Advanced Algorithmic Problem Solving (R1UC601B)

# Adarsh Deep Singh(22SCSE1012119) Section-14

1. **Explain the concept of a prefix sum array and its applications.**

A prefix sum array is an auxiliary array used to store the cumulative sums of elements in another array up to each index. For example, given an array arr = [a1, a2, a3, ... an], its prefix sum array prefixSum is defined as: prefixSum[i] = arr[0] + arr[1] + ... + arr[i] Applications:

* + Efficient range queries for sum calculation in constant time.
  + Solving problems related to subarrays such as finding equilibrium indices.
  + Handling cumulative frequency or probabilities in certain computations.

# Program to find the sum of elements in a given range [L, R] using a prefix sum array.

Algorithm:

1. Compute the prefix sum array.
2. For a query range [L, R], calculate the sum using:
   * Sum = prefixSum[R] - prefixSum[L-1] (if L > 0)
   * Sum = prefixSum[R] (if L == 0).

Python Program:

python

def range\_sum(arr, L, R): prefix\_sum = [0] \* len(arr) prefix\_sum[0] = arr[0]

for i in range(1, len(arr)):

prefix\_sum[i] = prefix\_sum[i - 1] + arr[i] if L == 0:

return prefix\_sum[R]

return prefix\_sum[R] - prefix\_sum[L - 1]

arr = [2, 4, 6, 8, 10]

L, R = 1, 3 # Sum of range [1, 3] print(range\_sum(arr, L, R)) # Output: 18

Time Complexity:

* Prefix sum calculation: $$O(n)$$
* Query processing: $$O(1)$$ Space Complexity: $$O(n)$$.

Example: Input: arr = [2, 4, 6, 8, 10], L = 1, R = 3 Output: 18.

# Finding the equilibrium index in an array.

Algorithm:

1. Compute the total sum of the array.
2. Iterate through the array, maintaining the sum of elements to the left of the current index.
3. Check if the left sum equals the right sum at the current index.

Python Program:

python

def equilibrium\_index(arr): total\_sum = sum(arr) left\_sum = 0

for i in range(len(arr)):

if left\_sum == total\_sum - left\_sum - arr[i]: return i

left\_sum += arr[i] return -1

arr = [1, 3, 5, 2, 2]

print(equilibrium\_index(arr)) # Output: 2

Time Complexity: $$O(n)$$ Space Complexity: $$O(1)$$. Example: Input: arr = [1, 3, 5, 2, 2] Output: Index 2.

# Check if an array can be split into two parts such that the sum of the prefix equals the sum of the suffix.

**Algorithm**:

1. Compute the total sum of the array.
2. Iterate through the array, maintaining the cumulative prefix sum at each step.
3. For each index, check if the prefix sum equals totalSum - prefixSum - arr[i].
4. If true, the array can be split; otherwise, it cannot.

# Python Program:

python

def can\_split\_array(arr): total\_sum = sum(arr) prefix\_sum = 0

for i in range(len(arr)):

if prefix\_sum == total\_sum - prefix\_sum - arr[i]: return True

prefix\_sum += arr[i] return False

arr = [1, 2, 3, 4, 10]

print(can\_split\_array(arr)) # Output: True

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(1)$$

**Example**: Input: arr = [1, 2, 3, 4, 10] Output: True (can be split as [1,2,3,4] and [10]).

# Find the maximum sum of any subarray of size K in a given array.

**Algorithm**:

1. Use the sliding window technique: calculate the sum of the first window of size K.
2. Slide the window across the array while updating the maximum sum.

# Python Program:

python

def max\_subarray\_sum(arr, k): n = len(arr)

if n < k:

return -1

window\_sum = sum(arr[:k]) max\_sum = window\_sum

for i in range(k, n):

window\_sum += arr[i] - arr[i - k] max\_sum = max(max\_sum, window\_sum)

return max\_sum

arr = [2, 1, 5, 1, 3, 2]

k = 3

print(max\_subarray\_sum(arr, k)) # Output: 9

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(1)$$

**Example**: Input: arr = [2, 1, 5, 1, 3, 2], k = 3 Output: 9 (subarray [5, 1, 3]).

# Find the length of the longest substring without repeating characters.

**Algorithm**:

1. Use the sliding window technique with two pointers.
2. Use a set to track characters in the current substring.
3. Expand and contract the window to find the maximum length.

