays to find the one with the target sum. This approach has a time complexity of O(n^2) due to the nested loop, and it is not the most optimized solution. For a more efficient solution, you can use the two pointers or sliding window technique.
Prefix sum is particularly useful in scenarios where you need to calculate the sum of elements in a subarray multiple times, as it allows for constant-time calculations after the initial precomputation.

Build a prefix sum

```
def build_prefix_sum(nums):
    prefix_sum = [0] * (len(nums) + 1)

    for i in range(1, len(nums) + 1):
        prefix_sum[i] = prefix_sum[i - 1] + nums[i - 1]

    return prefix_sum

# Example usage:
nums = [1, 4, 2, 10, 2, 3, 1, 0, 20]

prefix_sum_array = build_prefix_sum(nums)
print("Prefix Sum Array:", prefix_sum_array)
```

In this example, the build_prefix_sum function takes a list of numbers (nums) and returns the corresponding prefix sum array. The prefix sum at index i is the sum of all elements from index 0 to i-1 in the original list.
The resulting prefix_sum_array will look like this:
[0, 1, 5, 7, 17, 19, 22, 23, 23, 43]

Each element prefix_sum[i] represents the sum of elements from nums[0] to nums[i-1]. The first element is always 0 to handle the case where we want the sum of elements up to index 0.

Hashing

a. Basics

Given n = dic.length,

- Add or remove key-value pair: O(1)
- Check if key exists: O(1)
- Check if value exists: O(n)
- Access or modify value associated with key: O(1)
- Iterate over all keys, values, or both: O(n)

Note: the O(1) operations are constant relative to n. In reality, the hashing algorithm might be expensive. For example, if your keys are strings, then it will cost O(m) where m is the length of the string. The operations only take constant time relative to the size of the hash map.

SETS
Given n = set.length,

- Add or remove element: O(1)
- Check if element exists: O(1)

Note: The O(1) operations are constant relative to n. The above note applies here as well.

```
def first_non_repeating_char(s):
    char_count = {}

    # Count the occurrences of each character
    for char in s:
        char_count[char] = char_count.get(char, 0) + 1

    # Find the first non-repeating character
    for char in s:
        if char_count[char] == 1:
            return char

    return None

# Example usage:
input_str = "programming"
result = first_non_repeating_char(input_str)
```

```
if result:
    print(f"The first non-repeating character in '{input_str}' is:
{result}")
else:
    print(f"There is no non-repeating character in '{input_str}'.")
```

This code defines a function to find the first non-repeating character in a string using a hash table (char_count). The first loop counts the occurrences of each character, and the second loop finds the first character with a count of 1, indicating it's non-repeating.

b. Checking for existence

```
def has_duplicates(nums):
    seen = set()

    for num in nums:
        if num in seen:
            return True
        seen.add(num)

    return False

# Example usage:
numbers = [1, 2, 3, 4, 5, 1]
if has_duplicates(numbers):
    print("The array has duplicates.")
else:
    print("No duplicates found in the array.")
```

In this code, the has_duplicates function uses a hash set (seen) to keep track of unique elements encountered while iterating through the array. If it encounters a number that is already in the set, it returns True, indicating the presence of duplicates. Otherwise, it adds the number to the set.

c. Counting

```
def count_elements(nums):
    element_count = {}

    # Count the occurrences of each element
    for num in nums:
```

```
        element_count[num] = element_count.get(num, 0) + 1

    return element_count

# Example usage:
numbers = [1, 2, 3, 4, 1, 2, 2, 3, 4, 5]
result = count_elements(numbers)
print("Element counts:")
for num, count in result.items():
    print(f"{num}: {count}")
```

In this code, the count_elements function uses a hash table (element_count) to store the count of each unique element in the array. The loop iterates through the array, updating the count for each element. The final result is a dictionary where keys are elements, and values are their respective counts.

Linked Lists

a. Basics

Given n as the number of nodes in the linked list:

- Add or remove element given pointer before add/removal location: O(1)
- Add or remove element given pointer at add/removal location: O(1) if doubly linked
- Add or remove element at arbitrary position without pointer: O(n)
- Access element at arbitrary position without pointer: O(n)
- Check if element exists: O(n)
- Reverse between position i and j: O(j-1)
- Detect a cycle: O(n) using fast-slow pointers or hash map

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
```

```
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage:
my_list = LinkedList()
my_list.append(1)
my_list.append(2)
my_list.append(3)
my_list.append(4)

print("Linked List:")
my_list.display()
```

In this example, I've defined a Node class to represent each element in the linked list and a LinkedList class to handle the operations. The append method adds a new node to the end of the list, and the display method traverses the list and prints its elements.

b. Fast and slow pointers

The fast and slow pointers technique is often used with linked lists to detect cycles or find the middle element efficiently. Here's an example of using fast and slow pointers to detect a cycle in a linked list:

```
class ListNode:
    def __init__(self, value=0, next=None):
        self.value = value
        self.next = next

def has_cycle(head):
    if not head or not head.next:
        return False
```

```
    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next

    return True

# Example usage:
# Create a linked list with a cycle
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node5 = ListNode(5)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5
node5.next = node2  # Cycle: 5 -> 2

result = has_cycle(node1)
print("Has Cycle:", result)
```

In this example, the has_cycle function uses two pointers, slow and fast, initially pointing to the head of the linked list. The slow pointer advances one node at a time, while the fast pointer advances two nodes at a time. If there is a cycle in the linked list, the fast pointer will eventually catch up with the slow pointer. If there is no cycle, the fast pointer will reach the end of the list.
This technique is efficient and has a time complexity of O(n), where n is the number of nodes in the linked list. It's widely used in algorithms related to detecting cycles and finding the middle element in a linked list.

c. Reversing a linked list

Reversing a linked list is a classic algorithmic problem. Here's an example of how you can reverse a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def reverse(self):
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

# Example usage:
my_list = LinkedList()
my_list.append(1)
my_list.append(2)
my_list.append(3)
```

```
my_list.append(4)

print("Original Linked List:")
my_list.display()

my_list.reverse()

print("\nReversed Linked List:")
my_list.display()
```

In this example, the reverse method of the LinkedList class reverses the linked list by iteratively updating the next pointers of each node. The prev, current, and next_node pointers are used to perform the reversal.

