**Advanced Algorithmic Problem Solving (R1UC601B)**

NAME : Drishti Singh

ADMISSION NO: 22scse1011154

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

Ans. A **prefix sum array** is a derived array that stores the cumulative sum of elements from the original array. For a given array A of length n, the prefix sum array P is defined such that:

**P[i]=A[0]+A[1]+…+A[i],for 0≤i<nP[i] = A[0] + A[1] + \ldots + A[i], \quad \text{for } 0 \leq i < nP[i]=A[0]+A[1]+…+A[i],for 0≤i<n**

It enables efficient computation of the sum of any subarray in constant time after a linear time preprocessing.

**Example:**

Consider the array A = [3, 6, 2, 8, 4].

The corresponding prefix sum array P would be:

* P[0] = 3
* P[1] = 3 + 6 = 9
* P[2] = 9 + 2 = 11
* P[3] = 11 + 8 = 19
* P[4] = 19 + 4 = 23

Thus, P = [3, 9, 11, 19, 23].

To compute the sum of elements from index L to R (i.e., A[L] + A[L+1] + ... + A[R]), we can use:

Sum[L,R]=P[R]−P[L−1],if L>0\text{Sum}[L, R] = P[R] - P[L - 1], \quad \text{if } L > 0Sum[L,R]=P[R]−P[L−1],if L>0 Sum[L,R]=P[R],if L=0\text{Sum}[L, R] = P[R], \quad \text{if } L = 0Sum[L,R]=P[R],if L=0

**Applications:**

1. **Efficient Range Sum Queries**: Supports constant-time range sum computation after linear preprocessing.
2. **Finding Equilibrium Index**: Used to determine if there exists an index where the sum of elements to the left equals the sum to the right.
3. **Subarray Problems**: Frequently used in problems involving fixed or variable-length subarray computations.
4. **Matching Prefix and Suffix Sums**: Useful in problems that involve comparing different sections of an array.
5. **Multidimensional Arrays (2D Prefix Sums)**: Extended to matrices for quick computation of submatrix sums in image processing and grid-based problems.
6. **Binary Search Optimization**: Used in conjunction with binary search to find thresholds or bounds based on cumulative values.
7. Write a program to find the sum of elements in a given range [L, R] using a prefix sum array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans. **Algorithm:**

**Input**:

* An array A of size n
* Range indices L and R such that 0 ≤ L ≤ R < n

**Output**:

* Sum of elements from index L to R (inclusive)

**Steps**:

1. Initialize a prefix sum array P of size n.
2. Set P[0] = A[0].
3. For i = 1 to n - 1, compute:

P[i]=P[i−1]+A[i]P[i] = P[i - 1] + A[i]P[i]=P[i−1]+A[i]

1. If L == 0, return P[R].
2. Otherwise, return P[R] - P[L - 1].

**Python Program:**

def range\_sum\_prefix(arr, L, R):

n = len(arr)

prefix = [0] \* n

prefix[0] = arr[0]

for i in range(1, n):

prefix[i] = prefix[i - 1] + arr[i]

if L == 0:

return prefix[R]

else:

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

print("Sum of elements from index", L, "to", R, ":", range\_sum\_prefix(arr, L, R))

**Time and Space Complexities:**

* **Time Complexity**:
  + Prefix sum construction: **O(n)**
  + Query processing: **O(1)**
* **Space Complexity**:
  + Additional space for prefix sum array: **O(n)**

3. Solve the problem of finding the equilibrium index in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Ans**. Algorithm:**

1. Compute total sum of the array.
2. Initialize left\_sum = 0.
3. For each index i, check if:

left\_sum==total\_sum−left\_sum−A[i]

4.If yes, return i; else, update left\_sum += A[i].

**Code:**

def find\_equilibrium\_index(arr):

total = sum(arr)

left = 0

for i in range(len(arr)):

if left == total - left - arr[i]:

return i

left += arr[i]

return -1

* **Complexities:**

**Time**: O(n)

**Space**: O(1)

**4. 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. Calculate the total sum of the array.
2. Iterate through the array, maintaining a leftSum.
3. For each element, add it to leftSum.
4. Calculate rightSum as totalSum - leftSum.
5. If leftSum equals rightSum, the split is found. Return true.
6. If the loop finishes without finding a split, return false.

Program

def can\_split\_array(arr):

"""

Checks if an array can be split into two parts with equal sums.

Args:

arr: The input array of numbers.

Returns:

True if a split exists, False otherwise.

"""

total\_sum = sum(arr)

left\_sum = 0

for i in range(len(arr)):

left\_sum += arr[i]

right\_sum = total\_sum - left\_sum

if left\_sum == right\_sum:

return True

return False

# Example usage

arr1 = [1, 2, 3, 0, 3, 2, 1]

arr2 = [1, 2, 3, 4, 5]

print(f"Array 1: {arr1}, Can split: {can\_split\_array(arr1)}") # Output: True

print(f"Array 2: {arr2}, Can split: {can\_split\_array(arr2)}") # Output: False

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

**Algorithm**

1. Calculate the sum of the first K elements. This is the initial maxSum.
2. Use a sliding window:
   * Iterate from the Kth element to the end of the array.
   * Subtract the element that just left the window (the element at i - K).
   * Add the new element entering the window (the element at i).
   * Update currentSum.
   * Update maxSum if currentSum is greater.
3. Return maxSum.

**Program (Python):**

**def max\_sum\_subarray\_k(arr, k):**

**n = len(arr)**

**if k > n:**

**return None**

**current\_sum = sum(arr[:k])**

**max\_sum = current\_sum**

**for i in range(k, n):**

**current\_sum = current\_sum - arr[i - k] + arr[i]**

**max\_sum = max(max\_sum, current\_sum)**

**return max\_sum**

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

Algorithm:

1. Initialize a starting index start = 0, a maximum length max\_length = 0, and a dictionary char\_index\_map to store the index of each character.
2. Iterate through the string s with index i and character c.
3. If c is in char\_index\_map and its last seen index is greater than or equal to start, update start to char\_index\_map[c] + 1.
4. Update char\_index\_map[c] to i.
5. Calculate the current substring length as i - start + 1 and update max\_length if necessary.
6. Return max\_length.

Program:

def longest\_substring\_without\_repeating\_chars(s):

start = 0

max\_length = 0

char\_index\_map = {}

for i, c in enumerate(s):

if c in char\_index\_map and char\_index\_map[c] >= start:

start = char\_index\_map[c] + 1

char\_index\_map[c] = i

max\_length = max(max\_length, i - start + 1)

return max\_length

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the string.
* Space Complexity: O(min(m, n)), where n is the length of the string and m is the size of the character set.