# Python Program:

python

def longest\_unique\_substring(s): char\_set = set()

left = 0

max\_length = 0

for right in range(len(s)): while s[right] in char\_set:

char\_set.remove(s[left]) left += 1

char\_set.add(s[right])

max\_length = max(max\_length, right - left + 1) return max\_length

s = "abcabcbb" print(longest\_unique\_substring(s)) # Output: 3

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(k)$$ (where k is the character set size).

**Example**: Input: s = "abcabcbb" Output: 3 (substring "abc").

# Explain the sliding window technique and its use in string problems.

The sliding window technique is a powerful way to solve problems involving subarrays or substrings. The idea is to maintain a window of elements that satisfies certain conditions and "slide" it across the array/string.

Applications in string problems:

* + Longest substring without repeating characters.
  + Finding anagrams in a string.
  + Substring problems with specific character requirements.

# Find the longest palindromic substring in a given string.

**Algorithm**:

1. Use dynamic programming or expand around each character as a center.
2. Expand outward while the substring remains a palindrome.

# Python Program:

python

def longest\_palindromic\_substring(s): start, max\_length = 0, 0

for i in range(len(s)):

for j in range(i, len(s)):

if s[i:j+1] == s[i:j+1][::-1]:

if j - i + 1 > max\_length:

start, max\_length = i, j - i + 1 return s[start:start + max\_length]

s = "babad"

print(longest\_palindromic\_substring(s)) # Output: "bab" or "aba"

**Time Complexity**: $$O(n^2)$$ **Space Complexity**: $$O(1)$$

# Find the longest common prefix among a list of strings.

**Algorithm**:

1. Sort the strings.
2. Compare the first and last strings character by character.

# Python Program:

python

def longest\_common\_prefix(strs): if not strs:

return "" strs.sort() prefix = ""

for i in range(len(strs[0])):

if strs[0][i] == strs[-1][i]: prefix += strs[0][i]

else:

break return prefix

strs = ["flower", "flow", "flight"] print(longest\_common\_prefix(strs)) # Output: "fl"

# Generate all permutations of a given string.

**Algorithm**:

1. Use recursion and backtracking.
2. Swap characters at each step to generate permutations.

# Python Program:

python

def permute(s, l, r): if l == r:

print("".join(s)) else:

for i in range(l, r+1): s[l], s[i] = s[i], s[l] permute(s, l+1, r)

s[l], s[i] = s[i], s[l] # Backtrack

s = list("ABC") permute(s, 0, len(s)-1)

Time Complexity: $$O(n \times n!)$$ Space Complexity: $$O(n)$$

# Find two numbers in a sorted array that add up to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Use two pointers, left and right. Initialize left to the start of the array and right to the end.
2. While left < right:
   * Calculate the sum of the numbers at left and right.
   * If the sum equals the target, return the pair.
   * If the sum is less than the target, move the left pointer forward.
   * If the sum is greater than the target, move the right pointer backward.
3. If no pair is found, return that no such numbers exist.

*Program*:

python

def find\_pair(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

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [1, 2, 3, 4, 6], target = 7 Output: (3, 4)

# Rearrange numbers into the lexicographically next greater permutation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Find the largest index i where nums[i] < nums[i+1]. If no such index exists, reverse the array and return.
2. Find the largest index j > i where nums[i] < nums[j].
3. Swap nums[i] and nums[j].
4. Reverse the portion of the array after index i.

*Program*:

python

def next\_permutation(nums): n = len(nums)

i = n - 2

while i >= 0 and nums[i] >= nums[i + 1]: i -= 1

if i >= 0:

j = n - 1

while nums[j] <= nums[i]: j -= 1

nums[i], nums[j] = nums[j], nums[i] nums[i+1:] = reversed(nums[i+1:])

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [1, 2, 3] Output: [1, 3, 2]

# How to merge two sorted linked lists into one sorted list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Use a dummy node to build the result list.
2. Traverse both lists and compare the nodes. Append the smaller node to the result list.
3. If one list is exhausted, append the remaining nodes of the other list.