Stacks and queues

Find number of subarrays that fit an exact criteria

Let's consider a scenario where you have an array of numbers, and you want to find the number of subarrays that fit a specific criteria. For example, let's say you want to find the number of subarrays where the sum is equal to a given target value.
Here's an example using a stack to efficiently find the number of subarrays with a specific sum:

```
def count_subarrays_with_sum(nums, target_sum):
    count = 0
    current_sum = 0
    sum_frequency = {0: 1}  # Initialize with 0 to handle subarrays
starting from index 0

    for num in nums:
        current_sum += num
        if current_sum - target_sum in sum_frequency:
            count += sum_frequency[current_sum - target_sum]
        sum_frequency[current_sum] = sum_frequency.get(current_sum, 0) +
1

    return count

# Example usage:
nums = [1, 2, 3, 0, 1, 2]
target_sum = 3
```

```
result = count_subarrays_with_sum(nums, target_sum)
print(f"Number of subarrays with sum {target_sum}: {result}")
```

In this example, the count_subarrays_with_sum function uses a stack to efficiently find the number of subarrays with the given sum. It maintains a running sum (current_sum) and a dictionary (sum_frequency) to store the frequency of encountered sums. If the difference between the current sum and the target sum is present in the dictionary, it means we have found a subarray with the desired sum.
This algorithm has a time complexity of O(n), where n is the length of the input array. It efficiently counts the number of subarrays that meet the specified criteria.

a. Stacks

Stack operations are dependent on their implementation. A stack is only required to support pop and push. If implemented with a dynamic array:

Given n = stack.length,

- Push element: O(1)
- Pop element: O(1)
- Peek (see element at top of stack): O(1)
- Access or modify element at arbitrary index: O(1)
- Check if element exists: O(n)

Let's create a simple stack and perform some basic operations like pushing and popping elements.

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            print("Stack is empty.")
```

```
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty.")

    def size(self):
        return len(self.items)

# Example usage:
my_stack = Stack()

my_stack.push(1)
my_stack.push(2)
my_stack.push(3)

print("Stack:")
print("Peek:", my_stack.peek())
print("Size:", my_stack.size())

popped_item = my_stack.pop()
print("Popped item:", popped_item)

print("Stack after popping:")
print("Peek:", my_stack.peek())
print("Size:", my_stack.size())
```

In this example, I've defined a Stack class with basic stack operations. The push method adds an item to the stack, the pop method removes and returns the top item, the peek method returns the top item without removing it, and the size method returns the size of the stack.

b. String problems

Let's tackle a classic problem involving a stack and strings. How about checking if a given string containing parentheses is balanced? Here's an example:

```
def is_balanced(expression):
    stack = []
    opening_brackets = "({["
    closing_brackets = ")}]"

    for char in expression:
        if char in opening_brackets:
```

```
            stack.append(char)
        elif char in closing_brackets:
            if not stack or
opening_brackets[closing_brackets.index(char)] != stack.pop():
                return False

    return not stack

# Example usage:
balanced_str = "{[()]}"

if is_balanced(balanced_str):
    print(f"The string '{balanced_str}' is balanced.")
else:
    print(f"The string '{balanced_str}' is not balanced.")
```

In this example, the is_balanced function uses a stack to check if a given string with parentheses (and other brackets) is balanced. It iterates through the string, pushing opening brackets onto the stack and popping from the stack when a closing bracket is encountered. The string is balanced if, at the end, the stack is empty.

c. Queues

Queue operations are dependent on their implementation. A queue is only required to support dequeue and enqueue. If implemented with a doubly linked list:

Given n = queue.length,

- Enqueue element: O(1)
- Dequeue element: O(1)
- Peek (see element at front of queue): O(1)
- Access or modify element at arbitrary index: O(n)
- Check if element exists: O(n)

Note: Most programming languages implement queues in a more sophisticated manner than a simple doubly linked list. Depending on implementation, accessing elements by index may be faster than O(n), or O(n) but with a significant constant divisor.

Let's create a simple queue and perform some basic operations like enqueue and dequeue.

```
class Queue:
    def __init__(self):
```

```
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            print("Queue is empty.")

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            print("Queue is empty.")

    def size(self):
        return len(self.items)

# Example usage:
my_queue = Queue()

my_queue.enqueue(1)
my_queue.enqueue(2)
my_queue.enqueue(3)

print("Queue:")
print("Peek:", my_queue.peek())
print("Size:", my_queue.size())

dequeued_item = my_queue.dequeue()
print("Dequeued item:", dequeued_item)

print("Queue after dequeuing:")
print("Peek:", my_queue.peek())
print("Size:", my_queue.size())
```

In this example, I've defined a Queue class with basic queue operations. The enqueue method adds an item to the end of the queue, the dequeue method removes and returns the front item, the peek method returns the front item without removing it, and the size method returns the size of the queue.

d. Monotonic

Monotonic increasing stack

Let's implement a function to check if an array is monotonic, meaning it's entirely non-increasing or non-decreasing.

```
def is_monotonic(nums):
    increasing = decreasing = True

    for i in range(1, len(nums)):
        if nums[i] > nums[i - 1]:
            decreasing = False
        elif nums[i] < nums[i - 1]:
            increasing = False

    return increasing or decreasing

# Example usage:
monotonic_array1 = [1, 2, 2, 3]
monotonic_array2 = [3, 2, 1, 0]

print(f"Is {monotonic_array1} monotonic?
{is_monotonic(monotonic_array1)}")
print(f"Is {monotonic_array2} monotonic?
{is_monotonic(monotonic_array2)}")
```

In this example, the is_monotonic function iterates through the array and checks if it's either entirely non-increasing or non-decreasing. The increasing and decreasing variables are used to keep track of the monotonicity.

Trees and graphs

a. Binary trees

Given n as the number of nodes in the tree,

Most algorithms will run in $O(n \cdot k)$ time, where k is the work done at each node, usually O(1). This is just a general rule and not always the case. We are assuming here that BFS is implemented with an efficient queue.

Let's create a simple binary tree and perform a basic traversal, say, an in-order traversal.

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

def in_order_traversal(root):
    result = []
    if root:
        result.extend(in_order_traversal(root.left))
        result.append(root.value)
        result.extend(in_order_traversal(root.right))
    return result

# Example usage:
# Construct a simple binary tree:   1
#                                  /   \
#                                 2     3
#                                / \
#                               4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

print("In-order traversal:", in_order_traversal(root))
```