Example:

* Input: "abcabcbb"
* Explanation: The longest substring without repeating characters is "abc", with a length of 3.
* Output: 3

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

The sliding window technique is a method for efficiently processing contiguous segments of arrays or strings. A window of a certain size slides over the data, and operations are performed on the elements within the window. This technique reduces the time complexity of algorithms by avoiding redundant calculations.

Use in String Problems:

* Finding the longest substring with certain properties (e.g., without repeating characters).
* Counting occurrences of a pattern in a string.
* Finding substrings that satisfy specific conditions.

8. Find the longest palindromic substring in a given string.

Algorithm:

1. Initialize a variable longest\_palindrome to store the longest palindromic substring found so far.
2. Iterate through the string s with index i.
3. For each i, check for both odd and even length palindromes centered at i.
   * For odd length palindromes, expand around i (left = i, right = i).
   * For even length palindromes, expand around i and i+1 (left = i, right = i + 1).
4. While the characters at left and right are equal and within the string bounds, expand the window (decrement left, increment right).
5. Update longest\_palindrome if the current palindrome is longer.
6. Return longest\_palindrome.

Program:

def longest\_palindromic\_substring(s):

longest\_palindrome = ""

def expand\_around\_center(left, right):

while left >= 0 and right < len(s) and s[left] == s[right]:

left -= 1

right += 1

return s[left + 1:right]

for i in range(len(s)):

# Odd length palindromes

palindrome1 = expand\_around\_center(i, i)

if len(palindrome1) > len(longest\_palindrome):

longest\_palindrome = palindrome1

# Even length palindromes

palindrome2 = expand\_around\_center(i, i + 1)

if len(palindrome2) > len(longest\_palindrome):

longest\_palindrome = palindrome2

return longest\_palindrome

Time and Space Complexities:

* Time Complexity: O(n^2), where n is the length of the string.
* Space Complexity: O(1).

Example:

* Input: "babad"
* Explanation: The longest palindromic substring is "bab".
* Output: "bab"

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

Algorithm:

1. If the list of strings is empty, return an empty string.
2. Initialize the prefix to the first string in the list.
3. Iterate through the remaining strings in the list.
4. For each string, compare its characters with the characters in the prefix.
5. If the characters match, continue comparing.
6. If the characters don't match or the end of either string is reached, shorten the prefix to the matching part.
7. If the prefix becomes empty, return an empty string.
8. Return the final prefix.

Program:

def longest\_common\_prefix(strs):

if not strs:

return ""

prefix = strs[0]

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

j = 0

while j < len(prefix) and j < len(strs[i]) and prefix[j] == strs[i][j]:

j += 1

prefix = prefix[:j]

if not prefix:

return ""

return prefix

Time and Space Complexities:

* Time Complexity: O(S), where S is the total number of characters in all strings. In the worst case, we compare each character of each string.
* Space Complexity: O(1). The space used is constant.

Example:

* Input: ["flower", "flow", "flight"]
* Explanation: The longest common prefix is "fl".
* Output: "fl"

10. Generate all permutations of a given string.

Algorithm:

1. If the string is empty, return a list containing an empty string.
2. If the string has only one character, return a list containing the string itself.
3. For each character c in the string:
   * Recursively generate all permutations of the remaining characters (excluding c).
   * For each permutation p of the remaining characters, insert c at every possible position in p.
4. Return the list of all generated permutations.

Program:

def generate\_permutations(s):

if len(s) == 0:

return [""]

if len(s) == 1:

return [s]

permutations = []

for i, c in enumerate(s):

remaining\_chars = s[:i] + s[i+1:]

for p in generate\_permutations(remaining\_chars):

for j in range(len(p) + 1):

permutations.append(p[:j] + c + p[j:])

return permutations

Time and Space Complexities:

* Time Complexity: O(n!), where n is the length of the string.
* Space Complexity: O(n!), to store the output permutations.

Example:

* Input: "abc"
* Explanation: The permutations of "abc" are ["abc", "acb", "bac", "bca", "cab", "cba"].
* Output: ["abc", "acb", "bac", "bca", "cab", "cba"]

11. Find two numbers in a sorted array that add up to a target.

Algorithm:

1. Initialize two pointers, left at the beginning of the array and right at the end.
2. While left < right:
   * Calculate the sum of the numbers at left and right.
   * If the sum equals the target, return the indices left and right.
   * If the sum is less than the target, increment left.
   * If the sum is greater than the target, decrement right.
3. If no such pair is found, return an empty list or appropriate value.

Program:

def find\_two\_sum(arr, target):

left, right = 0, len(arr) - 1

while left < right:

current\_sum = arr[left] + arr[right]

if current\_sum == target:

return [left, right]

elif current\_sum < target:

left += 1

else:

right -= 1

return []

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(1).

Example:

* Input: arr = [2, 7, 11, 15], target = 9
* Explanation: The numbers at indices 0 and 1 (2 and 7) add up to 9.
* Output: [0, 1]

12. Rearrange numbers into the lexicographically next greater permutation.

Algorithm:

1. Find the largest index i such that nums[i] < nums[i+1]. If no such i exists, the permutation is the last one (descending order).
2. Find the largest index j > i such that nums[j] > nums[i].
3. Swap nums[i] and nums[j].
4. Reverse the subarray from i+1 to the end of the array.

Program:

def next\_permutation(nums):

# Find the largest index i such that nums[i] < nums[i+1]

i = len(nums) - 2

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

i -= 1

if i >= 0:

# Find the largest index j > i such that nums[j] > nums[i]

j = len(nums) - 1

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

j -= 1

# Swap nums[i] and nums[j]

nums[i], nums[j] = nums[j], nums[i]

# Reverse the subarray from i+1 to the end

left, right = i + 1, len(nums) - 1

while left < right:

nums[left], nums[right] = nums[right], nums[left]

left += 1

right -= 1

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(1).

Example:

* Input: nums = [1, 2, 3]
* Explanation: The next greater permutation is [1, 3, 2].
* Output: [1, 3, 2]

13. How to merge two sorted linked lists into one sorted list.

Algorithm:

1. Create a dummy node as the head of the merged list.
2. Initialize a tail pointer to the dummy node.
3. While both lists have nodes:
   * Compare the values of the current nodes in both lists.
   * Append the node with the smaller value to the tail of the merged list.
   * Move the pointer of the list from which the node was taken to the next node.
   * Move the tail pointer to the newly added node.
4. Append the remaining nodes of the non-empty list to the tail of the merged list.
5. Return the next node of the dummy node (the head of the merged list).