*Program*:

python

class ListNode:

def init (self, val=0, next=None): self.val = val

self.next = next

def merge\_two\_lists(l1, l2): dummy = ListNode() current = dummy

while l1 and l2:

if l1.val < l2.val: current.next = l1 l1 = l1.next

else:

current.next = l2 l2 = l2.next

current = current.next current.next = l1 or l2 return dummy.next

*Time Complexity*: O(n + m) *Space Complexity*: O(1)

*Example*: Input: l1 = [1, 3, 5], l2 = [2, 4, 6] Output: [1, 2, 3, 4, 5, 6]

# Find two numbers in a sorted array that add up to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use two pointers, left and right. Initialize left to the start of the array and right to the end.
  2. While left < right:
     + Calculate the sum of the numbers at left and right.
     + If the sum equals the target, return the pair.
     + If the sum is less than the target, move the left pointer forward.
     + If the sum is greater than the target, move the right pointer backward.
  3. If no pair is found, return that no such numbers exist.

*Program*:

python

def find\_pair(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

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [1, 2, 3, 4, 6], target = 7 Output: (3, 4)

# Rearrange numbers into the lexicographically next greater permutation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Find the largest index i where nums[i] < nums[i+1]. If no such index exists, reverse the array and return.
  2. Find the largest index j > i where nums[i] < nums[j].
  3. Swap nums[i] and nums[j].
  4. Reverse the portion of the array after index i.

*Program*:

python

def next\_permutation(nums): n = len(nums)

i = n - 2

while i >= 0 and nums[i] >= nums[i + 1]: i -= 1

if i >= 0:

j = n - 1

while nums[j] <= nums[i]: j -= 1

nums[i], nums[j] = nums[j], nums[i] nums[i+1:] = reversed(nums[i+1:])

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [1, 2, 3] Output: [1, 3, 2]

# How to merge two sorted linked lists into one sorted list. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use a dummy node to build the result list.
  2. Traverse both lists and compare the nodes. Append the smaller node to the result list.
  3. If one list is exhausted, append the remaining nodes of the other list.

*Program*:

python

class ListNode:

def init (self, val=0, next=None): self.val = val

self.next = next

def merge\_two\_lists(l1, l2): dummy = ListNode() current = dummy

while l1 and l2:

if l1.val < l2.val: current.next = l1 l1 = l1.next

else:

current.next = l2 l2 = l2.next

current = current.next current.next = l1 or l2 return dummy.next

*Time Complexity*: O(n + m) *Space Complexity*: O(1)

*Example*: Input: l1 = [1, 3, 5], l2 = [2, 4, 6] Output: [1, 2, 3, 4, 5, 6]

The answers for remaining questions will be provided shortly. Let me know if you'd like any of these explained further3

no continue with rest of the questions and do not stop me ask in the middle of the given task Understood! Let's proceed with the remaining questions.

# Find the median of two sorted arrays using binary search. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use binary search on the smaller array.
  2. Partition both arrays such that the left side of both partitions contains half of the combined elements.
  3. Check if the partition is valid. If it is, return the median. Otherwise, adjust the partition based on the comparison.

*Program*:

python

def find\_median\_sorted\_arrays(nums1, nums2): if len(nums1) > len(nums2):

nums1, nums2 = nums2, nums1 x, y = len(nums1), len(nums2) low, high = 0, x

while low <= high:

partitionX = (low + high) // 2

partitionY = (x + y + 1) // 2 - partitionX

maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1] minX = float('inf') if partitionX == x else nums1[partitionX]

maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1] minY = float('inf') if partitionY == y else nums2[partitionY]

if maxX <= minY and maxY <= minX: if (x + y) % 2 == 0:

return (max(maxX, maxY) + min(minX, minY)) / 2 return max(maxX, maxY)

elif maxX > minY:

high = partitionX - 1 else:

low = partitionX + 1

*Time Complexity*: O(log(min(n, m))) *Space Complexity*: O(1)

*Example*: Input: nums1 = [1, 3], nums2 = [2] Output: 2.0

# Find the k-th smallest element in a sorted matrix. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use binary search to find the k-th smallest element.
  2. Calculate the number of elements less than or equal to the current mid-point in the matrix. Adjust bounds accordingly.