For this example, I've defined a TreeNode class to represent each node in the binary tree, and the in_order_traversal function performs an in-order traversal, returning a list of values in the order they are visited.

b. Binary trees - DFS

Recursive:

Depth-First Search (DFS) is a traversal algorithm used to explore or visit all the nodes of a tree or graph along a particular branch before backtracking. In the case of binary trees, DFS can be implemented using recursion. Here's an example of a recursive DFS traversal on a binary tree:

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

```
        self.right = None

def dfs_recursive(node):
    if node is not None:
        # Process the current node (e.g., print its value)
        print(node.value)

        # Recursive DFS on the left subtree
        dfs_recursive(node.left)

        # Recursive DFS on the right subtree
        dfs_recursive(node.right)

# Example usage:
# Construct a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Perform DFS traversal recursively
print("DFS Recursive:")
dfs_recursive(root)
```

In this example, the dfs_recursive function takes a binary tree node as an argument and performs a depth-first traversal starting from that node. The traversal process involves visiting the current node, recursively traversing the left subtree, and then recursively traversing the right subtree.

This recursive approach elegantly captures the structure of the depth-first traversal. The order in which nodes are processed depends on whether you choose to visit the left subtree before the right subtree or vice versa.

DFS is widely used in various tree-related problems, and the recursive approach is intuitive and easy to implement. However, keep in mind that for very deep trees, recursive solutions may lead to a stack overflow due to the recursive call stack. In practice, for deep trees, an iterative solution using a stack might be preferred.

Iterative:

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

def dfs_iterative(root):
    if not root:
        return
```

```
    stack = [root]

    while stack:
        current_node = stack.pop()

        # Process the current node (e.g., print its value)
        print(current_node.value)

        # Push right child first (if exists) to ensure left child is
processed first
        if current_node.right:
            stack.append(current_node.right)

        # Push left child (if exists)
        if current_node.left:
            stack.append(current_node.left)

# Example usage:
# Construct a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Perform DFS traversal iteratively
print("DFS Iterative:")
dfs_iterative(root)
```

This example, the dfs_iterative function uses a stack to perform an iterative depth-first traversal of the binary tree. The stack starts with the root node, and in each iteration, a node is popped from the stack, and its children (if any) are pushed onto the stack.

This iterative approach is often preferred for deep trees to avoid potential stack overflow issues associated with recursive solutions. It also allows for more control over the order in which nodes are processed.

Note: The order of pushing right and left children onto the stack determines whether you want to visit the left subtree before the right subtree or vice versa. In this example, the right child is pushed first to ensure that the left child is processed first. Adjust the order based on your specific requirements.

c. Binary trees - BFS

Breadth-First Search (BFS) is a traversal algorithm that explores all the nodes of a tree or graph level by level, visiting all the neighbors of a node before moving on to the next level. Here's an example of BFS on a binary tree:

```
from collections import deque

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

def bfs(tree_root):
    if not tree_root:
        return

    queue = deque([tree_root])

    while queue:
        current_node = queue.popleft()

        # Process the current node (e.g., print its value)
        print(current_node.value)

        # Enqueue left child (if exists)
        if current_node.left:
            queue.append(current_node.left)

        # Enqueue right child (if exists)
        if current_node.right:
            queue.append(current_node.right)

# Example usage:
# Construct a binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Perform BFS traversal
print("BFS:")
```

```
bfs(root)
```

In this example, the bfs function uses a queue to perform a breadth-first traversal of the binary tree. The algorithm starts from the root, processes each node, and enqueues its children. This process continues until the queue is empty.

BFS is often used when you need to find the shortest path between two nodes, or when you want to visit nodes level by level. It ensures that you visit all nodes at a given level before moving on to the next level.

The deque from the collections module is used to efficiently implement the queue for BFS.

d. Binary search trees

Given n as the number of nodes in the tree,

- Add or remove element: O(n) worst case, O(log n) average case
- Check if element exists: O(n) worst case, O(log n) average case

The average case is when the tree is well balanced - each depth is close to full. The worst case is when the tree is just a straight line.

Let's create a simple binary search tree (BST) and perform some basic operations like insertion and in-order traversal.

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

def insert_bst(root, value):
    if not root:
        return TreeNode(value)

    if value < root.value:
        root.left = insert_bst(root.left, value)
    else:
        root.right = insert_bst(root.right, value)

    return root
```

```
def in_order_traversal(root):
    result = []
    if root:
        result.extend(in_order_traversal(root.left))
        result.append(root.value)
        result.extend(in_order_traversal(root.right))
    return result

# Example usage:
# Construct a binary search tree:
#        4
#       / \
#      2   5
#     / \
#    1   3
bst_root = None
values_to_insert = [4, 2, 5, 1, 3]

for value in values_to_insert:
    bst_root = insert_bst(bst_root, value)

print("In-order traversal of the BST:", in_order_traversal(bst_root))
```

In this example, the insert_bst function inserts a value into the binary search tree while maintaining its property that for each node, all nodes in its left subtree have values less than the node's value, and all nodes in its right subtree have values greater than the node's value. The in_order_traversal function is then used to perform an in-order traversal of the BST.

e. Graphs

Let's create a simple directed graph and perform a basic depth-first traversal.

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, node, visited):
        if node not in visited:
```

```
            print(node, end=" ")
            visited.add(node)
            for neighbor in self.graph[node]:
                self.dfs(neighbor, visited)

# Example usage:
my_graph = Graph()

# Adding edges to the graph
my_graph.add_edge(1, 2)
my_graph.add_edge(1, 3)
my_graph.add_edge(2, 4)
my_graph.add_edge(2, 5)
my_graph.add_edge(3, 6)

print("Depth-First Traversal starting from node 1:")
my_graph.dfs(1, set())
```

In this example, the Graph class uses an adjacency list to represent a directed graph. The add_edge method adds edges to the graph, and the dfs method performs a depth-first traversal starting from a given node. The set visited is used to keep track of visited nodes.

f. Graphs - DFS

Recursive

Depth-First Search (DFS) can also be implemented recursively for graph traversal. In a graph, nodes may have multiple neighbors, so it's common to use a visited set to keep track of nodes that have already been visited. Here's an example of recursive DFS on a graph:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

def dfs_recursive(graph, node, visited):
    if node not in visited:
        # Process the current node (e.g., print its value)
        print(node)

        visited.add(node)
```

```
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited)

# Example usage:
# Construct a graph
graph = Graph()
graph.add_edge(1, [2, 3])
graph.add_edge(2, [4, 5])
graph.add_edge(3, [6])
graph.add_edge(4, [])
graph.add_edge(5, [7])
graph.add_edge(6, [])
graph.add_edge(7, [])

# Set to keep track of visited nodes
visited_set = set()

# Perform DFS traversal recursively
print("DFS Recursive:")
dfs_recursive(graph.graph, 1, visited_set)
```

In this example, the Graph class represents a graph using an adjacency list, and dfs_recursive is a recursive function that performs depth-first traversal on the graph starting from a given node. The visited_set is used to keep track of visited nodes to avoid infinite loops in case of cycles.

