SHREYANSH SHARMA
22SCSE1012811

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

Concept

A **prefix sum array** is a data structure that allows efficient computation of the sum of elements in a subarray (a contiguous portion of an array). It is especially useful for scenarios where multiple range sum queries need to be answered quickly.

Given an array `A` of length `n`, the prefix sum array `P` is defined such that:

- `P[0] = A[0]`
- `P[1] = A[0] + A[1]`
- `P[2] = A[0] + A[1] + A[2]`
- ...
- `P[i] = A[0] + A[1] + ... + A[i]`

Example

Let `A = [3, 2, 4, 1, 5]`.

The prefix sum array `P` is:

- `P[0] = 3`
- `P[1] = 3 + 2 = 5`
- `P[2] = 5 + 4 = 9`
- `P[3] = 9 + 1 = 10`
- `P[4] = 10 + 5 = 15`

So, `P = [3, 5, 9, 10, 15]`.

Applications

1. **Range Sum Queries**: Quickly answer queries like "What is the sum of elements between indices `l` and `r`?"
2. **Cumulative Frequencies**: Used in problems involving frequency or count accumulation.
3. **Difference Arrays**: Useful in range update problems.
4. **2D Prefix Sum Arrays**: Applied in image processing or grid-based problems to compute sums in submatrices.
5. **Dynamic Programming Optimization**: Helpful in problems like subarray sums, partitioning, and rolling window calculations.

---

2. 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.

Algorithm

1. Input the array `A` of size `n`.
2. Create a prefix sum array `P` such that `P[i] = P[i-1] + A[i]`.
3. For each query `[L, R]`, compute the sum:
    - If `L == 0`: return `P[R]`.
    - Else: return `P[R] - P[L - 1]`.

Program

```
def build_prefix_sum(arr):
    n = len(arr)
    prefix = [0] * n
    prefix[0] = arr[0]
    for i in range(1, n):
        prefix[i] = prefix[i - 1] + arr[i]
    return prefix

def range_sum(prefix, L, R):
    if L == 0:
        return prefix[R]
    else:
        return prefix[R] - prefix[L - 1]

arr = [3, 2, 4, 1, 5]
prefix = build_prefix_sum(arr)
queries = [(0, 2), (1, 3), (2, 4)]
for L, R in queries:
    print(f"Sum of A[{L}..{R}] =", range_sum(prefix, L, R))
```

Complexity Analysis

- **Time Complexity**:
    - Building prefix array: O(n)
    - Each query: O(1)
    - For `q` queries: O(q)
- **Space Complexity**:

- Prefix 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.

Concept

An **equilibrium index** in an array is an index `i` such that the sum of elements before index `i` is equal to the sum of elements after index `i`.

Mathematically: For index `i`, `sum(arr[0..i-1]) == sum(arr[i+1..n-1])`

Algorithm

1. Calculate the total sum of the array.
2. Initialize `left_sum = 0`.
3. Iterate through the array:
   - At each index `i`, subtract `arr[i]` from `total_sum` to get the right sum.
   - If `left_sum == right_sum`, return `i` as the equilibrium index.
   - Add `arr[i]` to `left_sum` for the next iteration.
4. If no index found, return `-1`.

Program

```
def find_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 = [-7, 1, 5, 2, -4, 3, 0]
index = find_equilibrium_index(arr)
print(f"Equilibrium Index: {index}")
```

Example Explained

Given `arr = [-7, 1, 5, 2, -4, 3, 0]`

Total sum = 0

At `i = 3`:

- Left sum = -7 + 1 + 5 = -1
- Right sum = -4 + 3 + 0 = -1

They match → Equilibrium index is 3

Complexity Analysis

- **Time Complexity**:
  - One pass to get total sum: O(n)
  - One pass to find equilibrium: O(n)
  - Total: O(n)
- **Space Complexity**:
  - No extra space used apart from variables: 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. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Compute the total sum of the array.
2. Initialize `left_sum = 0`.
3. Traverse the array:
   - Subtract the current element from the total to get the right sum.
   - If `left_sum == right_sum`, the array can be split.
   - Add the current element to `left_sum` for the next iteration.
4. If no such point found, return `False`.

Program

```
def can_be_split(arr):
    total_sum = sum(arr)
    left_sum = 0
    for i in range(len(arr)):
        right_sum = total_sum - left_sum - arr[i]
```

```
        if left_sum == right_sum:
            return True
        left_sum += arr[i]
    return False

arr1 = [1, 2, 3, 3]
arr2 = [1, 2, 3, 4, 5]
print("Can arr1 be split?", can_be_split(arr1))
print("Can arr2 be split?", can_be_split(arr2))
```

Example

For `arr = [1, 2, 3, 3]`

Total sum = 9

At `i = 2`:

- Left sum = 1 + 2 = 3
- Right sum = 9 - 3 - 3 = 3

`left_sum == right_sum`

Complexity Analysis

- **Time Complexity**:
  - One traversal for sum: O(n)
  - One pass to check split point: O(n)
  - Total: O(n)
- **Space Complexity**:
  - No extra data structures used: O(1)

---

5. Find the maximum sum of any subarray of size K in a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Calculate the sum of the first `K` elements. This is your initial window.
2. Slide the window one element at a time:
   - Subtract the element leaving the window.

    - Add the new element entering the window.
    - Update the maximum sum if the current window sum is greater.
3. Return the maximum sum.

Program

```
def max_sum_subarray_k(arr, k):
    n = len(arr)
    if n < k:
        return "Invalid input: K is larger than array size"
    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("Maximum sum of subarray of size", k, "is:", max_sum_subarray_k(arr, k))
```

Example

For `arr = [2, 1, 5, 1, 3, 2]` and `K = 3`

Subarrays of size 3:

- `[2, 1, 5]` → sum = 8
- `[1, 5, 1]` → sum = 7
- `[5, 1, 3]` → sum = 9
- `[1, 3, 2]` → sum = 6

Maximum = 9

Complexity Analysis

- **Time Complexity**:
    - Initial sum: O(K)
    - Sliding the window: O(n - K)
    - Total: O(n)
- **Space Complexity**:
    - Only variables used: O(1)

---

6. Find the length of the longest substring without repeating characters. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a set to track characters in the current window.
2. Use two pointers `left` and `right` to maintain the window.
3. Move the `right` pointer forward:
    - If the character is not in the set, add it and update the max length.
    - If it is in the set, remove characters from the left until it's gone.
4. Continue until `right` reaches the end of the string.