Program:

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def merge\_two\_sorted\_lists(list1, list2):

dummy = ListNode()

tail = dummy

while list1 and list2:

if list1.val < list2.val:

tail.next = list1

list1 = list1.next

else:

tail.next = list2

list2 = list2.next

tail = tail.next

if list1:

tail.next = list1

elif list2:

tail.next = list2

return dummy.next

Time and Space Complexities:

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

Example:

* Input: list1 = [1, 2, 4], list2 = [1, 3, 4]
* Explanation: The merged sorted list is [1, 1, 2, 3, 4, 4].
* Output: [1, 1, 2, 3, 4, 4]

14. Find the median of two sorted arrays using binary search.

Algorithm:

1. Ensure nums1 is the shorter array. If not, swap them.
2. Initialize low = 0 and high = len(nums1).
3. Perform binary search:
   * Calculate partitionX and partitionY.
   * Find maxLeftX, minRightX, maxLeftY, and minRightY.
   * If the partitions are correct:
     + If (m + n) is even, median is (maxLeftX + maxLeftY) / 2.
     + If (m + n) is odd, median is max(maxLeftX, maxLeftY).
   * If maxLeftX > minRightY, move left in nums1.
   * Else, move right in nums1.
4. Return the median.

Program:

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

if len(nums1) > len(nums2):

nums1, nums2 = nums2, nums1

m, n = len(nums1), len(nums2)

low, high = 0, m

while low <= high:

partitionX = (low + high) // 2

partitionY = (m + n + 1) // 2 - partitionX

maxLeftX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]

minRightX = float('inf') if partitionX == m else nums1[partitionX]

maxLeftY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]

minRightY = float('inf') if partitionY == n else nums2[partitionY]

if maxLeftX <= minRightY and maxLeftY <= minRightX:

if (m + n) % 2 == 0:

return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0

else:

return max(maxLeftX, maxLeftY)

elif maxLeftX > minRightY:

high = partitionX - 1

else:

low = partitionX + 1

Time and Space Complexities:

* Time Complexity: O(log(min(m, n))), where m and n are the lengths of the arrays.
* Space Complexity: O(1).

Example:

* Input: nums1 = [1, 3], nums2 = [2]
* Explanation: The median is 2.
* Output: 2.0

15. Find the k-th smallest element in a sorted matrix.

Algorithm:

1. Initialize low to the smallest element and high to the largest.
2. Perform binary search:
   * Calculate mid.
   * Count the number of elements less than or equal to mid.
   * If the count is less than k, update low = mid + 1.
   * Else, update high = mid.
3. Return low.

Program:

def kth\_smallest(matrix, k):

n = len(matrix)

low = matrix[0][0]

high = matrix[n - 1][n - 1]

while low < high:

mid = (low + high) // 2

count = 0

for i in range(n):

j = n - 1

while j >= 0 and matrix[i][j] > mid:

j -= 1

count += (j + 1)

if count < k:

low = mid + 1

else:

high = mid

return low

Time and Space Complexities:

* Time Complexity: O(n log(max - min)), where n is the dimension of the matrix, and max and min are the largest and smallest elements.
* Space Complexity: O(1).

Example:

* Input: matrix = [[1, 5, 9], [10, 11, 13], [12, 13, 15]], k = 8
* Explanation: The 8th smallest element is 13.
* Output: 13

16. Find the majority element in an array that appears more than n/2 times.

Algorithm:

1. Use the Moore Voting Algorithm.
2. Initialize a candidate and a count to 0.
3. Iterate through the array:
   * If count is 0, set candidate to the current element and count to 1.
   * If the current element is equal to candidate, increment count.
   * Otherwise, decrement count.
4. The candidate will be the majority element.

Program:

def majority\_element(nums):

candidate = 0

count = 0

for num in nums:

if count == 0:

candidate = num

count = 1

elif num == candidate:

count += 1

else:

count -= 1

return candidate

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(1).

Example:

* Input: [3, 2, 3]
* Explanation: The majority element is 3.
* Output: 3

17. Calculate how much water can be trapped between the bars of a histogram.

Algorithm:

1. Initialize two pointers, left at the beginning and right at the end of the histogram.
2. Initialize left\_max and right\_max to 0.
3. Initialize water to 0.
4. While left < right:
   * If height[left] < height[right]:
     + If height[left] > left\_max, update left\_max.
     + Else, add left\_max - height[left] to water.
     + Increment left.
   * Else:
     + If height[right] > right\_max, update right\_max.
     + Else, add right\_max - height[right] to water.
     + Decrement right.
5. Return water.

Program:

def trap(height):

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

left\_max, right\_max = 0, 0

water = 0

while left < right:

if height[left] < height[right]:

if height[left] > left\_max:

left\_max = height[left]

else:

water += left\_max - height[left]

left += 1

else:

if height[right] > right\_max:

right\_max = height[right]

else:

water += right\_max - height[right]

right -= 1

return water

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the histogram.
* Space Complexity: O(1).

Example:

* Input: [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
* Explanation: The trapped water is 6.
* Output: 6

18. Find the maximum XOR of two numbers in an array. Algorithm:

1. Initialize maxXor to 0.
2. Iterate through all pairs of numbers (num1, num2) in the array.
3. Calculate the XOR of num1 and num2.
4. Update maxXor if the current XOR is greater.
5. Return maxXor.

Program:

def find\_max\_xor(nums):

max\_xor = 0

for i in range(len(nums)):

for j in range(i + 1, len(nums)):

max\_xor = max(max\_xor, nums[i] ^ nums[j])

return max\_xor

Time and Space Complexities:

* Time Complexity: O(n^2), where n is the length of the array.
* Space complexity : O(1) Example:
* Input: [3, 10, 5, 25, 2, 8]
* Explanation: The maximum XOR value is between 5 and 25, which is 28.
* Output: 28

19. How to find the maximum product subarray.

Algorithm:

1. Initialize max\_product and min\_product to the first element of the array.
2. Initialize result to max\_product.
3. Iterate through the array starting from the second element:
   * If the current element is negative, swap max\_product and min\_product.
   * Update max\_product to the maximum of the current element and the product of the current element and max\_product.
   * Update min\_product to the minimum of the current element and the product of the current element and min\_product.
   * Update result to the maximum of result and max\_product.
4. Return result.

Program:

def max\_product\_subarray(nums):

if not nums:

return 0

max\_product = nums[0]

min\_product = nums[0]

result = max\_product

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

if nums[i] < 0:

max\_product, min\_product = min\_product, max\_product

max\_product = max(nums[i], max\_product \* nums[i])

min\_product = min(nums[i], min\_product \* nums[i])

result = max(result, max\_product)

return result

Time and Space Complexities:

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(1).