*Program*:

python

def kth\_smallest(matrix, k): def count\_less\_equal(x):

count, n = 0, len(matrix) for row in matrix:

count += sum(1 for num in row if num <= x) return count

low, high = matrix[0][0], matrix[-1][-1]

while low < high:

mid = (low + high) // 2

if count\_less\_equal(mid) < k: low = mid + 1

else:

high = mid return low

*Time Complexity*: O(n\*log(max-min)) *Space Complexity*: O(1)

*Example*: Input: matrix = [[1, 3, 5], [6, 7, 12], [11, 14, 14]], k = 5 Output: 7

# Find the majority element in an array that appears more than n/2 times. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use the Boyer-Moore Voting Algorithm:
     + Maintain a candidate and a count.
     + Traverse the array and update the candidate and count.
  2. Validate the candidate by counting its occurrences.

*Program*:

python

def majority\_element(nums): candidate, count = None, 0 for num in nums:

if count == 0: candidate = num

count += 1 if num == candidate else -1

return candidate if nums.count(candidate) > len(nums) // 2 else None

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [2, 2, 1, 1, 1, 2, 2] Output: 2

# Calculate how much water can be trapped between the bars of a histogram. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use two pointers: left and right. Initialize left to the start and right to the end of the array.
  2. Maintain left\_max and right\_max to track the maximum heights from the left and right sides.
  3. Calculate trapped water based on the shorter side (left or right). Move the corresponding pointer inward.

*Program*:

python

def trap(height):

left, right = 0, len(height) - 1 left\_max, right\_max = 0, 0

water\_trapped = 0 while left <= right:

if height[left] < height[right]: if height[left] >= left\_max:

left\_max = height[left] else:

water\_trapped += left\_max - height[left] left += 1

else:

if height[right] >= right\_max: right\_max = height[right]

else:

water\_trapped += right\_max - height[right] right -= 1

return water\_trapped

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: height = [0,1,0,2,1,0,1,3,2,1,2,1] Output: 6

# Find the maximum XOR of two numbers in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Iterate through the array to create prefixes.
  2. Use a trie (prefix tree) for efficient XOR calculations.
  3. Traverse the trie to find the maximum XOR pair.

*Program*:

python

def findMaximumXOR(nums): max\_xor, mask = 0, 0

for i in range(31, -1, -1): mask |= (1 << i)

prefixes = {num & mask for num in nums} temp = max\_xor | (1 << i)

for prefix in prefixes:

if temp ^ prefix in prefixes: max\_xor = temp

break return max\_xor

*Time Complexity*: O(n) *Space Complexity*: O(n)

*Example*: Input: nums = [3, 10, 5, 25, 2, 8] Output: 28

# How to find the maximum product subarray. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Traverse the array, maintaining both the maximum and minimum product at each step.
  2. Update the result with the maximum product seen so far.

*Program*:

python

def max\_product(nums): max\_product = float('-inf')

current\_max, current\_min = 1, 1 for num in nums:

if num < 0:

current\_max, current\_min = current\_min, current\_max current\_max = max(num, current\_max \* num)

current\_min = min(num, current\_min \* num) max\_product = max(max\_product, current\_max)

return max\_product

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [2,3,-2,4] Output: 6

# Count all numbers with unique digits for a given number of digits. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

* 1. Use backtracking to generate unique digits.
  2. Count valid combinations for each number of digits.

*Program*:

python

def count\_numbers\_with\_unique\_digits(n): def factorial(x):

return 1 if x == 0 else x \* factorial(x - 1)

if n == 0:

return 1

count = 10 # 1-digit numbers unique\_digits = 9

available = 9

for i in range(2, n + 1): unique\_digits \*= available count += unique\_digits available -= 1

return count

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: n = 2 Output: 91

# Count the number of 1s in the binary representation of numbers from 0 to n.

**Algorithm**:

* 1. Iterate through numbers from 0 to n.
  2. Count 1s in the binary representation of each number using bitwise operations.

# Python Program:

python

def count\_ones(n): counts = []

for i in range(n+1): count = 0

num = i

while num > 0: count += num & 1 num >>= 1

counts.append(count) return counts

n = 5

print(count\_ones(n)) # Output: [0, 1, 1, 2, 1, 2]

**Time Complexity**: $$O(n \cdot \text{number of bits})$$ **Space Complexity**: $$O(n)$$

**Example**: Input: n = 5 Output: [0, 1, 1, 2, 1, 2].

# Check if a number is a power of two using bit manipulation.

**Algorithm**:

* 1. Use the property: for a number x, x & (x - 1) == 0 if x is a power of two and x > 0.