Program

```
def length_of_longest_substring(s):
    char_set = set()
    left = 0
    max_len = 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_len = max(max_len, right - left + 1)
    return max_len

s = "abcabcbb"
print("Length of longest substring without repeating characters:",
length_of_longest_substring(s))
```

Example

For `s = "abcabcbb"`:

Substrings without repeating:

- `"abc"` → length 3
- `"bca"` → length 3
- `"cab"` → length 3
- `"abcb"` → has repeat → length 3

Maximum = 3

Complexity Analysis

- **Time Complexity**:
  - Each character is visited at most twice (once by `right`, once by `left`) → O(n)
- **Space Complexity**:
  - Set of characters → O(min(n, m))
  - where `n` is string length, `m` is character set size (like 26 for lowercase letters, 128 for ASCII)

---

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

Concept

The **sliding window** is a range-based approach where we maintain a "window" that moves across the data structure (like a string or array). The window can expand or shrink based on the problem conditions.

Instead of recalculating values from scratch for every possible subarray or substring, we re-use previous computations smartly, which greatly reduces time complexity.

Algorithm

1. Initialize two pointers (usually `left` and `right`) to define the current window.
2. Move the `right` pointer to expand the window (e.g., include more characters).
3. Move the `left` pointer to shrink the window when a condition is violated (e.g., duplicates found).
4. Update the result based on the current window size or content.

Common Use Cases in String Problems

1. Longest Substring Without Repeating Characters

Problem: Find the longest substring with unique characters.

Use of sliding window:

- Expand `right` until a repeat is found.
- Shrink from the `left` until the repeat is removed.
- Efficiency: O(n)

2. Minimum Window Substring

Problem: Find the smallest substring of `s` that contains all characters of `t`.

Use of sliding window:

- Expand `right` to include needed characters.
- Shrink `left` to minimize the window while still containing all characters.
- Efficiency: O(n + m)

3. Anagrams in a String

Problem: Find all anagrams of `p` in `s`.

Use of sliding window:

- Use a frequency map for characters.
- Slide window and compare frequency maps.
- Efficiency: O(n)

---

8. 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. Iterate over each character in the string and treat it as the center.
2. Expand around the center, checking both left and right directions for matching characters.
3. Track the longest palindrome found during the expansion.
4. Return the longest palindromic substring.

Program

```
def expand_around_center(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return s[left+1:right]

def longest_palindromic_substring(s):
    if not s:
```

```
        return ""
    longest = ""
    for i in range(len(s)):
        palindrome1 = expand_around_center(s, i, i)
        palindrome2 = expand_around_center(s, i, i + 1)
        longest = max(longest, palindrome1, palindrome2, key=len)
    return longest

s = "babad"
print("Longest palindromic substring:", longest_palindromic_substring(s))
```

Example

For `s = "babad"`:

Expanding from index 2 (centered at 'a'):

- Odd-length palindrome: "aba"
- Even-length palindrome: "bb"

The longest palindrome found is "aba".

Complexity Analysis

- **Time Complexity**:
  - The function `expand_around_center` takes O(n) for each center, where `n` is the length of the string.
  - We perform this expansion for every character in the string, so the total time complexity is $O(n^2)$.
- **Space Complexity**:
  - We only use a few extra variables, so the space complexity is O(1), excluding the input string.

---

9. Find the longest common prefix among a list of strings. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Start by assuming that the first string in the list is the longest common prefix.
2. Iterate through the rest of the strings:

    - Compare the current string with the prefix.
    - If the current string doesn't match the prefix, shorten the prefix (cutting off characters from the end) until it matches.
3. If the prefix becomes empty at any point, return `""` (empty string).
4. The resulting prefix after all comparisons is the longest common prefix.

Program

```
def longest_common_prefix(strs):
    if not strs:
        return ""
    prefix = strs[0]
    for string in strs[1:]:
        while not string.startswith(prefix):
            prefix = prefix[:-1]
        if not prefix:
            return ""
    return prefix
```

Example

For `strs = ["flower", "flow", "flight"]`:

1. Start with the first string "flower".
2. Compare it with the second string "flow". The common prefix is "flow", so continue.
3. Compare it with the third string "flight". The common prefix is "fl", so shorten the prefix.
4. The longest common prefix is "fl".

Complexity Analysis

- **Time Complexity**:
    - The worst case is when we compare all strings with each other. For each string, we compare it character by character with the prefix.
    - If `n` is the number of strings and `m` is the length of the longest string, the time complexity is **O(n * m)**.
- **Space Complexity**:
    - We are only using a few extra variables for the prefix and string comparison, so the space complexity is **O(1)**, excluding the input list.

---

10. Generate all permutations of a given string. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Convert the string into a list of characters for easy swapping.
2. Use a recursive function that:
    - Fixes one character at the current index.
    - Recursively permutes the rest.
    - Swaps characters back to restore the original state (backtrack).
3. To avoid duplicate permutations (if the string contains duplicates), use a set.

Program

```
def permute(s):
    def backtrack(start):
        if start == len(chars):
            permutations.append("".join(chars))
            return
        seen = set()
        for i in range(start, len(chars)):
            if chars[i] in seen:
                continue
            seen.add(chars[i])
            chars[start], chars[i] = chars[i], chars[start]
            backtrack(start + 1)
            chars[start], chars[i] = chars[i], chars[start]

    chars = list(s)
    permutations = []
    backtrack(0)
    return permutations
```

Example

For `s = "abc"`:

- Start with index 0:
    - Fix `a`, permute `bc` → `abc`, `acb`
    - Fix `b`, permute `ac` → `bac`, `bca`
    - Fix `c`, permute `ab` → `cab`, `cba`

All 6 permutations are generated (3! = 6).

Complexity Analysis

- **Time Complexity**:
    - There are `n!` permutations for a string of length `n`.
    - Each permutation takes O(n) time to build (due to `"".join`).

    - Total time: **O(n × n!)**
- **Space Complexity**:
    - Output space for storing all permutations: **O(n × n!)**
    - Recursion stack: **O(n)**

---

11. 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. Initialize two pointers:
    - `left` at the start of the array
    - `right` at the end of the array
2. While `left < right`:
    - Compute the sum of the two elements.
    - If the sum is equal to the target → return the pair (or indices).
    - If the sum is less than the target → move `left` forward.
    - If the sum is greater than the target → move `right` backward.
3. If no such pair is found, return an indicator like `None`.

Program

```
def two_sum_sorted(arr, target):
    left = 0
    right = len(arr) - 1
    while left < right:
        curr_sum = arr[left] + arr[right]
        if curr_sum == target:
            return (arr[left], arr[right])
        elif curr_sum < target:
            left += 1
        else:
            right -= 1
    return None


arr = [1, 2, 3, 4, 6, 8, 11]
target = 10
print("Pair with target sum:", two_sum_sorted(arr, target))
```

Example

Given `arr = [1, 2, 3, 4, 6, 8, 11]` and `target = 10`:

- Try `1 + 11 = 12` → too big → move `right` left
- Try `1 + 8 = 9` → too small → move `left` right
- Try `2 + 8 = 10` found the pair!