Example:

* Input: [2, 3, -2, 4]
* Explanation: The maximum product subarray is [2, 3], with a product of 6.
* Output: 6

20. Count all numbers with unique digits for a given number of digits.

Algorithm:

1. If n is 0, return 1 (only one number with 0 digits: "").
2. If n is 1, return 10 (0, 1, 2, ..., 9).
3. Initialize count = 10 (for n = 1).
4. Initialize unique\_digits = 9 (choices for the first digit when n > 1).
5. For i from 2 to n:
   * unique\_digits \*= (10 - (i - 1))
   * count += unique\_digits
6. Return count.

Program:

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

if n == 0:

return 1

if n == 1:

return 10

count = 10

unique\_digits = 9

for i in range(2, min(n, 10) + 1):

unique\_digits \*= (10 - (i - 1))

count += unique\_digits

return count

Time and Space Complexities:

* Time Complexity: O(n), where n is the number of digits.
* Space Complexity: O(1).

Example:

* Input: 2
* Explanation: The numbers with unique digits are [0, 1, 2, ..., 9, 10, 12, ..., 98]. Total count is 91.
* Output: 91

21. How to count the number of 1s in the binary representation of numbers from 0 to n.

Algorithm:

1. Create an array dp of size n + 1.
2. dp[0] = 0.
3. For each number i from 1 to n:
   * dp[i] = dp[i // 2] + (i % 2).
4. Sum up all the values in dp.
5. Return the sum.

Program:

def count\_bits(n):

dp = [0] \*(n + 1)

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

dp[i] = dp[i // 2] + (i % 2)

return sum(dp)

Time and Space Complexities:

* Time Complexity: O(n), where n is the given number.
* Space Complexity: O(n).

Example:

* Input: n = 5
* Explanation:
  + 0 -> 0 (0 ones)
  + 1 -> 1 (1 one)
  + 2 -> 10 (1 one)
  + 3 -> 11 (2 ones)
  + 4 -> 100 (1 one)
  + 5 -> 101 (2 ones) Total 1s: 0 + 1 + 1 + 2 + 1 + 2 = 7
* Output: 7

22. How to check if a number is a power of two using bit manipulation.

Algorithm:

1. If the number n is less than or equal to 0, return false.
2. A number n is a power of two if and only if n & (n - 1) is 0.
3. Return the result of the check.

Program:

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

if n <= 0:

return False

return (n & (n - 1)) == 0

Time and Space Complexities:

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

Example:

* Input: 16
* Explanation: 16 in binary is 10000. 15 is 01111. 10000 & 01111 = 0.
* Output: True

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

**Concept of Bit Manipulation:**

Bit manipulation involves performing operations directly on the binary representation of numbers. These operations include AND, OR, XOR, NOT, left shift, and right shift.

**Advantages in Algorithm Design:**

* Efficiency: Bitwise operations are often faster than arithmetic operations.
* Space efficiency: Bits can be used to represent multiple states in a compact way.
* Conciseness: Bit manipulation can simplify complex logic.
* Problem-solving: Bit manipulation is useful for solving problems related to sets, subsets, and binary representations.

**25. Solve the problem of finding the next greater element for each element in an array.**

**Algorithm:**

1. Initialize an empty stack.
2. Initialize an output array result with -1 for each element.
3. Iterate through the array arr:
   * While the stack is not empty and the current element is greater than the element at the top of the stack:
     + Pop the index from the stack.
     + Set the result for the popped index to the current element.
   * Push the current index onto the stack.
4. Return the result array.

**Program:**

def next\_greater\_element(arr):

stack = []

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

for i, num in enumerate(arr):

while stack and num > arr[stack[-1]]:

index = stack.pop()

result[index] = num

stack.append(i)

return result

**Time and Space Complexities:**

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(n).

**Example:**

* Input: [1, 3, 2, 4]
* Explanation:
  + For 1, the next greater element is 3.
  + For 3, the next greater element is 4.
  + For 2, the next greater element is 4.
  + For 4, there is no next greater element.
* Output: [3, 4, 4, -1]

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

**Algorithm:**

1. Use two pointers, fast and slow.
2. Move fast n nodes ahead.
3. Move fast and slow together until fast reaches the end.
4. slow will be pointing to the node before the one to be deleted.
5. Change the next pointer of slow to skip the n-th node from the end.
6. Handle the edge case where the head is to be removed.

**Program:**

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def remove\_nth\_from\_end(head, n):

dummy = ListNode(0)

dummy.next = head

fast = dummy

slow = dummy

for \_ in range(n):

fast = fast.next

while fast.next:

fast = fast.next

slow = slow.next

slow.next = slow.next.next

return dummy.next

**Time and Space Complexities:**

* Time Complexity: O(L), where L is the length of the linked list.
* Space Complexity: O(1).

**Example:**

* Input: head = [1, 2, 3, 4, 5], n = 2
* Explanation: The 2nd node from the end is 4. After removal, the list is [1, 2, 3, 5].
* Output: [1, 2, 3, 5]

**27. Find the node where two singly linked lists intersect.**

**Algorithm:**

1. Calculate the lengths of both linked lists, lenA and lenB.
2. Find the difference diff = abs(lenA - lenB).
3. Move the pointer of the longer list diff nodes ahead.
4. Move both pointers simultaneously until they meet.
5. If the pointers meet, return the meeting node.
6. If either pointer reaches the end, the lists do not intersect, return None.

**Program:**

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

currA, currB = headA, headB

while currA:

lenA += 1

currA = currA.next

while currB:

lenB += 1

currB = currB.next

currA, currB = headA, headB

if lenA > lenB:

for \_ in range(lenA - lenB):

currA = currA.next

else:

for \_ in range(lenB - lenA):

currB = currB.next

while currA and currB and currA != currB:

currA = currA.next

currB = currB.next

return currA

**Time and Space Complexities:**

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

**Example:**

* Input:
  + listA = [4, 1, 8, 4, 5]
  + listB = [5, 6, 1, 8, 4, 5]
* Explanation: The intersection node is 8.
* Output: 8

**28. Implement two stacks in a single array.**

**Algorithm:**

1. Divide the array into two halves.
2. Use the left half for the first stack and the right half for the second stack.
3. Initialize two pointers, top1 and top2, to the start of their respective halves.
4. For the first stack, increment top1 when pushing and decrement when popping.
5. For the second stack, decrement top2 when pushing and increment when popping.
6. Handle stack overflow and underflow conditions.