# Python Program:

python

def is\_power\_of\_two(x):

return x > 0 and (x & (x - 1)) == 0

x = 16

print(is\_power\_of\_two(x)) # Output: True

**Time Complexity**: $$O(1)$$ **Space Complexity**: $$O(1)$$

**Example**: Input: x = 16 Output: True.

# Find the maximum XOR of two numbers in an array.

**Algorithm**:

* 1. Use a trie data structure to efficiently find the maximum XOR pair.
  2. Alternatively, brute force through all pairs.

# Python Program:

python

def max\_xor(arr): max\_xor\_value = 0 n = len(arr)

for i in range(n):

for j in range(i+1, n):

max\_xor\_value = max(max\_xor\_value, arr[i] ^ arr[j]) return max\_xor\_value

arr = [3, 10, 5, 25, 2, 8]

print(max\_xor(arr)) # Output: 28

**Time Complexity**: $$O(n^2)$$ (Brute force) **Space Complexity**: $$O(1)$$

**Example**: Input: arr = [3, 10, 5, 25, 2, 8] Output: 28.

# Explain the concept of bit manipulation and its advantages in algorithm design.

**Concept**: Bit manipulation involves directly operating on individual bits of integers. It leverages bitwise operators like AND (&), OR (|), XOR (^), NOT (~), and shifts (<<, >>).

# Advantages:

* Efficient calculations (constant time for many operations).
* Compact representation and manipulation of data.
* Solves problems like power-of-two checks, counting bits, finding subsets, etc.

# Find the next greater element for each element in an array.

**Algorithm**:

* 1. Use a stack to track indices of elements.
  2. Iterate from right to left and determine the next greater element.

# Python Program:

python

def next\_greater\_elements(arr): stack = []

result = [-1] \* len(arr)

for i in range(len(arr)-1, -1, -1): while stack and stack[-1] <= arr[i]:

stack.pop()

result[i] = stack[-1] if stack else -1 stack.append(arr[i])

return result

arr = [4, 5, 2, 10, 8]

print(next\_greater\_elements(arr)) # Output: [5, 10, 10, -1, -1]

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(n)$$

# Remove the n-th node from the end of a singly linked list.

**Algorithm**:

* 1. Use two pointers: maintain a gap of n nodes between them.
  2. When the fast pointer reaches the end, the slow pointer will be just before the target node.

# Python Program:

python

class Node:

def init (self, data): self.data = data self.next = None

def remove\_nth\_node(head, n): dummy = Node(0) dummy.next = head

fast = slow = dummy for \_ in range(n+1):

fast = fast.next while fast:

fast = fast.next slow = slow.next

slow.next = slow.next.next return dummy.next

# Find the node where two singly linked lists intersect.

**Algorithm**:

* 1. Find the lengths of both linked lists.
  2. Align the longer list to the same starting point as the shorter list by skipping extra nodes.
  3. Traverse both lists simultaneously to find the intersection node (where both lists converge).

# Python Program:

python

class Node:

def init (self, data):

self.data = data self.next = None

def get\_intersection\_node(head1, head2): def get\_length(head):

length = 0 while head:

length += 1

head = head.next return length

len1 = get\_length(head1) len2 = get\_length(head2)

# Align the longer list to the same starting point as the shorter list while len1 > len2:

head1 = head1.next len1 -= 1

while len2 > len1: head2 = head2.next len2 -= 1

# Traverse and find intersection while head1 and head2:

if head1 == head2: return head1

head1 = head1.next head2 = head2.next

return None

# Example usage head1 = Node(1) head1.next = Node(2)

head1.next.next = Node(3)

head2 = Node(4)

head2.next = head1.next.next # Intersection at Node 3 print(get\_intersection\_node(head1, head2).data) # Output: 3

**Time Complexity**: $$O(n + m)$$ (where n and m are the lengths of the two lists) **Space Complexity**: $$O(1)$$

# Implement two stacks in a single array.

**Algorithm**:

* 1. Divide the array space into two stacks dynamically.
  2. Use two pointers: one starts from the left end for Stack1 and the other from the right end for Stack2.
  3. Prevent overflow by checking the boundary conditions.