Complexity Analysis

- **Time Complexity**:
  - Each element is considered at most once → **O(n)**
- **Space Complexity**:
  - No extra space used → **O(1)**

---

12. 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 first decreasing element from the end**:
   - Start from the end and find the **first `i` such that `arr[i] < arr[i + 1]`.**
   - If no such `i` exists, the permutation is the **last** one → reverse the whole array.
2. **Find the element just larger than `arr[i]`** to its right:
   - Scan from the end to find **`j` such that `arr[j] > arr[i]`.**
3. **Swap `arr[i]` and `arr[j]`.**
4. **Reverse the subarray from `i + 1` to the end**.

Program

```
def next_permutation(arr):
    n = len(arr)

    i = n - 2
    while i >= 0 and arr[i] >= arr[i + 1]:
        i -= 1
    if i >= 0:
```

```
        j = n - 1
        while arr[j] <= arr[i]:
            j -= 1

        arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1:] = reversed(arr[i + 1:])
    return arr


arr = [1, 2, 3]
print("Next permutation:", next_permutation(arr))
```

Example Explained

For `arr = [1, 2, 3]`:

- Pivot: `2` (because `2 < 3`)
- Next larger element to the right: `3`
- Swap: `1, 3, 2`
- No need to reverse since the suffix is already smallest

Next permutation: `[1, 3, 2]`

For `arr = [3, 2, 1]`, it's the **last permutation**, so output: `[1, 2, 3]`.

Complexity Analysis

- **Time Complexity**:
  - Finding the pivot and successor: O(n)
  - Reversing the suffix: O(n)
  - Total: **O(n)**
- **Space Complexity**:
  - In-place rearrangement → **O(1)**

---

13. 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 (Iterative Approach)

1. Create a **dummy node** to act as the starting point.
2. Use a pointer (`tail`) to build the new list.
3. While both lists are non-empty:
    - Compare the current nodes of both lists.
    - Append the smaller one to the result list.
    - Move the pointer in the list you took the node from.
4. Once one list is exhausted, attach the remaining part of the other list.
5. Return the node following the dummy.

Program

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

def merge_sorted_lists(l1, l2):
    dummy = ListNode()
    tail = dummy
    while l1 and l2:
        if l1.val < l2.val:
            tail.next, l1 = l1, l1.next
        else:
            tail.next, l2 = l2, l2.next
        tail = tail.next
    tail.next = l1 or l2
    return dummy.next
```

Example

Input:

- `l1 = 1 → 3 → 5`
- `l2 = 2 → 4 → 6`

Output:

- `1 → 2 → 3 → 4 → 5 → 6`

Time & Space Complexity

- **Time**: O(m + n) — Every node in both lists is visited once.
- **Space**: O(1) — No extra memory used (in-place merging using existing nodes).

---

14. 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. Ensure `nums1` is the **smaller array**.
2. Let `x = len(nums1), y = len(nums2).`
3. Perform binary search on `nums1`:
    - `low = 0, high = x`

    Partition both arrays:

    - `partitionX = (low + high) // 2`
    - `partitionY = (x + y + 1) // 2 - partitionX`
4. Compare the left and right max/min values:
    - If `maxLeftX <= minRightY` and `maxLeftY <= minRightX`:
        - Found correct partition:
            - If total length even → median = `(max(left) + min(right)) / 2`
            - If odd → median = `max(left)`
    - Else adjust binary search range.

Program

```
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
        maxLeftX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        minRightX = float('inf') if partitionX == x else nums1[partitionX]
        maxLeftY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
        minRightY = float('inf') if partitionY == y else nums2[partitionY]
        if maxLeftX <= minRightY and maxLeftY <= minRightX:
            if (x + y) % 2 == 0:
                return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2
            else:
                return max(maxLeftX, maxLeftY)
        elif maxLeftX > minRightY:
```

```
            high = partitionX - 1
        else:
            low = partitionX + 1
```

Complexity Analysis

- **Time Complexity**:
  - O(log(min(m, n))) — Binary search is done on the **smaller** array.
- **Space Complexity**:
  - O(1) — No extra space used.

---

15. 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 (Binary Search on Values)

1. Set `low = matrix[0][0]`, `high = matrix[n-1][n-1]`.
2. While `low < high`:
   - Set `mid = (low + high) // 2`.
   - Count how many numbers are ≤ `mid` using a helper function.
   - If `count < k`, set `low = mid + 1`.
   - Else, set `high = mid`.
3. When `low == high`, return `low`.

Program

```
def count_less_equal(matrix, mid, n):
    count = 0
    row = n - 1
    col = 0
    while row >= 0 and col < n:
        if matrix[row][col] <= mid:
            count += row + 1
            col += 1
        else:
            row -= 1
    return count

def kthSmallest(matrix, k):
    n = len(matrix)
    low = matrix[0][0]
```

```
    high = matrix[n-1][n-1]
    while low < high:
        mid = (low + high) // 2
        if count_less_equal(matrix, mid, n) < k:
            low = mid + 1
        else:
            high = mid
    return low
```

Example

Given the matrix:

```
[1, 5, 9] [10, 11, 13] [12, 13, 15]
```

Let's find the 8th smallest element.

1. Values range from 1 to 15.
2. Binary search between 1 and 15:
    - `mid = 8` → count = 3 (only 1, 5, 9)
    - `mid = 12` → count = 6
    - `mid = 13` → count = 8 (found)
3. Result: **13** is the 8th smallest element.

Time Complexity

- Each iteration of binary search: O(n) (for the `count_less_equal`)
- Number of iterations: log(max−min)
- **Total time complexity**: O(n·log(max−min))

Space Complexity

- No extra space used (besides variables): **O(1)**

---

16. 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: Boyer-Moore Voting Algorithm

1. Set a variable called "candidate" to `None` and "count" to zero.
2. Loop through each number in the array:
    - If `count` is zero, set the current number as the candidate.
    - If the current number is equal to the candidate, increase `count` by one.
    - Otherwise, decrease `count` by one.
3. After finishing the loop, the value stored in "candidate" will be the majority element.

This works only if there is guaranteed to be a majority element (an element that appears more than half the times in the array).

Program