The order in which nodes are processed depends on the order in which they appear in the adjacency list of the graph. Recursive DFS is elegant and easy to understand but keep in mind that it might lead to a maximum recursion depth if the graph is deep. In practice, for deep graphs, an iterative solution using a stack might be preferred.

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

def dfs_iterative(graph, start_node):
    visited = set()
    stack = [start_node]

    while stack:
        current_node = stack.pop()
```

```
        if current_node not in visited:
            # Process the current node (e.g., print its value)
            print(current_node)

            visited.add(current_node)

            # Push unvisited neighbors onto the stack
            stack.extend(neighbor for neighbor in graph[current_node] if
neighbor not in visited)

# Example usage:
# Construct a graph
graph = Graph()
graph.add_edge(1, [2, 3])
graph.add_edge(2, [4, 5])
graph.add_edge(3, [6])
graph.add_edge(4, [])
graph.add_edge(5, [7])
graph.add_edge(6, [])
graph.add_edge(7, [])

# Perform DFS traversal iteratively
print("DFS Iterative:")
dfs_iterative(graph.graph, 1)
```

In this example, the Graph class represents a graph using an adjacency list, and dfs_iterative is an iterative function that performs depth-first traversal starting from a given node. The stack is used to keep track of nodes to be visited. The order in which nodes are processed depends on the order in which they are popped from the stack.

This iterative approach is often preferred for deep graphs to avoid potential maximum recursion depth issues associated with recursive solutions. It also allows for more control over the order in which nodes are processed.

g. Graphs - BFS

Breadth-First Search (BFS) is a traversal algorithm that explores all the nodes of a graph level by level. Here's an example of an iterative BFS implementation for a graph using a queue:

```
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])

    while queue:
        current_node = queue.popleft()

        if current_node not in visited:
            # Process the current node (e.g., print its value)
            print(current_node)

            visited.add(current_node)

            # Enqueue unvisited neighbors
            queue.extend(neighbor for neighbor in graph[current_node] if
neighbor not in visited)

# Example usage:
# Construct a graph
graph = Graph()
graph.add_edge(1, [2, 3])
graph.add_edge(2, [4, 5])
graph.add_edge(3, [6])
graph.add_edge(4, [])
graph.add_edge(5, [7])
graph.add_edge(6, [])
graph.add_edge(7, [])

# Perform BFS traversal
print("BFS Iterative:")
bfs(graph.graph, 1)
```

In this example, the Graph class represents a graph using an adjacency list, and bfs is an iterative function that performs breadth-first traversal starting from a given node. The deque from the collections module is used to efficiently implement the queue for BFS. BFS is often used when you need to find the shortest path between two nodes or when you want to visit nodes level by level. It guarantees the shortest path in an unweighted

graph. The order in which nodes are processed depends on the order in which they are dequeued from the queue.

h. Implicit Graphs

Implicit graphs are interesting! They don't store all the edges explicitly; instead, the edges are defined by a rule or formula. Let's consider a classic example: the knight's tour problem on a chessboard. The goal is to find a sequence of moves for a knight on an N×N chessboard such that the knight visits every square exactly once.

```
def is_valid_move(x, y, board, N):
    return 0 <= x < N and 0 <= y < N and board[x][y] == -1

def knight_tour(N):
    # Initialize the chessboard
    board = [[-1 for _ in range(N)] for _ in range(N)]

    # Possible moves for a knight
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1),
             (-2, -1), (-1, -2), (1, -2), (2, -1)]

    # Start the tour from the top-left corner
    board[0][0] = 0

    # Helper function for recursive backtracking
    def knight_tour_util(x, y, move_count):
        if move_count == N * N - 1:
            return True

        for move in moves:
            next_x, next_y = x + move[0], y + move[1]
            if is_valid_move(next_x, next_y, board, N):
                board[next_x][next_y] = move_count
                if knight_tour_util(next_x, next_y, move_count + 1):
                    return True
                board[next_x][next_y] = -1  # Backtrack if the current
move doesn't lead to a solution

        return False

    # Start the recursive backtracking
    if not knight_tour_util(0, 0, 1):
```

```
            print("Solution does not exist.")
            return

        # Print the solution
        for row in board:
            print(row)

# Example usage:
chessboard_size = 5
knight_tour(chessboard_size)
```

In this example, the knight_tour function solves the knight's tour problem using recursive backtracking. The chessboard is represented by a 2D list, and the knight makes valid moves according to the rules of chess. The knight_tour_util function is a recursive helper function that explores all possible moves, backtracking when necessary. The solution, if it exists, is printed as a sequence of knight moves on the chessboard.

Heaps

a. Basics

Given n = heap.length and talking about min heaps,

- Add an element: O(log n)
- Delete the minimum element: O(log n)
- Find the minimum element: O(1)
- Check if element exists: O(n)

Let's create a simple min-heap and perform basic operations like insertion and extraction of the minimum element.