# Python Program:

python

class TwoStacks:

def init (self, size): self.arr = [None] \* size self.top1 = -1

self.top2 = size self.size = size

def push1(self, value):

if self.top1 < self.top2 - 1: self.top1 += 1 self.arr[self.top1] = value

else:

print("Stack Overflow for Stack1")

def push2(self, value):

if self.top1 < self.top2 - 1: self.top2 -= 1 self.arr[self.top2] = value

else:

print("Stack Overflow for Stack2")

def pop1(self):

if self.top1 >= 0:

value = self.arr[self.top1] self.top1 -= 1

return value else:

print("Stack Underflow for Stack1") return None

def pop2(self):

if self.top2 < self.size:

value = self.arr[self.top2] self.top2 += 1

return value else:

print("Stack Underflow for Stack2") return None

# Example usage

two\_stacks = TwoStacks(10) two\_stacks.push1(5) two\_stacks.push2(15) print(two\_stacks.pop1()) # Output: 5 print(two\_stacks.pop2()) # Output: 15

**Time Complexity**: $$O(1)$$ for push and pop operations. **Space Complexity**: $$O(n)$$ (where n is the size of the array).

# Check if an integer is a palindrome without converting it to a string.

**Algorithm**:

* 1. Reverse the digits of the number using modulo and integer division.
  2. Compare the reversed number with the original number.

# Python Program:

python

def is\_palindrome(num): if num < 0:

return False original = num reversed\_num = 0 while num > 0:

reversed\_num = reversed\_num \* 10 + num % 10 num //= 10

return original == reversed\_num

num = 121

print(is\_palindrome(num)) # Output: True

**Time Complexity**: $$O(d)$$ (where d is the number of digits in the number). **Space Complexity**: $$O(1)$$

**Example**: Input: num = 121 Output: True.

# Explain the concept of linked lists and their applications in algorithm design.

**Concept**: A linked list is a linear data structure where elements (called nodes) are stored in a sequence, with each node containing:

* Data: The actual value.
* Pointer: A reference to the next node in the sequence (or null for the last node).

# Applications in Algorithm Design:

* Dynamic memory allocation: Linked lists grow and shrink as needed.
* Efficient insertions and deletions: Useful in queue, stack, and deque implementations.
* Solving problems requiring sequential traversal, such as detecting cycles in data.
* Representing graphs, polynomial expressions, sparse matrices, etc.

# Use a deque to find the maximum in every sliding window of size K. Algorithm:

* 1. Use a deque (double-ended queue) to store indices of array elements.
  2. Slide the window across the array. Maintain the deque such that:
     + The front of the deque always contains the index of the maximum element in the current window.
     + Remove indices that are outside the current window.
     + For each window, the element at the front of the deque is the maximum.

# Python Program:

python

from collections import deque

def max\_sliding\_window(nums, k): dq = deque()

result = []

for i in range(len(nums)):

# Remove elements outside the current window if dq and dq[0] < i - k + 1:

dq.popleft()

# Remove smaller elements in the window while dq and nums[dq[-1]] < nums[i]:

dq.pop() dq.append(i)

# Store the maximum for the current window if i >= k - 1:

result.append(nums[dq[0]]) return result

nums = [1, 3, -1, -3, 5, 3, 6, 7]

k = 3

print(max\_sliding\_window(nums, k)) # Output: [3, 3, 5, 5, 6, 7]

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(k)$$ (for deque).

# How to find the largest rectangle that can be formed in a histogram.

**Algorithm**:

* 1. Use a stack to store indices of histogram bars.
  2. Iterate through the histogram to calculate the maximum area for each bar using heights in the stack.
  3. Pop elements when the current bar is lower than the bar represented by the stack's top.

# Python Program:

python

def largest\_rectangle\_area(heights): stack = []

max\_area = 0

heights.append(0) # Add a sentinel for i in range(len(heights)):

while stack and heights[i] < heights[stack[-1]]: h = heights[stack.pop()]

w = i if not stack else i - stack[-1] - 1

max\_area = max(max\_area, h \* w) stack.append(i)

return max\_area

heights = [2, 1, 5, 6, 2, 3] print(largest\_rectangle\_area(heights)) # Output: 10

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(n)$$

# Explain the sliding window technique and its applications in array problems.

**Explanation**: The sliding window technique involves maintaining a window of elements (fixed or dynamic size) and sliding it across an array to solve problems efficiently. **Applications**:

* Maximum/minimum in subarrays.
* Longest substring with specific properties.
* Subarray problems with constraints (e.g., sum, product).