```
def majority_element(nums):
    count = 0
    candidate = None
    for num in nums:
        if count == 0:
            candidate = num
        if num == candidate:
            count += 1
        else:
            count -= 1
    return candidate
```

Time Complexity

- The function goes through the list only once, so the time complexity is **O(n)** where **n** is the number of elements in the list.

Space Complexity

- The function uses only a few variables, so the space complexity is **O(1)** (constant space).

---

17. 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. **Initialization**:
    - Create two arrays `left_max[]` and `right_max[]` of the same size as the input height array.
    - Set a variable `water_trapped` to 0.
2. **Fill the `left_max[]` array**:
    - For each index `i`, compute the maximum of `left_max[i-1]` and `height[i]`.
3. **Fill the `right_max[]` array**:
    - For each index `i`, compute the maximum of `right_max[i+1]` and `height[i]`.
4. **Compute water trapped**:
    - For each index `i`, compute the trapped water as the difference between the minimum of `left_max[i]` and `right_max[i]`, and the height of the bar at `i`. Add this to the total `water_trapped`.

Program

```
def trap(height):
    n = len(height)
    if n == 0:
        return 0

    right_max = [0] * n
    water_trapped = 0

    left_max[0] = height[0]
    for i in range(1, n):
        left_max[i] = max(left_max[i - 1], height[i])

    right_max[n - 1] = height[n - 1]
    for i in range(n - 2, -1, -1):
        right_max[i] = max(right_max[i + 1], height[i])

    for i in range(n):
        water_trapped += min(left_max[i], right_max[i]) - height[i]
    return water_trapped
```

Example

Let's walk through an example:

Given the array `height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]`.

- **Left Max Array**: `[0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3]`

- **Right Max Array**: `[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 1]`
- **Trapped Water Calculation**: For each index, the water trapped is calculated as:
  - `min(left_max[i], right_max[i]) - height[i]`
  - For index 0, `min(0, 3) - 0 = 0`
  - For index 1, `min(1, 3) - 1 = 0`
  - For index 2, `min(1, 3) - 0 = 1`
  - For index 3, `min(2, 3) - 2 = 0`
  - Continue for all indices.

Summing these values gives the total trapped water: 6.

Time Complexity

- **Filling the `left_max[]` array** takes O(n).
- **Filling the `right_max[]` array** takes O(n).
- **Calculating the trapped water** takes O(n).

Thus, the overall **time complexity** is O(n), where `n` is the length of the input array.

Space Complexity

- The space complexity is O(n) for storing the `left_max` and `right_max` arrays.

---

18. 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. **Trie Representation**:
   - Use a Trie to store binary representations of numbers.
   - Each node has two children: one for `0` and one for `1`.
2. **Insert Numbers into the Trie**:
   - Convert each number to its binary form and insert it into the Trie.
3. **Find Maximum XOR**:
   - For each number, find the best XOR by trying to take the opposite bit at each position in the Trie.

Program

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

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

    def insert(self, num):
        node = self.root
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

    def find_max_xor(self, num):
        node = self.root
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            opposite_bit = 1 - bit
            if opposite_bit in node.children:
                max_xor |= (1 << i)
                node = node.children[opposite_bit]
            else:
                node = node.children[bit]
        return max_xor

def find_maximum_xor(nums):
    trie = Trie()
    max_xor = 0
    for num in nums:
        trie.insert(num)
    for num in nums:
        max_xor = max(max_xor, trie.find_max_xor(num))
    return max_xor
```

Time Complexity

- **Insert and find operations** take O(32) each (for 32-bit integers).
- **Overall Complexity**: O(n), where `n` is the number of elements.

Space Complexity

- **Trie Space**: O(n), due to the Trie storing all bit representations of numbers.

Example

For `nums = [3, 10, 5, 25, 2, 8]`, the maximum XOR is 28. This is achieved by XORing 3 and 25.

Summary

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

---

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

Algorithm

1. Initialize `max_product` and `min_product` to the first element of the array, and `result` to that element as well.
2. For each subsequent element, compute the possible values of `max_product` and `min_product`:
    - If the current element is negative, swap `max_product` and `min_product`.
    - Update `max_product` to the maximum of:
        - The current element itself.
        - The current element multiplied by the previous `max_product`.
        - The current element multiplied by the previous `min_product`.
    - Update `min_product` similarly.
3. Update the result to be the maximum of the current result and `max_product`.

Program

```
def maxProduct(nums):
    if not nums:
        return 0

    max_product = min_product = result = nums[0]
    for num in nums[1:]:

        if num < 0:
            max_product, min_product = min_product, max_product
```

```
    max_product = max(num, max_product * num)
    min_product = min(num, min_product * num)

    result = max(result, max_product)
  return result
```

Example

Let's consider an example where `nums = [2, 3, -2, 4]`.

1. Initialize: `max_product = 2`, `min_product = 2`, `result = 2`.
2. Process the second element `3`:
    - `max_product = max(3, 2 * 3) = 6`
    - `min_product = min(3, 2 * 3) = 3`
    - `result = max(2, 6) = 6`
3. Process the third element `-2`:
    - Swap `max_product` and `min_product` because the element is negative.
    - `max_product = max(-2, 3 * -2) = -2`
    - `min_product = min(-2, 6 * -2) = -12`
    - `result = max(6, -2) = 6`
4. Process the fourth element `4`:
    - `max_product = max(4, -2 * 4) = 4`
    - `min_product = min(4, -12 * 4) = -48`
    - `result = max(6, 4) = 6`

The final result is 6.

Time Complexity

- **O(n)**

Space Complexity

- **O(1)**

---

20. 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. If `n > 10`, return 0 because we cannot have more than 10 unique digits.

2. Initialize a result variable as 9 (for the first digit).
3. For each subsequent digit from the 2nd to the `n`-th digit:
    - Multiply the result by `10 - i`, where `i` is the current digit position (starting from 1).
4. Return the final result.

Program

```
def count_numbers_with_unique_digits(n):
    if n == 0:
        return 1
    if n > 10:
        return 0
    result = 9
    product = 9
    for i in range(1, n):
        product *= (10 - i)
    result += product
    return result
```

Time Complexity

- **O(n)**

Space Complexity

- **O(1)**

---

21. How to count the number of 1s in the binary representation of numbers from 0 to n. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Initialize a list `count` of size `n+1` where each element is initialized to 0.
2. For each number `i` from 1 to `n`, calculate `count[i]` as:
    - `count[i] = count[i >> 1] + (i & 1)`

3. After filling the `count` array, sum all the values from `count[0]` to `count[n]` to get the total number of 1s in the binary representations of all numbers from 0 to `n`.

Program

```
def count_ones(n):
    count = [0] * (n + 1)
    for i in range(1, n + 1):
        count[i] = count[i >> 1] + (i & 1)
    return sum(count)

n = 5
print(f"Total number of 1s in binary representation of numbers from 0 to {n} is:
{count_ones(n)}")
```

Example

Let's consider `n = 5`.

