Advanced Algorithmic Problem Solving (R1UC601B)
Assignment for MTE

Yuvraj Singh
22SCSE1010656

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

A prefix sum array is a new array where each element at index *i* stores the sum of all elements in the original array up to and including index *i*. For example, if the original array is [a_1, a_2, a_3, ..., an], the prefix sum array would be [a_1, a_1+a_2, $a_1+a_2+a_3$, ..., $a_1+a_2+...+an$].

Applications of prefix sum arrays include:

- **Range Sum Queries:** Efficiently calculating the sum of elements within any given range [L, R] in the original array by subtracting the prefix sum at index *L-1* from the prefix sum at index *R*.
- **Solving Subarray Sum Problems:** Determining if a subarray with a specific sum exists.
- **Image Processing:** Calculating sums of pixel intensities in rectangular regions.
- **Data Analysis:** Quickly computing cumulative sums for various metrics.

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 a suitable example.

Algorithm:

1. **Create a Prefix Sum Array:**
   - Initialize a prefix sum array of the same size as the input array.
   - Set the first element of the prefix sum array equal to the first element of the input array.
   - Iterate through the input array from the second element (index 1) to the end.
   - For each element at index *i* in the input array, calculate the prefix sum at index *i* as the sum of the element at index *i* and the prefix sum at index *i-1*.
2. **Calculate Range Sum:**
   - Given the range [L, R].
   - If *L* is 0, the sum of elements in the range is simply the prefix sum at index *R*.
   - If *L* is greater than 0, the sum of elements in the range is the prefix sum at index *R* minus the prefix sum at index *L-1*.

Program (Python Example):

```
def prefix_sum_array(arr):
    n = len(arr)
    prefix_sum = [0] * n
    if n > 0:
        prefix_sum[0] = arr[0]
        for i in range(1, n):
            prefix_sum[i] = prefix_sum[i-1] + arr[i]
    return prefix_sum

def range_sum(prefix_sum, L, R):
    if L < 0 or R >= len(prefix_sum) or L > R:
        return "Invalid range"
    if L == 0:
        return prefix_sum[R]
    else:
        return prefix_sum[R] - prefix_sum[L-1]

# Example
arr = [1, 2, 3, 4, 5]
prefix_arr = prefix_sum_array(arr)
print(f"Original Array: {arr}")
print(f"Prefix Sum Array: {prefix_arr}")

sum_range_1_3 = range_sum(prefix_arr, 1, 3)
print(f"Sum of elements in range [1, 3]: {sum_range_1_3}")  # Output: 9

sum_range_0_2 = range_sum(prefix_arr, 0, 2)
print(f"Sum of elements in range [0, 2]: {sum_range_0_2}")  # Output: 6
```

Time and Space Complexities:

- **Time Complexity:**
  - Creating the prefix sum array takes O(n) time, where n is the size of the input array.
  - Calculating the range sum using the prefix sum array takes O(1) time.
- **Space Complexity:** O(n) for storing the prefix sum array.

Suitable Example:

Consider the array arr = [2, 5, 1, 8, 3]. The prefix sum array would be prefix_arr = [2, 7, 8, 16, 19].

To find the sum of elements in the range [1, 3] (which are 5, 1, and 8 in the original array):

range_sum(prefix_arr, 1, 3) = prefix_arr[3] - prefix_arr[1-1] = prefix_arr[3] - prefix_arr[0] = 16 - 2 = 14.

This is correct as 5 + 1 + 8 = 14.

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

Algorithm:

An equilibrium index of an array is an index such that the sum of elements at lower indices is equal to the sum of elements at higher indices.

1. **Calculate the Total Sum:** Calculate the sum of all elements in the array.
2. **Iterate and Check:**
   - Initialize a variable left_sum to 0.
   - Iterate through the array from left to right (index i from 0 to n-1).
   - For each index i, calculate the right_sum as total_sum - left_sum - array[i].
   - If left_sum is equal to right_sum, then i is an equilibrium index. Return i.
   - Update left_sum by adding the current element array[i] to it.
3. **No Equilibrium Index:** If the loop completes without finding an equilibrium index, return -1.