```
import heapq

class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, value):
        heapq.heappush(self.heap, value)

    def extract_min(self):
        if not self.is_empty():
```

```
            return heapq.heappop(self.heap)
        else:
            print("Heap is empty.")
            return None

    def is_empty(self):
        return len(self.heap) == 0

# Example usage:
my_heap = MinHeap()

# Insert elements into the heap
my_heap.insert(4)
my_heap.insert(2)
my_heap.insert(8)
my_heap.insert(1)

print("Min-heap:")
while not my_heap.is_empty():
    min_element = my_heap.extract_min()
    print(min_element)
```

In this example, the MinHeap class uses the heapq module in Python, which provides an implementation of a binary heap. The insert method adds an element to the heap, and the extract_min method removes and returns the minimum element. The is_empty method checks if the heap is empty.

b. Top k

Find top k elements with heap

To find the top k elements in a collection efficiently, you can use a heap data structure. A max heap is suitable for this task because it allows for quick retrieval of the maximum element. Here's an example using Python's built-in heapq module to find the top k elements:

```
import heapq

def find_top_k_elements(nums, k):
    # Convert the input list to a min heap
    min_heap = nums[:k]
    heapq.heapify(min_heap)

    # Iterate through the rest of the elements
```

```
    for num in nums[k:]:
        # If the current element is larger than the smallest element in
the heap
        if num > min_heap[0]:
            # Replace the smallest element with the current element
            heapq.heappop(min_heap)
            heapq.heappush(min_heap, num)

    # The top k elements are now in the min heap
    return sorted(min_heap, reverse=True)

# Example usage:
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
k = 3

top_k_elements = find_top_k_elements(numbers, k)
print(f"Top {k} elements:", top_k_elements)
```

In this example, find_top_k_elements initializes a min heap with the first k elements from the input list (nums). The heap is then iteratively updated by replacing the smallest element in the heap with any larger element encountered in the remaining elements of the list.

Finally, the top k elements are retrieved from the heap and returned. Note that the elements are sorted in reverse order to get them in descending order.
This approach has a time complexity of O(n * log(k)), where n is the size of the input list. It's efficient for finding the top k elements when k is much smaller than n.

Greedy

a. Basics

Greedy algorithms make decisions at each step by choosing the locally optimal solution, with the hope that these choices will lead to a globally optimal solution. Let's implement a simple greedy algorithm for the coin change problem.

```
def greedy_coin_change(coins, target_amount):
    coins.sort(reverse=True)  # Sort coins in descending order

    change = []
    remaining_amount = target_amount
```

```
    for coin in coins:
        while remaining_amount >= coin:
            change.append(coin)
            remaining_amount -= coin

    if remaining_amount == 0:
        return change
    else:
        print("Greedy solution does not exist.")
        return None

# Example usage:
available_coins = [25, 10, 5, 1]
target_amount = 63

result = greedy_coin_change(available_coins, target_amount)

if result:
    print(f"Greedy coin change for {target_amount} cents: {result}")
```

In this example, the greedy_coin_change function uses a greedy approach to make change for a given target amount using a set of available coin denominations. It sorts the coins in descending order and iteratively selects the largest possible coin at each step until the target amount is reached. If the target amount is not reached, it prints a message indicating that a greedy solution does not exist.

Binary Search

a. Basics

Binary search runs in O(log n) in the worst case, where n is the size of your initial search space.

Binary search is an efficient algorithm for finding a target value within a sorted array.

```
def binary_search(nums, target):
    low, high = 0, len(nums) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_value = nums[mid]

        if mid_value == target:
```

```
            return mid  # Target found, return the index
        elif mid_value < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1  # Target not found

# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target_value = 7

result = binary_search(sorted_array, target_value)

if result != -1:
    print(f"Target {target_value} found at index {result}")
else:
    print(f"Target {target_value} not found in the array")
```

In this example, binary_search takes a sorted array (nums) and a target value (target). The algorithm initializes two pointers, low and high, representing the range of the array to search. It iteratively narrows down the search range by comparing the middle element (mid_value) with the target.

If the middle element is equal to the target, the search is successful, and the index is returned. If the middle element is less than the target, the search continues in the right half of the array. If the middle element is greater than the target, the search continues in the left half of the array.

The algorithm repeats these steps until the target is found or the search range becomes empty (low > high), indicating that the target is not present in the array.
Binary search has a time complexity of O(log n), making it significantly more efficient than linear search for large datasets.

Binary search: duplicate elements, left-most insertion point

When dealing with duplicate elements in a sorted array and you want to find the left-most insertion point (the first occurrence or the position where a new element should be inserted to maintain sorted order), you can modify the binary search algorithm accordingly. Here's an example in Python:

```
def leftmost_insertion_point(nums, target):
    low, high = 0, len(nums)

    while low < high:
        mid = (low + high) // 2
        mid_value = nums[mid]

        if mid_value < target:
            low = mid + 1
        else:
            high = mid

    return low

# Example usage:
sorted_array = [1, 2, 2, 2, 3, 4, 4, 5, 6]
target_value = 4

insertion_point = leftmost_insertion_point(sorted_array, target_value)

print(f"Left-most insertion point for {target_value}:
{insertion_point}")
```

In this modified version, the key change is in the termination condition of the while loop. The loop continues as long as low is less than high (not low <= high). This ensures that the search range is reduced to a single element when the insertion point is found.

The logic inside the loop is adjusted so that when the middle element (mid_value) is less than the target, the search continues in the right half by updating low to mid + 1. If the middle element is greater than or equal to the target, the search continues in the left half by updating high to mid.

The final low value gives the left-most insertion point for the target in the sorted array. If the target is already present in the array, this will be the left-most occurrence. If the target is not present, it will be the position where the target should be inserted to maintain sorted order.

Binary search: duplicate elements, right-most insertion point

```
def rightmost_insertion_point(nums, target):
    low, high = 0, len(nums)

    while low < high:
        mid = (low + high) // 2
        mid_value = nums[mid]
```

```
        if mid_value <= target:
            low = mid + 1
        else:
            high = mid

    return low

# Example usage:
sorted_array = [1, 2, 2, 2, 3, 4, 4, 5, 6]
target_value = 4

insertion_point = rightmost_insertion_point(sorted_array, target_value)

print(f"Right-most insertion point for {target_value}:
{insertion_point}")
```

In this modified version, the key change is in the logic when the middle element (mid_value) is less than or equal to the target. If the middle element is less than or equal to the target, the search continues in the right half by updating low to mid + 1. If the middle element is greater than the target, the search continues in the left half by updating high to mid.

The final low value gives the right-most insertion point for the target in the sorted array. If the target is already present in the array, this will be the right-most occurrence. If the target is not present, it will be the position where the target should be inserted to maintain sorted order.

Binary search: greedy problems

Binary search is often used in solving greedy problems where you need to optimize a certain criteria, and you can make a decision to move left or right based on that criteria. The decision at each step is typically based on whether a certain condition is satisfied or not.

Let's consider an example of a greedy problem where binary search can be applied. Suppose you have a list of elements representing the costs of building houses, and you want to find the minimum budget needed to build a certain number of houses. The cost of building each house is given in the array costs, and you have a budget target_budget. You want to build as many houses as possible within the budget. The cost array is sorted in ascending order.