**Program:**

class TwoStacks:

def \_\_init\_\_(self, n):

self.array = [None] \* n

self.size = n

self.top1 = -1

self.top2 = n

def push1(self, x):

if self.top1 < self.top2 - 1:

self.top1 += 1

self.array[self.top1] = x

else:

raise Exception("Stack Overflow")

def push2(self, x):

if self.top2 > self.top1 + 1:

self.top2 -= 1

self.array[self.top2] = x

else:

raise Exception("Stack Overflow")

def pop1(self):

if self.top1 >= 0:

x = self.array[self.top1]

self.top1 -= 1

return x

else:

raise Exception("Stack Underflow")

def pop2(self):

if self.top2 < self.size:

x = self.array[self.top2]

self.top2 += 1

return x

else:

raise Exception("Stack Underflow")

def peek1(self):

if self.top1 >= 0:

return self.array[self.top1]

else:

raise Exception("Stack Underflow")

def peek2(self):

if self.top2 < self.size:

return self.array[self.top2]

else:

raise Exception("Stack Underflow")

def is\_empty1(self):

return self.top1 == -1

def is\_empty2(self):

return self.top2 == self.size

**Time and Space Complexities:**

* Time Complexity: O(1) for all operations.
* Space Complexity: O(n), where n is the size of the array.

**Example:**

stacks = TwoStacks(6)

stacks.push1(1)

stacks.push1(2)

stacks.push2(4)

stacks.push2(3)

print(stacks.pop1()) # Output: 2

print(stacks.pop2()) # Output: 3

**29. Write a program to check if an integer is a palindrome without converting it to a string.**

**Algorithm:**

1. If the number is negative, it cannot be a palindrome, return false.
2. Initialize reversed\_num = 0 and original\_num = num.
3. While num > 0:
   * Get the last digit of num: last\_digit = num % 10.
   * Append the last digit to reversed\_num: reversed\_num = reversed\_num \* 10 + last\_digit.
   * Remove the last digit from num: num //= 10.
4. Compare original\_num and reversed\_num. If they are equal, the number is a palindrome, return true. Otherwise, return false.

**Program:**

def is\_palindrome(num):

if num < 0:

return False

reversed\_num = 0

original\_num = num

while num > 0:

last\_digit = num % 10

reversed\_num = reversed\_num \* 10 + last\_digit

num //= 10

return original\_num == reversed\_num

**Time and Space Complexities:**

* Time Complexity: O(log10(n)), where n is the number of digits in the integer.
* Space Complexity: O(1).

**Example:**

* Input: 121
* Explanation: The reversed number is also 121.
* Output: True

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

**Concept of Linked Lists:**

A linked list is a linear data structure where elements are stored in nodes. Each node contains a data field and a reference (or pointer) to the next node in the sequence.

**Applications in Algorithm Design:**

* Dynamic memory allocation: Linked lists can grow or shrink dynamically.
* Implementing other data structures: Stacks, queues, and hash tables can be implemented using linked lists.
* Representing sequences: Linked lists can efficiently represent ordered data.
* Graph representation: Adjacency lists use linked lists to store neighbors of a vertex.

**31. Use a deque to find the maximum in every sliding window of size K.**

**Algorithm:**

1. Initialize an empty deque dq.
2. Iterate through the array nums:
   * While the deque is not empty and the element at the front of the deque is out of the current window, remove it.
   * While the deque is not empty and the current element is greater than the element at the back of the deque, remove elements from the back.
   * Add the current element's index to the back of the deque.
   * If the current index is greater than or equal to k - 1, the maximum element in the current window is at the front of the deque. Add it to the result.
3. Return the result.

**Program:**

from collections import deque

def max\_sliding\_window(nums, k):

dq = deque()

result = []

for i, num in enumerate(nums):

while dq and dq[0] < i - k + 1:

dq.popleft()

while dq and nums[dq[-1]] < num:

dq.pop()

dq.append(i)

if i >= k - 1:

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

return result

**Time and Space Complexities:**

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(k).

**Example:**

* Input: nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3
* Explanation:
  + Window 1: [1, 3, -1], max = 3
  + Window 2: [3, -1, -3], max = 3
  + Window 3: [-1, -3, 5], max = 5
  + Window 4: [-3, 5, 3], max = 5
  + Window 5: [5, 3, 6], max = 6
  + Window 6: [3, 6, 7], max = 7
* Output: [3, 3, 5, 5, 6, 7]

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

**Algorithm:**

1. Initialize an empty stack.
2. Initialize max\_area = 0.
3. Iterate through the histogram heights:
   * While the stack is not empty and the current height is less than the height at the top of the stack:
     + Pop the index from the stack.
     + Calculate the area of the rectangle with the popped height.
     + Update max\_area.
   * Push the current index onto the stack.
4. While the stack is not empty:
   * Pop the index from the stack.
   * Calculate the area of the rectangle with the popped height.
   * Update max\_area.
5. Return max\_area.

**Program:**

def largest\_rectangle\_area(heights):

stack = []

max\_area = 0

heights.append(0) # Add a 0 at the end to clear the stack

for i, h in enumerate(heights):

while stack and heights[stack[-1]] > h:

top = stack.pop()

width = i - stack[-1] - 1 if stack else i

area = heights[top] \* width

max\_area = max(max\_area, area)

stack.append(i)

return max\_area

**Time and Space Complexities:**

* Time Complexity: O(n), where n is the number of bars in the histogram.
* Space Complexity: O(n).

**Example:**

* Input: [2, 1, 5, 6, 2, 3]
* Explanation: The largest rectangle has an area of 10 (5 x 2).
* Output: 10

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

**Algorithm:**

1. Initialize a dictionary prefix\_sums to store the cumulative sum up to each index and their frequency.
2. Initialize current\_sum = 0 and count = 0.
3. Put (0,1) in the prefix\_sums dictionary.
4. Iterate through the array nums:
   * Update current\_sum.
   * If current\_sum - k is in prefix\_sums, increment count by the frequency of current\_sum - k.
   * Update the frequency of current\_sum in prefix\_sums.
5. Return count.

**Program:**

def subarray\_sum\_equals\_k(nums, k):

prefix\_sums = {0: 1}

current\_sum = 0

count = 0

for num in nums:

current\_sum += num

if current\_sum - k in prefix\_sums:

count += prefix\_sums[current\_sum - k]

if current\_sum in prefix\_sums:

prefix\_sums[current\_sum] += 1

else:

prefix\_sums[current\_sum] = 1

return count

**Time and Space Complexities:**

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(n).

**Example:**

* Input: nums = [1, 1, 1], k = 2
* Explanation: The subarrays that sum to 2 are [1, 1] and [1, 1].
* Output: 2

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

**Algorithm:**

1. Count the frequency of each element using a dictionary.
2. Create a min-heap (priority queue) to store the k most frequent elements.
3. Iterate through the element frequencies:
   * Push the element and its frequency onto the heap.
   * If the size of the heap exceeds k, pop the element with the smallest frequency.
4. Extract the elements from the heap.
5. Return the k most frequent elements.