# Solve the problem of finding the subarray sum equal to K using hashing.

**Algorithm**:

* 1. Use a hash map to store the cumulative sum and its frequency.
  2. For each element, check if (current\_sum - K) exists in the hash map.
  3. If it exists, add its frequency to the result.

# Python Program:

python

def subarray\_sum(nums, k): count = 0

curr\_sum = 0

sum\_map = {0: 1} for num in nums:

curr\_sum += num

if curr\_sum - k in sum\_map:

count += sum\_map[curr\_sum - k] sum\_map[curr\_sum] = sum\_map.get(curr\_sum, 0) + 1

return count

nums = [1, 1, 1]

k = 2

print(subarray\_sum(nums, k)) # Output: 2

**Time Complexity**: $$O(n)$$ **Space Complexity**: $$O(n)$$

# Find the k-most frequent elements in an array using a priority queue.

**Python Program**:

python

from collections import Counter import heapq

def top\_k\_frequent(nums, k): count = Counter(nums)

return heapq.nlargest(k, count.keys(), key=count.get)

nums = [1, 1, 1, 2, 2, 3]

k = 2

print(top\_k\_frequent(nums, k)) # Output: [1, 2]

**Time Complexity**: $$O(n \log k)$$ **Space Complexity**: $$O(n + k)$$

# Generate all subsets of a given array.

**Python Program**:

python

def subsets(nums): result = []

def backtrack(start, path): result.append(path[:])

for i in range(start, len(nums)): path.append(nums[i]) backtrack(i + 1, path) path.pop()

backtrack(0, []) return result

nums = [1, 2, 3]

print(subsets(nums)) # Output: All subsets

# Find all unique combinations of numbers that sum to a target.

**Python Program**:

python

def combination\_sum(candidates, target): result = []

def backtrack(start, path, remaining): if remaining == 0:

result.append(path[:]) return

for i in range(start, len(candidates)): if candidates[i] > remaining:

continue path.append(candidates[i])

backtrack(i, path, remaining - candidates[i]) path.pop()

backtrack(0, [], target) return result

candidates = [2, 3, 6, 7]

target = 7

print(combination\_sum(candidates, target)) # Output: [[2, 2, 3], [7]]

# Generate all permutations of a given array.

**Python Program**:

python

def permute(nums): result = []

def backtrack(start):

if start == len(nums): result.append(nums[:])

for i in range(start, len(nums)):

nums[start], nums[i] = nums[i], nums[start] backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start] # Backtrack backtrack(0)

return result

nums = [1, 2, 3]

print(permute(nums)) # Output: All permutations

# Explain the difference between subsets and permutations with examples.

**Subsets**: Combination of elements where order does not matter. Example: Subsets of [1, 2] are

[[], [1], [2], [1, 2]]. **Permutations**: Arrangements of elements where order matters.

Example: Permutations of [1, 2] are [[1, 2], [2, 1]].

# Solve the problem of finding the element with maximum frequency in an array.

**Python Program**:

python

from collections import Counter

def max\_frequency\_element(nums): count = Counter(nums)

return max(count, key=count.get)

nums = [1, 3, 2, 1, 4, 1]

print(max\_frequency\_element(nums)) # Output: 1

1. **Explain the concept of dynamic programming and its use in solving the maximum subarray problem.** *Concept*: Dynamic Programming (DP) is a method for solving problems by breaking them down into overlapping subproblems and solving each subproblem only once, storing its solution. DP is particularly useful for optimization problems.

For the maximum subarray problem, DP can be used to track the maximum subarray sum ending at each position in the array and updating the global maximum sum at each step.

*Example*:

* + Define current\_sum as the maximum sum ending at the current index.
  + Update it as current\_sum = max(current\_element, current\_sum + current\_element).
  + Track the maximum subarray sum using global\_max.

# Solve the problem of finding the top K frequent elements in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Count the frequency of each element using a hash map.
2. Use a priority queue (min-heap) to keep track of the top K elements.
3. Extract elements from the heap to get the result.