The binary representations of numbers from 0 to 5 are:

- 0: `0` → 0 ones
- 1: `1` → 1 one
- 2: `10` → 1 one
- 3: `11` → 2 ones
- 4: `100` → 1 one
- 5: `101` → 2 ones

So, the total number of 1s is 0 + 1 + 1 + 2 + 1 + 2 = 7.

Thus, the result for `n = 5` is 7.

Time Complexity

- **Time Complexity**: O(n), because we loop through all numbers from 1 to `n`, and for each number, we perform constant-time operations (bit-shifting and addition).

Space Complexity

- **Space Complexity**: O(n), because we store the count of 1s for each number from 0 to `n` in the `count` array.

---

22. How to check if a number is a power of two using bit manipulation. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. If the number is less than or equal to zero, it cannot be a power of two.
2. For a positive number, check if `n & (n - 1)` equals 0. This will be true only if the number is a power of two.

Program

```
def is_power_of_two(n):

    if n <= 0:
        return False
    return (n & (n - 1)) == 0


n = 16
if is_power_of_two(n):
    print(f"{n} is a power of two.")
else:
    print(f"{n} is not a power of two.")
```

Example

- For `n = 16`, `n - 1 = 15`, and `n & (n - 1)` results in `16 & 15 = 0`, which confirms that 16 is a power of two. Thus, the output will be:

  ```
  16 is a power of two.
  ```

Time Complexity

- **O(1)**

Space Complexity

- **O(1)**

---

23. How to 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. **Trie Representation**:
   - Use a Trie to store binary representations of numbers.
   - Each node has two children: one for `0` and one for `1`.
2. **Insert Numbers into the Trie**:
   - Convert each number to its binary form and insert it into the Trie.
3. **Find Maximum XOR**:
   - For each number, find the best XOR by trying to take the opposite bit at each position in the Trie.

Program

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

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

    def insert(self, num):
        node = self.root
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            if bit not in node.children:
                node.children[bit] = TrieNode()
            node = node.children[bit]

    def find_max_xor(self, num):
        node = self.root
        max_xor = 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1
            opposite_bit = 1 - bit
```

```
            if opposite_bit in node.children:
                max_xor |= (1 << i)
                node = node.children[opposite_bit]
            else:
                node = node.children[bit]
        return max_xor

def find_maximum_xor(nums):
    trie = Trie()
    max_xor = 0
    for num in nums:
        trie.insert(num)
    for num in nums:
        max_xor = max(max_xor, trie.find_max_xor(num))
    return max_xor
```

Time Complexity

- **Insert and find operations** take O(32) each (for 32-bit integers).
- **Overall Complexity**: O(n), where `n` is the number of elements.

Space Complexity

- **Trie Space**: O(n), due to the Trie storing all bit representations of numbers.

Example

For `nums = [3, 10, 5, 25, 2, 8]`, the maximum XOR is 28. This is achieved by XORing 3 and 25.

Summary

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

---

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

Concept

Bit manipulation refers to the process of directly manipulating bits or binary digits of data. Bits are the most basic unit of information in computing, represented as either `0` or `1`. Bit

manipulation involves performing operations on these bits, and it is a fundamental concept in computer science and programming.

In bit manipulation, operations are typically performed using **bitwise operators**, which operate on the individual bits of integers. These operators allow for efficient manipulation and checking of individual bits in numbers.

Common Bitwise Operators

1. **AND (`&`)**:
   - Compares each bit of two numbers. The result is `1` if both corresponding bits are `1`, and `0` otherwise.
   - Example: `5 & 3 → 0101 & 0011 = 0001` (Result: 1)
2. **OR (`|`)**:
   - Compares each bit of two numbers. The result is `1` if at least one of the corresponding bits is `1`.
   - Example: `5 | 3 → 0101 | 0011 = 0111` (Result: 7)
3. **XOR (`^`)**:
   - Compares each bit of two numbers. The result is `1` if the bits are different, and `0` if they are the same.
   - Example: `5 ^ 3 → 0101 ^ 0011 = 0110` (Result: 6)
4. **NOT (`~`)**:
   - Flips all the bits of a number (also known as bitwise negation).
   - Example: `~5 → ~0101 = 1010` (Result: -6 in two's complement form)
5. **Left Shift (`<<`)**:
   - Shifts the bits of a number to the left, filling the rightmost bits with zeros. It essentially multiplies the number by a power of 2.
   - Example: `5 << 1 → 0101 << 1 = 1010` (Result: 10)
6. **Right Shift (`>>`)**:
   - Shifts the bits of a number to the right, filling the leftmost bits with the sign bit (for signed integers) or zeros (for unsigned integers). It essentially divides the number by a power of 2.
   - Example: `5 >> 1 → 0101 >> 1 = 0010` (Result: 2)

Advantages of Bit Manipulation in Algorithm Design

1. **Efficiency**:
   - **Fast Operations**: Bitwise operations are generally much faster than arithmetic operations, as they directly manipulate the binary representations of numbers. Modern processors are optimized for bit-level operations, making them much more efficient for certain tasks, especially in low-level programming.
2. **Memory Efficiency**:
   - **Compact Storage**: Bit manipulation allows storing multiple boolean values in a single integer. For example, using an integer (32 bits in a 32-bit machine) to store up to 32 boolean values can save a lot of memory compared to using an array of booleans. This is particularly useful in memory-constrained systems.

3. **Problem Solving**:
    - **Binary Representation**: Many problems are naturally represented in binary. For example, checking whether a number is a power of two, counting the number of set bits (1s) in an integer, or finding the number of trailing zeros are problems that can be solved using bitwise manipulation in a very efficient manner.

---

25. Solve the problem of finding the next greater element for each element in an array.
Write its algorithm, program. Find its time and space complexities. Explain with
suitable example.

Algorithm

1. Initialize an empty stack.
2. Traverse the array from right to left.
3. For each element, while the stack is not empty and the top of the stack is less than or equal to the current element, pop the stack.
4. If the stack is empty, the next greater element is `-1`. Otherwise, the next greater element is the top of the stack.
5. Push the current element onto the stack.
6. Repeat steps 3-5 for all elements.

Program