Program (Python Example):

```
def find_equilibrium_index(arr):
    n = len(arr)
    total_sum = sum(arr)
    left_sum = 0
    for i in range(n):
        right_sum = total_sum - left_sum - arr[i]
        if left_sum == right_sum:
            return i
        left_sum += arr[i]
    return -1

# Example
```

```
arr1 = [-7, 1, 5, 2, -4, 3, 0]
index1 = find_equilibrium_index(arr1)
print(f"Equilibrium index of {arr1}: {index1}")  # Output: 3

arr2 = [1, 2, 3]
index2 = find_equilibrium_index(arr2)
print(f"Equilibrium index of {arr2}: {index2}")  # Output: -1
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the size of the array.
- **Space Complexity:** O(1), as we are using a constant amount of extra space for variables.

Suitable Example:

Consider the array arr = [-7, 1, 5, 2, -4, 3, 0].

- Total sum = -7 + 1 + 5 + 2 - 4 + 3 + 0 = 0.

Let's trace the iteration:

- i = 0: left_sum = 0, right_sum = 0 - 0 - (-7) = 7. left_sum != right_sum. left_sum becomes -7.
- i = 1: left_sum = -7, right_sum = 0 - (-7) - 1 = 6. left_sum != right_sum. left_sum becomes -6.
- i = 2: left_sum = -6, right_sum = 0 - (-6) - 5 = 1. left_sum != right_sum. left_sum becomes -1.
- i = 3: left_sum = -1, right_sum = 0 - (-1) - 2 = -1. left_sum == right_sum. Equilibrium index is 3.

The element at index 3 is 2. The sum of elements before it (-7 + 1 + 5 = -1) is equal to the sum of elements after it (-4 + 3 + 0 = -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 a suitable example.

Algorithm:

1. **Calculate the Total Sum:** Calculate the sum of all elements in the array.
2. **Iterate and Check:**
    - Initialize a variable left_sum to 0.
    - Iterate through the array from left to right (index i from 0 to n-1).
    - Update left_sum by adding the current element array[i].

- Calculate the right_sum as total_sum - left_sum.
- If left_sum is equal to right_sum, it means the array can be split at index i+1 (after the current element) into two parts with equal sums. Return True.

3. **No Split Found:** If the loop completes without finding such a split, return False.

Program (Python Example):

```
def can_split_equal_sum(arr):
    n = len(arr)
    total_sum = sum(arr)
    left_sum = 0
    for i in range(n):
        left_sum += arr[i]
        right_sum = total_sum - left_sum
        if left_sum == right_sum:
            return True
    return False

# Example
arr1 = [1, 2, 3, 3]
can_split1 = can_split_equal_sum(arr1)
print(f"Can {arr1} be split into two parts with equal sum? {can_split1}")  # Output: True

arr2 = [1, 2, 3, 4, 5]
can_split2 = can_split_equal_sum(arr2)
print(f"Can {arr2} be split into two parts with equal sum? {can_split2}")  # Output: False
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the size of the array.
- **Space Complexity:** O(1), as we are using a constant amount of extra space.

Suitable Example:

For the array arr = [1, 2, 3, 3], the total sum is 9. The array can be split into [1, 2, 3] and [3], both having a sum of 6.

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 a

suitable example.

Algorithm:

1. **Calculate the sum of the first K elements.** Store this as max_sum and window_sum.
2. **Use a sliding window:** Iterate from the Kth element to the end of the array.
    - Subtract the first element of the current window (arr[i-K]) from window_sum.
    - Add the next element (arr[i]) to window_sum.
    - Update max_sum if window_sum is greater.
3. Return max_sum.