```
def binary_search_for_houses(costs, target_budget):
    low, high = 0, len(costs) - 1
```

```
    while low <= high:
        mid = (low + high) // 2
        mid_cost = costs[mid]

        if mid_cost <= target_budget:
            low = mid + 1
        else:
            high = mid - 1

    return high  # Right-most index where cost is within budget

# Example usage:
costs = [1, 2, 3, 4, 5]
target_budget = 8

max_houses = binary_search_for_houses(costs, target_budget)
print(f"The maximum number of houses within the budget is: {max_houses +
1}")
```

In this example, the binary search function binary_search_for_houses searches for the right-most index where the cost of building a house is within the budget. The result gives you the maximum number of houses you can build within the budget.

This is a common pattern in greedy problems where you need to make decisions about whether to move left or right based on a certain condition. The binary search helps efficiently find the right-most index satisfying the condition.

If looking for a maximum

When using binary search to find a maximum value in a certain range, the key idea is similar to regular binary search, but the condition for moving left or right is reversed. Instead of searching for the exact match or the right-most occurrence, you're looking for the maximum value. Here's an example:

Let's say you have a sorted array of integers, and you want to find the maximum element that satisfies a certain condition.

```
def binary_search_for_maximum(nums):
    low, high = 0, len(nums) - 1

    while low < high:
        mid = (low + high + 1) // 2  # Ensure mid is biased to the right
        mid_value = nums[mid]

        if condition_satisfied(mid_value):
            low = mid
        else:
            high = mid - 1

    return nums[low]
```

```
# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# Assume the condition is that the value should be less than or equal to
a certain threshold
threshold = 6

maximum_value = binary_search_for_maximum(sorted_array, threshold)
print(f"The maximum value less than or equal to {threshold} is:
{maximum_value}")
```

In this example, the condition_satisfied function represents the condition you want to check. The binary search is adjusted to move left or right based on whether the condition is satisfied or not. The mid is biased to the right to make sure the search space is reduced in each step.

This pattern is useful in various scenarios, such as finding the maximum element that satisfies a specific condition in a sorted array or making decisions based on maximizing a certain objective function.

b. On Arrays

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            low = mid + 1  # Search the right half
        else:
            high = mid - 1  # Search the left half

    return -1  # Target not found

# Example usage:
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target_element = 8

result = binary_search(sorted_array, target_element)

if result != -1:
    print(f"Element {target_element} found at index {result}.")
else:
```

```
    print(f"Element {target_element} not found in the array.")
```

In this example, the binary_search function takes a sorted array and a target element as input and returns the index of the target element if it exists in the array. The search range is narrowed down by adjusting the low and high pointers until the target is found or the search range becomes empty.

c. On solution spaces

Binary search can also be applied to solution spaces where the solutions have a certain order or property. Let's consider an example where we need to find the square root of a given number with a certain precision using binary search.

```
def binary_search_sqrt(target, precision):
    if target < 0:
        return None  # Square root of a negative number is undefined

    low, high = 0, target

    while low <= high:
        mid = (low + high) / 2
        mid_squared = mid * mid

        if abs(mid_squared - target) < precision:
            return mid  # Found a close enough approximation

        elif mid_squared < target:
            low = mid + precision  # Search the right half

        else:
            high = mid - precision  # Search the left half

    return None  # Square root not found within the precision

# Example usage:
target_number = 9
precision = 0.001

result = binary_search_sqrt(target_number, precision)

if result is not None:
    print(f"The square root of {target_number} is approximately
{result:.3f}.")
else:
    print(f"Square root not found within the precision.")
```

In this example, the binary_search_sqrt function uses binary search to find an approximation of the square root of a given number. The precision is controlled by adjusting the search range based on the comparison of the square of the current midpoint with the target number. This allows us to find an approximate solution within the specified precision.

Backtracking

a. Basics

Backtracking is a powerful technique for solving problems by exploring all possible solutions and "backtracking" when a dead-end is reached. Let's implement a backtracking algorithm for solving the N-Queens problem.

```
def is_safe(board, row, col, n):
    # Check if there is a queen in the same row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check if there is a queen in the upper diagonal on the left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check if there is a queen in the lower diagonal on the left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens_util(board, col, n):
    if col == n:
        return [board[i][:] for i in range(n)]

    result = []
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            solutions = solve_n_queens_util(board, col + 1, n)
            if solutions:
                result.extend(solutions)
```

```
            board[i][col] = 0  # Backtrack

    return result

def solve_n_queens(n):
    board = [[0] * n for _ in range(n)]
    return solve_n_queens_util(board, 0, n)

# Example usage:
n_queens_solutions = solve_n_queens(4)

if n_queens_solutions:
    print(f"Solutions for 4-Queens Problem:")
    for solution in n_queens_solutions:
        for row in solution:
            print(row)
        print()
else:
    print("No solution found.")
```

For this example, the solve_n_queens function uses backtracking to find all solutions to the N-Queens problem. The is_safe function checks if placing a queen in a certain position is safe, and the solve_n_queens_util function is a recursive backtracking function that explores all possible placements of queens on the chessboard. The solutions, if found, are returned as a list of chessboards.

b. Generation

```
def generate_permutations(nums):
    def backtrack(start):
        if start == len(nums) - 1:
            result.append(nums[:])  # Add a copy of the current
permutation
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]  # Swap elements
            backtrack(start + 1)  # Recur for the next position
            nums[start], nums[i] = nums[i], nums[start]  # Backtrack
(undo the swap)

    result = []
    backtrack(0)
```

```
    return result

# Example usage:
input_nums = [1, 2, 3]
permutations = generate_permutations(input_nums)

if permutations:
    print(f"Permutations of {input_nums}:")
    for perm in permutations:
        print(perm)
else:
    print("No permutations found.")
```

In this example, the generate_permutations function uses backtracking to generate all permutations of a given set of numbers. The backtrack function is a recursive backtracking function that swaps elements at different positions to explore all possible permutations. The generated permutations are stored in the result list.

c. More constrained backtracking

Let's implement a backtracking algorithm for generating all valid combinations of k elements from a set of n elements.

```
def generate_combinations(n, k):
    def backtrack(start, current_combination):
        if len(current_combination) == k:
            result.append(current_combination[:])  # Add a copy of the
current combination
            return

        for i in range(start, n + 1):
            current_combination.append(i)
            backtrack(i + 1, current_combination)  # Recur for the next
position
            current_combination.pop()  # Backtrack (remove the last
element)

    result = []
    backtrack(1, [])
    return result

# Example usage:
n_elements = 4
k_combinations = 2
```

```
combinations = generate_combinations(n_elements, k_combinations)

if combinations:
    print(f"Combinations of {k_combinations} elements from
{n_elements}:")
    for combination in combinations:
        print(combination)
else:
    print("No combinations found.")
```

In this example, the generate_combinations function uses backtracking to generate all valid combinations of k elements from a set of n elements. The backtrack function is a recursive backtracking function that explores all possible combinations by choosing elements at different positions and backtracking when needed. The generated combinations are stored in the result list.