```
def next_greater_elements(nums):
    stack = []
    result = [-1] * len(nums)
    for i in range(len(nums) - 1, -1, -1):
        while stack and stack[-1] <= nums[i]:
            stack.pop()
        if stack:
            result[i] = stack[-1]
        stack.append(nums[i])
    return result


nums = [4, 1, 2, 1, 3]
print(next_greater_elements(nums))  # Output: [-1, 3, 3, 3, -1]
```

Time Complexity

- **O(n)**, where `n` is the number of elements in the array. Each element is pushed and popped from the stack at most once.

Space Complexity

- **O(n)**, for the stack and the result array.

Example

For the array `[4, 1, 2, 1, 3]`, the next greater elements are `[-1, 3, 3, 3, -1]`.

---

26. Remove the n-th node from the end of a singly linked list. Write its algorithm, program.
Find its time and space complexities. Explain with suitable example.

Algorithm

1. Initialize two pointers, `first` and `second`, both pointing to the head of the list.
2. Move the `first` pointer `n` steps ahead.
3. Move both pointers one step at a time until the `first` pointer reaches the end of the list.
4. The `second` pointer will be just before the node to be removed.
5. Remove the node by updating the next pointer of the `second` pointer.

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
    first = dummy
    second = dummy
    for _ in range(n + 1):
        first = first.next
    while first:
        first = first.next
        second = second.next
```

```
    second.next = second.next.next
    return dummy.next


head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
n = 2
new_head = remove_nth_from_end(head, n)
```

Time Complexity

- **O(L)**, where `L` is the length of the linked list. We traverse the list once.

Space Complexity

- **O(1)**, as we are using a constant amount of extra space.

Example

For the linked list `1 -> 2 -> 3 -> 4 -> 5` and `n = 2`, the resulting list is `1 -> 2 -> 3 -> 5`.

---

27. Find the node where two singly linked lists intersect. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Traverse both lists to find their lengths.
2. Align the starting points of both lists by moving the pointer of the longer list ahead by the difference in lengths.
3. Traverse both lists together until you find the intersection point.

Program

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

def get_intersection_node(headA, headB):
    if not headA or not headB:
```

```
        return None
    pa, pb = headA, headB
    while pa is not pb:
        pa = pa.next if pa else headB
        pb = pb.next if pb else headA
    return pa


common = ListNode(8, ListNode(10))
headA = ListNode(4, ListNode(1, common))
headB = ListNode(5, ListNode(6, ListNode(1, common)))
intersection = get_intersection_node(headA, headB)
```

Time Complexity

- **O(L1 + L2)**, where `L1` and `L2` are the lengths of the two linked lists.

Space Complexity

- **O(1)**, as we are using a constant amount of extra space.

Example

For the linked lists `4 -> 1 -> 8 -> 10` and `5 -> 6 -> 1 -> 8 -> 10`, the intersecting node is `8`.

---

28. Implement two stacks in a single array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use two pointers, `top1` and `top2`, to keep track of the top of each stack.
2. `top1` starts at the beginning of the array and grows towards the end.
3. `top2` starts at the end of the array and grows towards the beginning.
4. Ensure that the two stacks do not overlap.

Program

```
class TwoStacks:
    def __init__(self, n):
        self.size = n
```

```
        self.arr = [0] * n
        self.top1 = -1
        self.top2 = n

    def push1(self, x):
        if self.top1 < self.top2 - 1:
            self.top1 += 1
            self.arr[self.top1] = x
        else:
            print("Stack Overflow")

    def push2(self, x):
        if self.top1 < self.top2 - 1:
            self.top2 -= 1
            self.arr[self.top2] = x
        else:
            print("Stack Overflow")

    def pop1(self):
        if self.top1 >= 0:
            x = self.arr[self.top1]
            self.top1 -= 1
            return x
        else:
            print("Stack Underflow")
            return -1

    def pop2(self):
        if self.top2 < self.size:
            x = self.arr[self.top2]
            self.top2 += 1
            return x
        else:
            print("Stack Underflow")
            return -1


ts = TwoStacks(5)
ts.push1(5)
ts.push2(10)
ts.push2(15)
ts.push1(11)
ts.push2(7)
print(ts.pop1())
print(ts.pop2())
```

Time Complexity

- **O(1)** for push and pop operations.

Space Complexity

- **O(n)**, where `n` is the size of the array.

Example

For an array of size 5, we can push and pop elements from both stacks without overlap.

---

29. Write a program to check if an integer is a palindrome without converting it to a string.
Write its algorithm, program. Find its time and space complexities. Explain with
suitable example.

Algorithm

1. If the number is negative, return `False`.
2. Reverse the integer and compare it with the original integer.
3. If they are the same, the number is a palindrome.

Program

```
def is_palindrome(x):
    if x < 0:
        return False
    original = x
    reversed_num = 0
    while x > 0:
        digit = x % 10
        reversed_num = reversed_num * 10 + digit
        x //= 10
    return original == reversed_num


print(is_palindrome(121))  # Output: True
```

```
print(is_palindrome(-121))  # Output: False
print(is_palindrome(10))  # Output: False
```

Time Complexity

- **O(log10(n))**, where `n` is the number of digits in the integer.

Space Complexity

- **O(1)**, as we are using a constant amount of extra space.

Example

For the integer `121`, the function returns `True` because `121` is a palindrome.

---

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

Concept

A **linked list** is a linear data structure where each element is a separate object. Each element (or node) of a list consists of at least two parts: a data field and a reference (or link) to the next node in the sequence. The last node has a reference to `null`.

Applications

1. **Dynamic Memory Allocation**: Linked lists allow for efficient memory usage as they allocate memory as needed.
2. **Implementing Other Data Structures**: Linked lists are used to implement stacks, queues, and graphs.
3. **Efficient Insertions/Deletions**: Linked lists allow for efficient insertion and deletion of elements, especially when the position is known.
4. **Symbolic Computation**: Linked lists are used in symbolic computation for representing mathematical expressions.
5. **Sparse Matrices**: Linked lists are used to represent sparse matrices efficiently.

Example

A simple linked list `1 -> 2 -> 3` can be used to represent a sequence of numbers with efficient insertion and deletion operations.

---

31. Use a deque to find the maximum in every sliding window of size K. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a deque to store the indices of the elements.
2. Traverse the array and for each element, remove elements from the front of the deque if they are out of the current window.
3. Remove elements from the back of the deque if they are smaller than the current element.
4. Add the current element to the deque.
5. The element at the front of the deque is the maximum for the current window.

Program

```
from collections import deque

def max_sliding_window(nums, k):
    if not nums or k == 0:
        return []
    deq = deque()
    result = []
    for i in range(len(nums)):

        if deq and deq[0] == i - k:
            deq.popleft()

        while deq and nums[deq[-1]] < nums[i]:
            deq.pop()

        deq.append(i)

        if i >= k - 1:
            result.append(nums[deq[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)**, where `n` is the number of elements in the array. Each element is added and removed from the deque at most once.

Space Complexity

- **O(k)**, for the deque storing the indices of the elements.