Program (Python Example):

```
def max_subarray_sum_k(arr, k):
    n = len(arr)
    if k > n:
        return "Invalid k"

    window_sum = sum(arr[:k])  # Sum of first k elements
    max_sum = window_sum

    for i in range(k, n):
        window_sum = window_sum - arr[i - k] + arr[i]  # Slide the window
        max_sum = max(max_sum, window_sum)

    return max_sum

# Example
arr = [1, 4, 2, 10, 23, 3, 1, 0, 20]
k = 4
max_sum = max_subarray_sum_k(arr, k)
print(f"Maximum sum of subarray of size {k}: {max_sum}")  # Output: 39
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array arr = [1, 4, 2, 10, 23, 3, 1, 0, 20] and k = 4, the maximum sum subarray

is [10, 23, 3, 1], with a sum of 37. The algorithm correctly identifies this.

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

Algorithm:

1. Initialize a seen dictionary to store the last seen index of each character.
2. Initialize start and max_length to 0.
3. Iterate through the string:
    - If the current character is in seen and its last seen index is greater than or equal to start, update start to seen[character] + 1.
    - Update seen[character] with the current index.
    - Update max_length to the maximum of max_length and the current substring length (index - start + 1).
4. Return max_length.

Program (Python Example):

```
def longest_unique_substring_length(s):
    seen = {}
    start = 0
    max_length = 0

    for index, character in enumerate(s):
        if character in seen and seen[character] >= start:
            start = seen[character] + 1
        seen[character] = index
        max_length = max(max_length, index - start + 1)
    return max_length

# Example
s = "abcabcbb"
length = longest_unique_substring_length(s)
print(f"Length of longest unique substring: {length}")  # Output: 3
```

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.

Suitable Example:

For the string "abcabcbb", the longest substring without repeating characters is "abc", with a length of 3.

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

The sliding window technique is a method for efficiently processing contiguous portions of a sequence (like an array or string). Instead of examining every possible sub-sequence, it maintains a "window" that moves through the data, processing only the elements within the window at any given time. This avoids redundant calculations.

In string problems, the sliding window technique is useful for:

- Finding substrings that satisfy certain conditions (e.g., containing a specific set of characters).
- Calculating statistics on substrings (e.g., maximum sum of a substring of length K).
- Optimizing string matching and searching algorithms.

8. Find the longest palindromic substring in a given string. Write its algorithm, program. Find its time and space complexities. Explain with a suitable example.

Algorithm:

1. Iterate through each character in the string.
2. For each character, consider it as the center of a potential palindrome and expand outwards, checking for both odd-length and even-length palindromes.
3. Keep track of the longest palindrome found so far.

Program (Python Example):

```
def longest_palindrome(s):
    n = len(s)
    if n < 2:
        return s

    longest = ""

    for i in range(n):
        # Odd length palindromes
        l, r = i, i
```

```
        while l >= 0 and r < n and s[l] == s[r]:
            if (r - l + 1) > len(longest):
                longest = s[l : r + 1]
            l -= 1
            r += 1

        # Even length palindromes
        l, r = i, i + 1
        while l >= 0 and r < n and s[l] == s[r]:
            if (r - l + 1) > len(longest):
                longest = s[l : r + 1]
            l -= 1
            r += 1
    return longest

# Example
s = "babad"
result = longest_palindrome(s)
print(f"Longest palindrome: {result}")  # Output: "bab" or "aba"
```

Time and Space Complexities:

- **Time Complexity:** O(n^2), where n is the length of the string.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the string "babad", the longest palindromic substring is "bab" (or "aba").

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

Algorithm:

1. If the list of strings is empty, return "".
2. Take the first string as the initial prefix.
3. Iterate through the remaining strings.
    - For each string, compare it character by character with the current prefix.
    - Shorten the prefix if it doesn't match the current string.
4. Return the final prefix.

Program (Python Example):

```
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:
            break
    return prefix

# Example
strs = ["flower", "flow", "flight"]
result = longest_common_prefix(strs)
print(f"Longest common prefix: {result}")  # Output: "fl"
```

Time and Space Complexities:

- **Time Complexity:** O(S), where S is the total number of characters in all strings.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the list of strings ["flower", "flow", "flight"], the longest common prefix is "fl".

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

Algorithm:

1. If the string is empty or has only one character, return the string itself as the only permutation.
2. For a string of length n:
    - Fix the first character and recursively generate all permutations of the remaining (n-1) characters.
    - For each of the permutations obtained from the recursive call, insert the fixed first character at all possible positions.

Program (Python Example):