Dynamic programming

a. Basics

Top down memoization

Dynamic programming with memoization is a powerful technique for optimizing recursive algorithms by storing and reusing intermediate results. Let's implement a top-down memoization approach for solving the Fibonacci sequence.

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]

    if n <= 1:
        result = n
    else:
        result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)

    memo[n] = result
    return result

# Example usage:
n = 10
result = fibonacci(n)
```

```
print(f"The {n}-th Fibonacci number is: {result}")
```

In this example, the fibonacci function uses top-down memoization to calculate the n-th Fibonacci number. The memo dictionary is used to store previously computed Fibonacci numbers, preventing redundant calculations and significantly improving the efficiency of the recursive algorithm.

b. Framework for DP

Let's create a simple framework for solving dynamic programming problems using a bottom-up approach. We'll use a table to store intermediate results to avoid redundant computations.

```
def dynamic_programming_framework(target):
    # Initialize a table to store intermediate results
    dp_table = [0] * (target + 1)

    # Base case
    dp_table[0] = 1

    # Fill in the table in a bottom-up manner
    for i in range(1, target + 1):
        # Update the table based on the recurrence relation
        dp_table[i] += dp_table[i - 1]

        if i >= 2:
            dp_table[i] += dp_table[i - 2]

        if i >= 3:
            dp_table[i] += dp_table[i - 3]

    # The final result is stored in the last entry of the table
    return dp_table[target]

# Example usage:
target_value = 5
result = dynamic_programming_framework(target_value)

print(f"Number of ways to reach {target_value}: {result}")
```

In this framework, we use a bottom-up approach to fill in the dp_table by iteratively updating each entry based on the recurrence relation. The final result is stored in the last entry of the table. You can customize this framework for different dynamic

programming problems by adjusting the base case, recurrence relation, and the size of the table based on the problem's requirements.

c. 1D Problems

Let's create a dynamic programming framework for solving 1D dynamic programming problems. We'll use a 1D list to store intermediate results.

```
def dp_1d_framework(nums):
    n = len(nums)

    # Initialize a 1D list to store intermediate results
    dp = [0] * n

    # Base case
    dp[0] = nums[0]

    # Fill in the list in a bottom-up manner
    for i in range(1, n):
        # Update the list based on the recurrence relation
        dp[i] = max(dp[i - 1] + nums[i], nums[i])

    # The final result is the maximum value in the list
    return max(dp)

# Example usage:
input_nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
result = dp_1d_framework(input_nums)

print(f"The maximum sum of a subarray is: {result}")
```

In this framework, we use a bottom-up approach to fill in the dp list. The base case is initialized, and then we iteratively update each entry based on the recurrence relation. The final result is the maximum value in the list, representing the solution to the specific 1D dynamic programming problem. You can adapt this framework for different 1D problems by adjusting the base case and recurrence relation accordingly.

d. Matrix DP

Let's create a dynamic programming framework for solving problems involving 2D matrices. We'll use a 2D list to store intermediate results.

```
def dp_matrix_framework(matrix):
    rows, cols = len(matrix), len(matrix[0])
```

```
    # Initialize a 2D list to store intermediate results
    dp = [[0] * cols for _ in range(rows)]

    # Base cases (initialize the first row and column)
    dp[0][0] = matrix[0][0]

    for i in range(1, rows):
        dp[i][0] = dp[i - 1][0] + matrix[i][0]

    for j in range(1, cols):
        dp[0][j] = dp[0][j - 1] + matrix[0][j]

    # Fill in the list in a bottom-up manner
    for i in range(1, rows):
        for j in range(1, cols):
            # Update the list based on the recurrence relation
            dp[i][j] = matrix[i][j] + min(dp[i - 1][j], dp[i][j - 1])

    # The final result is stored in the bottom-right corner of the
matrix
    return dp[rows - 1][cols - 1]

# Example usage:
input_matrix = [
    [1, 3, 1],
    [1, 5, 1],
    [4, 2, 1]
]

result = dp_matrix_framework(input_matrix)

print(f"The minimum path sum is: {result}")
```

In this framework, we use a bottom-up approach to fill in the dp matrix. The base cases are initialized, and then we iteratively update each entry based on the recurrence relation. The final result is stored in the bottom-right corner of the matrix, representing the solution to the specific 2D dynamic programming problem. You can adapt this framework for different 2D matrix problems by adjusting the base cases and recurrence relation accordingly.

Advanced

a. Difference array

The difference array, also known as the prefix sum array, is a useful concept in computer science. It is often used to efficiently perform range updates in an array.

```
class DifferenceArray:
    def __init__(self, nums):
        self.nums = nums
        self.diff_array = [0] * len(nums)

        # Initialize the difference array
        self.diff_array[0] = nums[0]
        for i in range(1, len(nums)):
            self.diff_array[i] = nums[i] - nums[i - 1]

    def update_range(self, start, end, value):
        # Update the difference array
        self.diff_array[start] += value
        if end + 1 < len(self.nums):
            self.diff_array[end + 1] -= value

    def apply_updates(self):
        # Update the original array using the difference array
        for i in range(len(self.nums)):
            if i == 0:
                self.nums[i] = self.diff_array[i]
            else:
                self.nums[i] = self.nums[i - 1] + self.diff_array[i]