Example

For the array `[1, 3, -1, -3, 5, 3, 6, 7]` and `k = 3`, the maximums for each window are `[3, 3, 5, 5, 6, 7]`.

---

32. How to find the largest rectangle that can be formed in a histogram. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a stack to store the indices of the bars.
2. Traverse the array and for each bar, push its index onto the stack if it is taller than the bar at the index stored at the top of the stack.
3. If the current bar is shorter, pop the stack and calculate the area with the popped index as the smallest (or minimum height) bar.
4. Update the maximum area.
5. Repeat until all bars are processed.

Program

```
def largest_rectangle_area(heights):
    stack = []
    max_area = 0
    index = 0
    while index < len(heights):
        if not stack or heights[index] >= heights[stack[-1]]:
            stack.append(index)
            index += 1
        else:
            top_of_stack = stack.pop()
            height = heights[top_of_stack]
```

```
            width = index if not stack else index - stack[-1] - 1
            max_area = max(max_area, height * width)
    while stack:
        top_of_stack = stack.pop()
        height = heights[top_of_stack]
        width = index if not stack else index - stack[-1] - 1
        max_area = max(max_area, height * width)
    return max_area


heights = [2, 1, 5, 6, 2, 3]
print(largest_rectangle_area(heights))  # Output: 10
```

Time Complexity

- **O(n)**, where `n` is the number of bars in the histogram. Each bar is pushed and popped from the stack at most once.

Space Complexity

- **O(n)**, for the stack storing the indices of the bars.

Example

For the histogram `[2, 1, 5, 6, 2, 3]`, the largest rectangle has an area of 10.

---

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

Concept

The **sliding window** technique is used to efficiently solve problems involving contiguous subarrays or subsequences. It involves maintaining a "window" that moves across the array while updating the result.

Applications

1. **Maximum/Minimum Sum Subarray of Size K**: Find the subarray of size `K` with the maximum or minimum sum.
2. **Longest Substring with K Distinct Characters**: Find the longest substring with at most `K` distinct characters.

3. **First Negative Number in Every Window of Size K**: Find the first negative number in every window of size `K`.

Example

For the array `[1, 2, 3, 4, 5]` and `k = 3`, the sliding window technique can be used to find the maximum sum of any contiguous subarray of size 3, which is 12 (3 + 4 + 5).

---

34. Solve the problem of finding the subarray sum equal to K using hashing. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a hashmap to store the cumulative sum up to each index.
2. Traverse the array and maintain the cumulative sum.
3. For each element, check if the cumulative sum minus `K` exists in the hashmap. If it does, it means there is a subarray with sum `K`.
4. Update the hashmap with the current cumulative sum.

Program

```
def subarray_sum(nums, k):
    count = 0
    cumulative_sum = 0
    hashmap = {0: 1}
    for num in nums:
        cumulative_sum += num
        if cumulative_sum - k in hashmap:
            count += hashmap[cumulative_sum - k]
        if cumulative_sum in hashmap:
            hashmap[cumulative_sum] += 1
        else:
            hashmap[cumulative_sum] = 1
    return count


nums = [1, 1, 1]
```

```
k = 2
print(subarray_sum(nums, k))  # Output: 2
```

Time Complexity

- **O(n)**, where `n` is the number of elements in the array. Each element is processed once.

Space Complexity

- **O(n)**, for the hashmap storing the cumulative sums.

Example

For the array `[1, 1, 1]` and `k = 2`, there are 2 subarrays with sum 2: `[1, 1]` and `[1, 1]`.

---

35. Find the k-most frequent elements in an array using a priority queue. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a hashmap to count the frequency of each element.
2. Use a min-heap to keep track of the `K` most frequent elements.
3. Traverse the hashmap and maintain a heap of size `K`.
4. Extract the elements from the heap to get the `K` most frequent elements.

Program

```
import heapq
from collections import Counter

def top_k_frequent(nums, k):
    if not nums or k == 0:
        return []

    count = Counter(nums)

    heap = []
    for num, freq in count.items():
```

```
        heapq.heappush(heap, (freq, num))
        if len(heap) > k:
            heapq.heappop(heap)

    result = [num for freq, num in heap]
    return result


nums = [1, 1, 1, 2, 2, 3]
k = 2
print(top_k_frequent(nums, k))  # Output: [1, 2]
```

Time Complexity

- **O(n log k)**, where `n` is the number of elements in the array. Each element is inserted into the heap at most once.

Space Complexity

- **O(n)**, for the hashmap and the heap.

Example

For the array `[1, 1, 1, 2, 2, 3]` and `k = 2`, the most frequent elements are `[1, 2]`.

---

36. Generate all subsets of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use backtracking to explore all possible subsets.
2. For each element, include it in the current subset and recursively generate subsets for the remaining elements.
3. Exclude the element and recursively generate subsets for the remaining elements.

Program

```
def subsets(nums):
    def backtrack(start, path):
        result.append(path)
        for i in range(start, len(nums)):
```

```
        backtrack(i + 1, path + [nums[i]])
    result = []
    backtrack(0, [])
    return result


nums = [1, 2, 3]
print(subsets(nums))  # Output: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

Time Complexity

- **O(2^n)**, where `n` is the number of elements in the array. There are `2^n` possible subsets.

Space Complexity

- **O(2^n)**, for storing all the subsets.

Example

For the array `[1, 2, 3]`, the subsets are `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`.

---

37. Find all unique combinations of numbers that sum to a target. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use backtracking to explore all possible combinations.
2. For each element, include it in the current combination and recursively find combinations for the remaining elements.
3. Exclude the element and recursively find combinations for the remaining elements.

Program

```
def combination_sum(candidates, target):
    def backtrack(start, path, target):
        if target == 0:
            result.append(path)
            return
        if target < 0:
```

```
            return
        for i in range(start, len(candidates)):
            backtrack(i, path + [candidates[i]], target - candidates[i])
    result = []
    candidates.sort()
    backtrack(0, [], target)
    return result


candidates = [2, 3, 6, 7]
target = 7
print(combination_sum(candidates, target))  # Output: [[2, 2, 3], [7]]
```

Time Complexity

- **O(2^n)**, where `n` is the number of elements in the array. There are `2^n` possible combinations.

Space Complexity

- **O(2^n)**, for storing all the combinations.

Example

For the array `[2, 3, 6, 7]` and `target = 7`, the combinations are `[[2, 2, 3], [7]]`.

---

38. Generate all permutations of a given array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use backtracking to explore all possible permutations.
2. For each element, include it in the current permutation and recursively generate permutations for the remaining elements.
3. Exclude the element and recursively generate permutations for the remaining elements.

Program

```
def permute(nums):
    def backtrack(path):
        if len(path) == len(nums):
```

```
            result.append(path)
            return
        for num in nums:
            if num not in path:
                backtrack(path + [num])
    result = []
    backtrack([])
    return result


nums = [1, 2, 3]
print(permute(nums))  # Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Time Complexity

- **O(n!)**, where `n` is the number of elements in the array. There are `n!` possible permutations.