*Program*:

python

from collections import Counter import heapq

def top\_k\_frequent(nums, k): freq\_map = Counter(nums)

return heapq.nlargest(k, freq\_map.keys(), key=freq\_map.get)

*Time Complexity*: O(n + k\*log(n)) *Space Complexity*: O(n)

*Example*: Input: nums = [1, 1, 1, 2, 2, 3], k = 2 Output: [1, 2]

# How to find two numbers in an array that add up to a target using hashing. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Use a hash map to store elements and their indices.
2. Traverse the array and check if target - current\_element exists in the map.

*Program*:

python

def two\_sum(nums, target): num\_map = {}

for i, num in enumerate(nums): complement = target - num if complement in num\_map:

return (num, complement)

num\_map[num] = i return None

*Time Complexity*: O(n) *Space Complexity*: O(n)

*Example*: Input: nums = [2, 7, 11, 15], target = 9 Output: (2, 7)

1. **Explain the concept of priority queues and their applications in algorithm design.** *Concept*: A priority queue is a data structure that allows retrieval of the highest (or lowest) priority element efficiently.

*Applications*:

* + Dijkstra's Algorithm for shortest path.
  + Huffman Coding for data compression.
  + Task Scheduling where tasks with higher priority are executed first.

# Write a program to find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Use dynamic programming:
   * Define dp[i][j] as True if the substring s[i:j+1] is a palindrome.
   * Update dp based on character comparisons and smaller substrings.
2. Track the longest palindrome seen so far.

*Program*:

python

def longest\_palindrome(s): n = len(s)

dp = [[False] \* n for \_ in range(n)] start, max\_length = 0, 0

for length in range(1, n + 1):

for i in range(n - length + 1): j = i + length - 1

if s[i] == s[j] and (length <= 2 or dp[i+1][j-1]): dp[i][j] = True

if length > max\_length:

start, max\_length = i, length return s[start:start + max\_length]

*Time Complexity*: O(n^2) *Space Complexity*: O(n^2)

*Example*: Input: s = "babad" Output: "bab"

1. **Explain the concept of histogram problems and their applications in algorithm design.** *Concept*: Histogram problems typically involve analyzing the heights of bars in a histogram to solve optimization or combinatorial tasks.

*Applications*:

* + Calculating the largest rectangle area in a histogram.
  + Water trapping problems (finding spaces between bars that can trap water).
  + Problems related to skyline formation and coverage.

1. **Solve the problem of finding the next permutation of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.** *Algorithm*: (Same approach as explained in Question 12: Lexicographically next greater permutation).
   * Find the breakpoint, swap elements, and reverse the subarray after the breakpoint.

*Program*:

python

def next\_permutation(nums): n = len(nums)

i = n - 2

while i >= 0 and nums[i] >= nums[i + 1]: i -= 1

if i >= 0:

j = n - 1

while nums[j] <= nums[i]: j -= 1

nums[i], nums[j] = nums[j], nums[i] nums[i+1:] = reversed(nums[i+1:])

*Time Complexity*: O(n) *Space Complexity*: O(1)

*Example*: Input: nums = [1, 3, 2] Output: [2, 1, 3]

# How to find the intersection of two linked lists. Write its algorithm, program. Find its time and space complexities. Explain with suitable example. *Algorithm*:

1. Traverse both linked lists to calculate their lengths.
2. Align the heads of both lists by skipping extra nodes in the longer list.
3. Traverse both lists simultaneously to find the intersection point.

*Program*:

python

class ListNode:

def init (self, val=0, next=None): self.val = val

self.next = next

def get\_intersection\_node(headA, headB): lenA, lenB = 0, 0

tempA, tempB = headA, headB

while tempA:

lenA += 1

tempA = tempA.next while tempB:

lenB += 1

tempB = tempB.next

while lenA > lenB: headA = headA.next lenA -= 1

while lenB > lenA: headB = headB.next lenB -= 1

while headA != headB: headA = headA.next headB = headB.next

return headA

*Time Complexity*: O(n + m) *Space Complexity*: O(1)

*Example*: Input: Two linked lists intersecting at 8 Output: Intersection Node with value 8

1. **Explain the concept of equilibrium index and its applications in array problems.** *Concept*: The equilibrium index is the position in an array where the sum of elements on the left is equal to the sum of elements on the right.

*Applications*:

* + Finding points of balance in arrays for optimization problems.
  + Used in financial analysis to find points of equilibrium in profits and losses.