```
def get_permutations(s):
    if len(s) <= 1:
        return [s]

    permutations = []
    first_char = s[0]
    rest_chars = s[1:]
    perms_of_rest = get_permutations(rest_chars)

    for perm in perms_of_rest:
        for i in range(len(perm) + 1):
            new_perm = perm[:i] + first_char + perm[i:]
            permutations.append(new_perm)
    return permutations

# Example
s = "abc"
result = get_permutations(s)
print(f"Permutations of {s}: {result}")  # Output: ['abc', 'acb', 'bac', 'bca', 'cab',
'cba']
```

Time and Space Complexities:

- **Time Complexity:** O(n! * n), where n is the length of the string. (n! for the number of permutations, and n for the string concatenation in each permutation)
- **Space Complexity:** O(n!), to store the generated permutations.

Suitable Example:

For the string "abc", the permutations are "abc", "acb", "bac", "bca", "cab", and "cba".

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 a suitable example.

Algorithm:

1. Use two pointers, left starting at the beginning of the array, and right starting at the end.
2. While left < right:

    - Calculate the sum of the elements 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 -1, -1.

Program (Python Example):

```
def find_two_sum(arr, target):
    left = 0
    right = 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 -1, -1

# Example
arr = [2, 7, 11, 15]
target = 9
result = find_two_sum(arr, target)
print(f"Indices: {result}")  # Output: (0, 1)
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array [2, 7, 11, 15] and target 9, the numbers 2 and 7 at indices 0 and 1 add up to 9.

12. Rearrange numbers into the lexicographically next greater permutation. Write its algorithm, program. Find its time and space complexities. Explain with a suitable example.

Algorithm:

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

Program (Python Example):

```
def next_permutation(arr):
    n = len(arr)
    # Find the largest i such that arr[i] < arr[i+1]
    i = n - 2
    while i >= 0 and arr[i] >= arr[i + 1]:
        i -= 1

    if i == -1:  # Array is in descending order
        arr.reverse()
        return

    # Find the largest j > i such that arr[j] > arr[i]
    j = n - 1
    while arr[j] <= arr[i]:
        j -= 1

    # Swap arr[i] and arr[j]
    arr[i], arr[j] = arr[j], arr[i]

    # Reverse the subarray from i+1 to the end
    left = i + 1
    right = n - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1

# Example
arr = [1, 2, 3]
next_permutation(arr)
print(f"Next permutation: {arr}")  # Output: [1, 3, 2]
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array [1, 2, 3], the next greater permutation is [1, 3, 2].

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

Algorithm:

1. Create a dummy node as the head of the merged list.
2. Use pointers to traverse both lists.
3. Compare the values at the current nodes of the two lists.
4. Append the smaller value node to the merged list and move the pointer of the list from which the smaller element was taken.
5. If one list is exhausted, append the remaining nodes of the other list to the merged list.
6. Return the next of the dummy node.

Program (Python Example):

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

def merge_two_lists(l1, l2):
    dummy = ListNode()
    tail = dummy

    whilel1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
```

```
    if l1:
        tail.next = l1
    elif l2:
        tail.next = l2

    return dummy.next

# Example
# Create sorted linked lists: 1->2->4, 1->3->4
l1 = ListNode(1, ListNode(2, ListNode(4)))
l2 = ListNode(1, ListNode(3, ListNode(4)))
merged_list = merge_two_lists(l1, l2)

# Print the merged list (1->1->2->3->4->4)
while merged_list:
    print(merged_list.val, end="->")
    merged_list = merged_list.next
print("None")
```

Time and Space Complexities:

- **Time Complexity:** O(m + n), where m and n are the lengths of the two lists.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

Merging the sorted linked lists 1->2->4 and 1->3->4 results in the sorted linked list 1->1->2->3->4->4.

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

Algorithm:

1. Ensure nums1 is the shorter array. If not, swap them.
2. Initialize low = 0 and high = len(nums1).
3. Perform binary search on the shorter array:
    - Calculate partitionX and partitionY.
    - Find maxLeftX, minRightX, maxLeftY, and minRightY.
    - If the partitions are correct (conditions are met), calculate the median.
    - If maxLeftX > minRightY, move high to partitionX - 1.