**Program:**

import heapq

from collections import Counter

def top\_k\_frequent(nums, k):

frequency\_map = Counter(nums)

min\_heap = []

for element, frequency in frequency\_map.items():

heapq.heappush(min\_heap, (frequency, element))

if len(min\_heap) > k:

heapq.heappop(min\_heap)

result = [element for frequency, element in min\_heap]

return result

**Time and Space Complexities:**

* Time Complexity: O(n log k), where n is the length of the array.
* Space Complexity: O(n).

**Example:**

* Input: nums = [1, 1, 1, 2, 2, 3], k = 2
* Explanation: The 2 most frequent elements are 1 and 2.
* Output: [1, 2]

**36. Generate all subsets of a given array.**

**Algorithm:**

1. Initialize an empty list subsets to store all subsets.
2. Iterate through all possible bitmasks from 0 to 2^n - 1, where n is the length of the array:
   * For each bitmask, create a subset by including elements at the indices where the corresponding bit is set.
   * Add the subset to the subsets list.
3. Return the subsets list.

**Program:**

def get\_subsets(nums):

subsets = []

n = len(nums)

for i in range(2\*\*n):

subset = []

for j in range(n):

if (i >> j) & 1:

subset.append(nums[j])

subsets.append(subset)

return subsets

**Time and Space Complexities:**

* Time Complexity: O(n \* 2^n), where n is the length of the array.
* Space Complexity: O(2^n).

**Example:**

* Input: [1, 2, 3]
* Explanation: The subsets are [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3].
* Output: [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]

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

**Algorithm:**

1. Sort the candidates array.
2. Initialize an empty list combinations to store the result.
3. Define a recursive function find\_combinations:
   * Base case: If the target is 0, add the current combination to the result.
   * Base case: If the target is negative or we have exhausted all candidates, return.
   * For each candidate starting from the current index:
     + Include the candidate in the current combination.
     + Recursively call find\_combinations with the updated target and the same index (to allow duplicates).
     + Backtrack: Remove the candidate from the current combination.
4. Call the recursive function with the initial parameters.
5. Return the combinations list.

**Program:**

def combination\_sum(candidates, target):

candidates.sort()

combinations = []

def find\_combinations(combination, remaining\_target, start\_index):

if remaining\_target == 0:

combinations.append(list(combination))

return

if remaining\_target < 0:

return

for i in range(start\_index, len(candidates)):

combination.append(candidates[i])

find\_combinations(combination, remaining\_target - candidates[i], i)

combination.pop()

find\_combinations([], target, 0)

return combinations

**Time and Space Complexities:**

* Time Complexity: O(k \* 2^n), where k is the average length of the combinations.
* Space Complexity: O(k \* 2^n).

**Example:**

* Input: candidates = [2, 3, 6, 7], target = 7
* Explanation:
  + [2, 2, 3] -> 2+2+3 = 7
  + [7] -> 7
* Output: [[2, 2, 3], [7]]

**38. Generate all permutations of a given array.**

(See problem 10)

**39. Explain the difference between subsets and permutations with examples.**

**Subsets:**

* A subset is a selection of elements from a set, where the order of elements does not matter.
* Example: For the set {1, 2, 3}, the subsets are {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}.

**Permutations:**

* A permutation is an arrangement of elements from a set, where the order of elements matters.
* Example: For the set {1, 2, 3}, the permutations are (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).

**Key Differences:**

* Order: Order matters in permutations, but not in subsets.
* Number of elements: A subset can have any number of elements from 0 to the size of the original set, while a permutation must have the same number of elements as the original set.

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

**Algorithm:**

1. Use a dictionary to store the frequency of each element in the array.
2. Iterate through the dictionary to find the element with the maximum frequency.
3. Return the element with the maximum frequency.

**Program:**

def find\_max\_frequency\_element(arr):

frequency\_map = {}

for element in arr:

if element in frequency\_map:

frequency\_map[element] += 1

else:

frequency\_map[element] = 1

max\_frequency = 0

max\_frequency\_element = None

for element, frequency in frequency\_map.items():

if frequency > max\_frequency:

max\_frequency = frequency

max\_frequency\_element = element

return max\_frequency\_element

**Time and Space Complexities:**

* Time Complexity: O(n), where n is the length of the array.
* Space Complexity: O(n).

**Example:**

* Input: [1, 2, 2, 3, 2, 4]
* Explanation: The element 2 appears

**41. Maximum Subarray Sum using Kadane's Algorithm**

Kadane's algorithm is an efficient dynamic programming approach to find the maximum sum of a contiguous subarray within a one-dimensional array.

**Algorithm:**

1. Initialize two variables: max\_so\_far (to store the overall maximum sum found so far) to negative infinity and current\_max (to store the maximum sum ending at the current position) to 0.
2. Iterate through the array.
3. For each element, update current\_max by taking the maximum of the current element and the sum of current\_max and the current element. This step decides whether to start a new subarray from the current element or extend the previous subarray.
4. Update max\_so\_far by taking the maximum of max\_so\_far and current\_max. This step keeps track of the overall maximum sum encountered.
5. After iterating through the entire array, max\_so\_far will hold the maximum subarray sum.

**Program (Python):**

Python

def max\_subarray\_sum\_kadane(arr):

max\_so\_far = float('-inf')

current\_max = 0

for x in arr:

current\_max = max(x, current\_max + x)

max\_so\_far = max(max\_so\_far, current\_max)

return max\_so\_far

# Example

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

max\_sum = max\_subarray\_sum\_kadane(arr)

print(f"Maximum subarray sum: {max\_sum}")

**Time and Space Complexity:**

* **Time Complexity:** O(n), where n is the size of the array, as we iterate through the array once.
* **Space Complexity:** O(1), as we use a constant amount of extra space for variables.

**42. Dynamic Programming and Maximum Subarray Problem**

**Concept of Dynamic Programming:**

Dynamic programming is an algorithmic technique for solving complex problems by breaking them down into smaller overlapping subproblems. It solves each subproblem only once and stores the results in a table (often called a "dp table") to avoid redundant computations. This approach is particularly useful for problems exhibiting two key properties:

1. **Optimal Substructure:** The optimal solution to the overall problem can be constructed from the optimal solutions to its subproblems.
2. **Overlapping Subproblems:** The same subproblems are encountered multiple times in the recursive solution.

**Use in Maximum Subarray Problem:**

Kadane's algorithm, as seen above, is a form of dynamic programming. Let's define dp[i] as the maximum sum of a contiguous subarray ending at index i.