# Example usage:
original_array = [1, 2, 3, 4, 5]
diff_array_instance = DifferenceArray(original_array)

# Update the range [1, 3] by adding 2
diff_array_instance.update_range(1, 3, 2)

# Apply the updates to the original array
diff_array_instance.apply_updates()

print("Updated array:", original_array)
```

In this example, we use a DifferenceArray class to encapsulate the concept. The update_range method updates the difference array for a given range, and the

apply_updates method applies the updates to the original array. This allows us to efficiently perform range updates in constant time.

b. Tries

Tries, also known as prefix trees, are tree-like data structures that are used to store an associative array where the keys are sequences, usually strings. Each node in the trie represents a common prefix of a set of keys.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                current.children[char] = TrieNode()
            current = current.children[char]
        current.is_end_of_word = True

    def search(self, word):
        current = self.root
        for char in word:
            if char not in current.children:
                return False
            current = current.children[char]
        return current.is_end_of_word

    def starts_with_prefix(self, prefix):
        current = self.root
        for char in prefix:
            if char not in current.children:
                return False
            current = current.children[char]
        return True

# Example usage:
trie = Trie()
words = ["apple", "app", "apricot", "banana"]
```

```
# Insert words into the trie
for word in words:
    trie.insert(word)

# Search for words
print(trie.search("apple"))     # True
print(trie.search("apricot"))   # True
print(trie.search("ap"))        # False

# Check if a prefix exists
print(trie.starts_with_prefix("ban"))   # False
print(trie.starts_with_prefix("app"))   # True
```

In this example, we define a TrieNode class representing a node in the Trie and a Trie class that provides methods for inserting words, searching for words, and checking if a prefix exists.

c. Bit manipulation

i. Setting bit

```
def set_bit(num, bit_position):
    return num | (1 << bit_position)

# Example usage:
number = 5  # Binary representation: 0b0101
bit_position_to_set = 1
result = set_bit(number, bit_position_to_set)
print(result)  # Output: 7 (binary representation: 0b0111)
```

ii. Clearing bit

```
def clear_bit(num, bit_position):
    return num & ~(1 << bit_position)

# Example usage:
number = 7  # Binary representation: 0b0111
bit_position_to_clear = 1
result = clear_bit(number, bit_position_to_clear)
print(result)  # Output: 5 (binary representation: 0b0101)
```

iii. Toggling bit

```
def toggle_bit(num, bit_position):
    return num ^ (1 << bit_position)

# Example usage:
number = 5  # Binary representation: 0b0101
bit_position_to_toggle = 1
result = toggle_bit(number, bit_position_to_toggle)
print(result)  # Output: 7 (binary representation: 0b0111)
```

iv. Checking if bit set

```
def is_bit_set(num, bit_position):
    return (num & (1 << bit_position)) != 0

# Example usage:
number = 7  # Binary representation: 0b0111
bit_position_to_check = 1
result = is_bit_set(number, bit_position_to_check)
print(result)  # Output: True
```

d. Modular arithmetic

i. Addition

```
def add_modulo(a, b, modulus):
    return (a + b) % modulus

# Example usage:
result = add_modulo(5, 7, 10)
print(result)  # Output: 2 (because (5 + 7) % 10 = 12 % 10 = 2)
```

ii. Multiplication

```
def multiply_modulo(a, b, modulus):
    return (a * b) % modulus

# Example usage:
result = multiply_modulo(3, 4, 7)
print(result)  # Output: 5 (because (3 * 4) % 7 = 12 % 7 = 5)
```

iii. Exponential

```
def power_modulo(base, exponent, modulus):
    result = 1
    base = base % modulus

    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent //= 2
        base = (base * base) % modulus

    return result

# Example usage:
result = power_modulo(2, 5, 10)
print(result)  # Output: 2 (because (2^5) % 10 = 32 % 10 = 2)
```

e. Dijkstra's

Dijkstra's algorithm is a popular algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. The algorithm maintains a set of vertices whose shortest distance from the source is known. It repeatedly selects the vertex with the smallest tentative distance, updates the distances of its neighbors, and adds the vertex to the set of visited vertices.

```
import heapq

def dijkstra(graph, start):
    # Initialize distances and priority queue
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # Skip if outdated entry
        if current_distance > distances[current_vertex]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
```

```
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage:
graph = {
    'A': {'B': 1, 'D': 3},
    'B': {'A': 1, 'D': 4, 'E': 2},
    'C': {'E': 5},
    'D': {'A': 3, 'B': 4, 'E': 1},
    'E': {'B': 2, 'C': 5, 'D': 1}
}

start_vertex = 'A'
shortest_distances = dijkstra(graph, start_vertex)

print(f"Shortest distances from vertex {start_vertex}:
{shortest_distances}")
```

In this example, the dijkstra function takes a weighted graph and a starting vertex as input and returns a dictionary containing the shortest distances from the starting vertex to all other vertices. The priority queue is implemented using Python's heapq module. The algorithm is efficient and guarantees the shortest paths when all edge weights are non-negative.

More time complexity

This summary provides guidelines on expected time complexities based on input size constraints in problem-solving, particularly in the context of coding interviews and competitive programming:

n <= 10:

- Expected time complexity: O(n^2 · n!) or O(4^n) (factorial or exponential).
- Consider backtracking or brute-force recursive algorithms.
- Given the small input size, correctness is often sufficient for efficiency.

10 < n <= 20:

- Expected time complexity: O(2^n).
- Consider algorithms that involve considering all subsets/subsequences.
- Backtracking and recursion are common approaches.

20 < n <= 100:

- Expected time complexity: O(n^3).
- Problems marked as "easy" on platforms like LeetCode may have this bound.

- Consider brute-force solutions with nested loops and look for optimizations.

100 < n <= 1,000:

- Expected time complexity: O(n^2) or O(n · logn).
- Sorting the input or using a heap may be helpful.
- Aim for an O(n) algorithm if possible.
- Techniques like hash maps, sliding window, monotonic stack, binary search, or heaps may be necessary.

1,000 < n < 100,000:

- Common constraint on platforms like LeetCode.
- Expected time complexity: O(n · logn) or O(n).
- Consider sorting, heap usage, or other efficient techniques.
- Aim for O(n) if possible; otherwise, use advanced techniques.

100,000 < n < 1,000,000:

- Rare constraint.
- Expected time complexity: O(n · logn) or O(n).
- Incorporating a hash map is often necessary.

1,000,000 < n:

- Huge inputs, typically in the range of 10^9 or more.
- Expected time complexity: O(logn) or O(1) (logarithmic or constant).
- Binary search or clever tricks with hash maps are common.
- O(n) is rare and seen in very advanced problems.
-

Understanding these guidelines helps in choosing appropriate algorithms and optimizing solutions based on the expected input size. It's crucial for effective problem-solving in coding interviews and competitive programming.