Space Complexity

- **O(n!)**, for storing all the permutations.

Example

For the array `[1, 2, 3]`, the permutations are `[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]`.

---

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

Subsets

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

Permutations

1. A **permutation** is an arrangement of all the elements of a set into a sequence or order.
2. The number of permutations of a set with `n` elements is `n!`.

3. 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]`.

---

40. Solve the problem of finding the element with maximum frequency in an array. Write its algorithm, program. Find its time and space complexities. Explain with suitable example.

Algorithm

1. Use a hashmap to count the frequency of each element.
2. Traverse the hashmap to find the element with the maximum frequency.

Program

```
from collections import Counter

def max_frequency_element(nums):
    if not nums:
        return None

    count = Counter(nums)

    max_freq = max(count.values())
    for num, freq in count.items():
        if freq == max_freq:
            return num
nums = [1, 2, 2, 3, 3, 3, 4]
print(max_frequency_element(nums))  # Output:
```

Time Complexity

- **O(n)**, where `n` is the number of elements in the array. Each element is processed once.

Space Complexity

- **O(n)**, for the hashmap storing the frequencies.

Example

For the array `[1, 2, 2, 3, 3, 3, 4]`, the element with the maximum frequency is `3`.

---

41. Write a program to find the maximum subarray sum using Kadane's algorithm.

Algorithm (Kadane's Algorithm)

1. Initialize two variables: `max_current` and `max_global` to the first element of the array.
2. Traverse the array starting from the second element.
3. For each element, update `max_current` to be the maximum of the current element itself or the sum of `max_current` and the current element.
4. Update `max_global` to be the maximum of `max_global` and `max_current`.
5. Return `max_global`.

Program

```
def max_subarray_sum(nums):
    if not nums:
        return 0
    max_current = max_global = nums[0]
    for num in nums[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current
    return max_global

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(nums))  # Output: 6
```

Time Complexity

- **O(n)**, where `n` is the number of elements in the array. Each element is processed once.

Space Complexity

- **O(1)**, as we are using a constant amount of extra space.

Example

For the array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the maximum subarray sum is 6 (4 + -1 + 2 + 1).

42. Explain the concept of dynamic programming and its use in solving the maximum subarray problem.

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. This is often achieved using a table (usually an array or a matrix) to store intermediate results.

Maximum Subarray Problem

Problem: Given an array of integers, find the contiguous subarray (containing at least one number) which has the largest sum.

Algorithm (Kadane's Algorithm)

1. Initialize two variables: `max_so_far` and `max_ending_here` to the first element of the array.
2. Iterate through the array starting from the second element.
3. For each element, update `max_ending_here` to be the maximum of the current element itself or the sum of `max_ending_here` and the current element.
4. Update `max_so_far` to be the maximum of `max_so_far` and `max_ending_here`.
5. Return `max_so_far`.

Example

For the array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the maximum subarray sum is `6` (subarray `[4, -1, 2, 1]`).

Time Complexity

- **O(n)**

Space Complexity

- **O(1)**

---

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. Use a hash map to count the frequency of each element.
2. Use a min-heap (or max-heap) to keep track of the top `K` frequent elements.
3. Extract the top `K` elements from the heap.

Program

```
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))
```

Time Complexity

- **O(n log k)**

Space Complexity

- **O(n)**

---

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. Use a hash map to store the complement of each element (`target - element`).

2. For each element, check if it exists in the hash map.
3. If it does, return the element and its complement.
4. Otherwise, add the complement to the hash map.

Program

```
def two_sum(nums, target):
    complement_map = {}
    for num in nums:
        complement = target - num
        if complement in complement_map:
            return [complement, num]
        complement_map[num] = True
    return None


nums = [2, 7, 11, 15]
target = 9
print(two_sum(nums, target))  # Output: [2, 7]
```

Time Complexity

- **O(n)**

Space Complexity

- **O(n)**

---

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

Priority Queues are data structures where each element has a priority, and elements are served based on their priority. They are often implemented using heaps.

Applications

1. **Dijkstra's algorithm** for shortest path
2. **Huffman coding** for data compression
3. **Task scheduling** in operating systems

---

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

1. Use dynamic programming to check all substrings.
2. Maintain a 2D table `dp` where `dp[i][j]` is true if the substring `s[i:j+1]` is a palindrome.
3. Update the table based on the conditions:
    - `dp[i][j]` is true if `s[i] == s[j]` and `dp[i+1][j-1]` is true.
4. Track the longest palindromic substring found.

Program

```
def longest_palindromic_substring(s):
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    start, max_length = 0, 1
    for i in range(n):
        dp[i][i] = True
    for cl in range(2, n + 1):
        for i in range(n - cl + 1):
            j = i + cl - 1
            if s[i] == s[j] and (cl == 2 or dp[i + 1][j - 1]):
                dp[i][j] = True
            if cl > max_length:
                start = i
                max_length = cl
    return s[start:start + max_length]


s = "babad"
print(longest_palindromic_substring(s))  # Output: "bab" or "aba"
```

Time Complexity

- **$O(n^2)$**

Space Complexity

- **$O(n^2)$**

---

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

Histogram Problems involve analyzing the distribution of data points. Common problems include finding the largest rectangle in a histogram.

Applications

1. **Stock market analysis**
2. **Image processing**
3. **Data visualization**

---

48. 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

1. Find the first decreasing element from the end of the array.
2. Swap it with the smallest element larger than it.
3. Reverse the suffix.

Program

```
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:])
    return nums
```

```
nums = [1, 2, 3]
print(next_permutation(nums))
```

Time Complexity

- **O(n)**

Space Complexity

- **O(1)**

---

49. 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. Use two pointers to traverse both lists.
2. When a pointer reaches the end of a list, redirect it to the head of the other list.
3. The pointers will meet at the intersection node.

Program

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

def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None
    pa, pb = headA, headB
    while pa is not pb:
        pa = pa.next if pa else headB
        pb = pb.next if pb else headA
    return pa
```

Time Complexity

- **O(n + m)**

Space Complexity

- **O(1)**

---

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

Equilibrium Index is a position in an array where the sum of elements before it is equal to the sum of elements after it.

Applications

1. **Balancing problems**
2. **Pivot selection** in quicksort

Example

For the array `[-7, 1, 5, 2, -4, 3, 0]`, the equilibrium index is `3` (sum of elements before and after index `3` are equal).