The recurrence relation can be defined as:

dp[i] = max(arr[i], dp[i-1] + arr[i])

This means that the maximum sum ending at index i is either the element itself (starting a new subarray) or the maximum sum ending at the previous index plus the current element (extending the previous subarray).

The overall maximum subarray sum is then the maximum value in the dp array.

**Program (Python - Dynamic Programming Approach):**

Python

def max\_subarray\_sum\_dp(arr):

n = len(arr)

if n == 0:

return 0

dp = [0] \* n

dp[0] = arr[0]

max\_so\_far = dp[0]

for i in range(1, n):

dp[i] = max(arr[i], dp[i-1] + arr[i])

max\_so\_far = max(max\_so\_far, dp[i])

return max\_so\_far

# Example

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

max\_sum = max\_subarray\_sum\_dp(arr)

print(f"Maximum subarray sum (DP): {max\_sum}")

**Time and Space Complexity (DP Approach):**

* **Time Complexity:** O(n), as we iterate through the array once to fill the dp table.
* **Space Complexity:** O(n), due to the dp array of size n.

Kadane's algorithm is a space-optimized version of this dynamic programming approach, reducing the space complexity to O(1) by only keeping track of the current maximum sum.

**43. 43. 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 Frequencies:** Create a hash map (or dictionary) to store the frequency of each element in the array. Iterate through the array and update the counts.
2. **Use a Min-Heap (Priority Queue):** Create a min-heap of size K. Iterate through the frequency map.
   * If the heap size is less than K, add the (frequency, element) pair to the heap.
   * If the heap size is equal to K, compare the current element's frequency with the minimum frequency in the heap (the root of the min-heap).
     + If the current frequency is greater than the minimum frequency, remove the root of the heap and add the current (frequency, element) pair.
3. **Extract Top K:** After iterating through all the elements, the min-heap will contain the K elements with the highest frequencies. Extract the elements from the heap (the order might be from lowest to highest frequency within the top K).

**Program (Python):**

Python

import heapq

from collections import Counter

def top\_k\_frequent(nums, k):

frequency\_map = Counter(nums)

min\_heap = []

for num, freq in frequency\_map.items():

if len(min\_heap) < k:

heapq.heappush(min\_heap, (freq, num))

else:

if freq > min\_heap[0][0]:

heapq.heappop(min\_heap)

heapq.heappush(min\_heap, (freq, num))

top\_k = [item[1] for item in min\_heap]

return top\_k

# Example

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

k = 2

top\_k = top\_k\_frequent(nums, k)

print(f"Top {k} frequent elements: {top\_k}")

**Time and Space Complexities:**

* **Time Complexity:**
  + Counting frequencies: O(n), where n is the size of the array.
  + Building and maintaining the min-heap: O(m log k), where m is the number of unique elements in the array. In the worst case, m can be equal to n, so it becomes O(n log k).
  + Overall: **O(n log k)** in the worst case.
* **Space Complexity:**
  + Frequency map: O(m), where m is the number of unique elements.
  + Min-heap: O(k).
  + Overall: **O(n)** in the worst case (if all elements are unique).

**Example Explanation:**

For nums = [1, 1, 1, 2, 2, 3] and k = 2:

1. **Frequency Map:** {1: 3, 2: 2, 3: 1}
2. **Min-Heap:**
   * Initially empty.
   * Process (1, 3): Heap becomes [(3, 1)]
   * Process (2, 2): Heap becomes [(2, 2), (3, 1)] (min at the root)
   * Process (3, 1): 1 is not greater than 2, so the heap remains [(2, 2), (3, 1)]
3. **Extract Top K:** The heap contains (2, 2) and (3, 1). The elements are 2 and 1. Thus, the top 2 frequent elements are [1, 2] (the order in the output might vary).

**44. 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. Create an empty hash set (or dictionary) to store the numbers encountered so far.
2. Iterate through the array.
3. For each number num in the array, calculate the complement needed to reach the target (i.e., complement = target - num).
4. Check if the complement exists in the hash set.
   * If it does, it means we have found two numbers that add up to the target. Return the indices (or the numbers themselves).
   * If it doesn't, add the current num to the hash set.
5. If the loop completes without finding a pair, return an indication that no such pair exists.

**Program (Python):**

Python

def find\_two\_sum\_hashing(nums, target):

num\_map = {}

for index, num in enumerate(nums):

complement = target - num

if complement in num\_map:

return [num\_map[complement], index]

num\_map[num] = index

return None

# Example

nums = [2, 7, 11, 15]

target = 9

result = find\_two\_sum\_hashing(nums, target)

if result:

print(f"Two numbers with sum {target} found at indices: {result}")

else:

print(f"No two numbers with sum {target} found.")

**Time and Space Complexities:**

* **Time Complexity:** O(n), where n is the size of the array, as we iterate through the array once. Hash set operations (insertion and lookup) take O(1) on average.
* **Space Complexity:** O(n), as we may need to store all the numbers in the hash set in the worst case.

**Example Explanation:**

For nums = [2, 7, 11, 15] and target = 9:

1. Initialize num\_map = {}.
2. **Iteration 1:** num = 2, complement = 9 - 2 = 7. 7 is not in num\_map. Add 2 with its index: num\_map = {2: 0}.
3. **Iteration 2:** num = 7, complement = 9 - 7 = 2. 2 is in num\_map. Return the indices: [num\_map[2], 1], which is [0, 1].

**45. Explain the concept of priority queues and their applications in algorithm design.**

**Concept of Priority Queues:**

A priority queue is an abstract data type that operates similarly to a regular queue, but instead of retrieving elements in a first-in, first-out (FIFO) manner, it retrieves elements based on their priority. Each element in a priority queue is associated with a priority, and the element with the highest (or lowest, depending on the implementation) priority is always at the front.

Priority queues are typically implemented using data structures like heaps (binary heaps are common due to their efficiency).

**Key Operations:**

* **Insert (or enqueue):** Adds a new element with its priority to the queue.
* **Extract-Min (or dequeue for min-priority queue):** Removes and returns the element with the minimum priority.
* **Extract-Max (or dequeue for max-priority queue):** Removes and returns the element with the maximum priority.
* **Peek (or top):** Returns the element with the highest/lowest priority without removing it.
* **Decrease-Key/Increase-Key:** Allows changing the priority of an element already in the queue.

**Applications in Algorithm Design:**

Priority queues are incredibly useful in various algorithm design scenarios:

1. **Scheduling Algorithms:**
   * **Job Scheduling:** Prioritizing tasks based on urgency or importance.
   * **Operating Systems:** Managing processes based on priority levels.
2. **Graph Algorithms:**
   * **Dijkstra's Algorithm:** Finding the shortest path in a weighted graph. The priority queue stores nodes to visit, prioritized by their current shortest distance from the source.
   * **Prim's Algorithm and Kruskal's Algorithm:** Finding the Minimum Spanning Tree (MST) of a graph. Priority queues help select the edges with the minimum weights.
3. **Heapsort:** An efficient sorting algorithm that uses a binary heap to sort elements.
4. **Event Simulation:** Managing events in chronological order based on their timestamps.
5. **Data Compression (Huffman Coding):** Building the Huffman tree by repeatedly merging the nodes with the lowest frequencies.
6. **K Largest/Smallest Elements:** Efficiently finding the K largest or smallest elements in a collection.
7. **Median Maintenance:** Keeping track of the median of a dynamically changing set of numbers. Two priority queues (a min-heap for the larger half and a max-heap for the smaller half) can be used.
8. **Load Balancing:** Distributing tasks to servers based on their current load (e.g., using a min-priority queue to find the least loaded server).

In essence, any algorithm that requires repeatedly finding and processing the element with the highest or lowest priority can benefit from using a priority queue. They provide an efficient way to maintain order based on a specific criterion.

**46. 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:**

We can use a dynamic programming approach or an expand around center approach. Let's detail the expand around center approach, which is often more intuitive for this problem.

1. Initialize start (starting index of the longest palindrome found so far) to 0 and max\_len (length of the longest palindrome found so far) to 1.
2. Iterate through each character i in the string. Consider this character as the potential center of a palindrome.
3. **Odd Length Palindromes:** Expand outwards from the center i with two pointers, left = i and right = i. While left >= 0, right < len(s), and s[left] == s[right], increment right and decrement left. After the loop, the length of the palindrome found is right - left - 1. If this length is greater than max\_len, update max\_len and start.
4. **Even Length Palindromes:** Expand outwards from the potential center formed by i and i + 1 with two pointers, left = i and right = i + 1. While left >= 0, right < len(s), and s[left] == s[right], increment right and decrement left. After the loop, the length of the palindrome found is right - left - 1. If this length is greater than max\_len, update max\_len and start.
5. After iterating through all possible centers, the longest palindromic substring is s[start : start + max\_len].

**Program (Python):**

Python

def longest\_palindromic\_substring(s):

if not s:

return ""

n = len(s)

start = 0

max\_len = 1

def expand\_around\_center(left, right):

while left >= 0 and right < n and s[left] == s[right]:

left -= 1

right += 1

return right - left - 1, left + 1

for i in range(n):

# Odd length palindromes

length1, start1 = expand\_around\_center(i, i)

if length1 > max\_len:

max\_len = length1

start = start1

# Even length palindromes

length2, start2 = expand\_around\_center(i, i + 1)

if length2 > max\_len:

max\_len = length2

start = start2

return s[start : start + max\_len]

# Example

s = "babad"

longest\_palindrome = longest\_palindromic\_substring(s)

print(f"Longest palindromic substring: {longest\_palindrome}")

s = "cbbd"

longest\_palindrome = longest\_palindromic\_substring(s)

print(f"Longest palindromic substring: {longest\_palindrome}")

**Time and Space Complexities:**

* **Time Complexity:** O(n^2), where n is the length of the string. We iterate through each character as a potential center, and the expansion around the center can take O(n) in the worst case.
* **Space Complexity:** O(1), as we use a constant amount of extra space for variables.

**Example Explanation:**

For s = "babad":

* When i = 1 ('a'), expanding around it gives "bab" (length 3). start = 0, max\_len = 3.
* When i = 2 ('b'), expanding around it gives "aba" (length 3). start remains 0, max\_len remains 3.

For s = "cbbd":

* When i = 1 ('b'), expanding around i and i + 1 gives "bb" (length 2). start = 1, max\_len = 2.

**47. Explain the concept of histogram problems and their applications in algorithm design**

**Definition:**

Histogram problems typically involve analyzing a histogram (bar chart) represented as an array of bar heights to solve geometric or combinatorial problems, such as finding the **largest rectangular area** that can be formed under the histogram.

**Key Problem:**

* **Largest Rectangle in Histogram**:  
  Given an array of heights, find the largest area of a rectangle formed by adjacent bars.

**Applications in Algorithm Design:**

1. **Stack-based techniques**: Solve range queries efficiently using monotonic stacks.
2. **Matrix-based problems**: Extended to 2D matrices to find the largest rectangle of 1s.
3. **Skyline problem**: Used in computational geometry and graphics.
4. **Text editor layout/rendering**: Determines optimal block arrangements.
5. **Histogram water trapping problem**: Used to calculate how much water can be trapped between bars.

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

**Problem Statement:**

Given two singly linked lists, determine the node at which they intersect (if they do).

**Algorithm:**

1. Find the lengths of both linked lists.
2. Align the starting points by advancing the longer list by the difference in lengths.
3. Traverse both lists simultaneously; if they intersect, return the intersecting node.
4. If no intersection occurs, return None.

**Code (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

# Align both lists

tempA, tempB = headA, headB

if lenA > lenB:

for \_ in range(lenA - lenB):

tempA = tempA.next

else:

for \_ in range(lenB - lenA):

tempB = tempB.next

# Traverse and check for intersection

while tempA and tempB:

if tempA == tempB:

return tempA

tempA = tempA.next

tempB = tempB.next

return None

**Example:**

Given two lists:

* A = 1 -> 2 -> 3 -> 4 -> 5
* B = 6 -> 7 -> 3 -> 4 -> 5  
  The intersection occurs at node 3.

**Output**: The intersection node is 3.

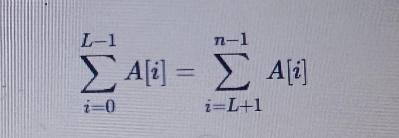
**Complexities:**

* **Time Complexity**: O(m + n), where m and n are the lengths of the two lists.
* **Space Complexity**: O(1), since no extra space is used apart from the pointers.

**50. Explain the concept of equilibrium index and its applications in array problems**

**Definition:**

An **equilibrium index** is an index in an array where the sum of elements to the left of the index is equal to the sum of elements to the right of the index.



**Applications:**

1. **Array Partitioning**: Used in problems where arrays need to be split based on sums.
2. **Load Balancing**: In applications like task scheduling or memory partitioning.
3. **Prefix and Suffix Sum Problems**: Optimization problems that involve balancing left and right sections of an array.
4. **Equilibrium in Physics or Economics**: Can represent equilibrium in systems where balances need to be maintained.