- Else, move low to partitionX + 1.

Program (Python Example):

```
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
    return -1

# Example
nums1 = [1, 3]
nums2 = [2]
median = find_median_sorted_arrays(nums1, nums2)
print(f"Median: {median}")  # Output: 2.0
```

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), as we use only a constant amount of extra space.

Suitable Example:

For nums1 = [1, 3] and nums2 = [2], the median of the merged sorted array [1, 2, 3] is 2.0.

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

Algorithm:

1. Use binary search on the range of possible values (from the smallest to the largest element in the matrix).
2. For each mid value in the binary search:
   - Count the number of elements in the matrix that are less than or equal to mid.
3. Adjust the search range based on the count:
   - If the count is less than k, search in the higher range.
   - If the count is greater than or equal to k, search in the lower range.

Program (Python Example):

```
def kth_smallest(matrix, k):
    n = len(matrix)
    low = matrix[0][0]
    high = matrix[n - 1][n - 1]

    while low < high:
        mid = low + (high - low) // 2
        count = 0
        for r in range(n):
            c = n - 1
            while c >= 0 and matrix[r][c] > mid:
                c -= 1
            count += (c + 1)
        if count < k:
            low = mid + 1
        else:
            high = mid
    return low
```

```
# Example
matrix = [
    [1,  5,  9],
    [10, 11, 13],
    [12, 13, 15]
]
k = 8
result = kth_smallest(matrix, k)
print(f"8th smallest element: {result}")  # Output: 13
```

Time and Space Complexities:

- **Time Complexity:** O(n * log(max - min)), where n is the number of rows/columns in the matrix, and max and min are the maximum and minimum values in the matrix.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the matrix

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

the 8th smallest element is 13.

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 a suitable example.

Algorithm (Boyer-Moore Voting Algorithm):

1. Initialize a candidate and a count to 0.
2. 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.
3. The candidate is the majority element (if it exists).

Program (Python Example):

```
def find_majority_element(arr):
```

```
    candidate = 0
    count = 0

    for num in arr:
        if count == 0:
            candidate = num
            count = 1
        elif num == candidate:
            count += 1
        else:
            count -= 1
    return candidate

# Example
arr = [3, 2, 3]
majority_element = find_majority_element(arr)
print(f"Majority element: {majority_element}")  # Output: 3
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array [3, 2, 3], the majority element is 3.

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 a suitable example.

Algorithm:

1. **Initialization:**
    - Initialize two pointers, left at 0 and right at the end of the histogram (array).
    - Initialize left_max and right_max to 0.
    - Initialize water to 0.
2. **Iteration:**
    - While left < right:
        - If histogram[left] < histogram[right]:

        - If histogram[left] > left_max, update left_max.
        - Else, add left_max - histogram[left] to water.
        - Increment left.
    - Else:
        - If histogram[right] > right_max, update right_max.
        - Else, add right_max - histogram[right] to water.
        - Decrement right.
3. **Return:** Return water.

Program (Python Example):

```
def trap(histogram):
    left = 0
    right = len(histogram) - 1
    left_max = 0
    right_max = 0
    water = 0

    while left < right:
        if histogram[left] < histogram[right]:
            if histogram[left] > left_max:
                left_max = histogram[left]
            else:
                water += left_max - histogram[left]
            left += 1
        else:
            if histogram[right] > right_max:
                right_max = histogram[right]
            else:
                water += right_max - histogram[right]
            right -= 1
    return water

# Example
histogram = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
trapped_water = trap(histogram)
print(f"Trapped water: {trapped_water}")  # Output: 6
```

Time and Space Complexities:

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

- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the histogram [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1], the trapped water is 6 units.

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

Algorithm:

1. **Initialization:**
   - Initialize max_xor to 0.
   - Create a set seen to store prefixes of numbers.
2. **Iteration:**
   - Iterate through each number in the array.
   - For each number, iterate from the most significant bit (31 for 32-bit integers) to the least significant bit.
     - Calculate the current prefix.
     - Calculate the opposite prefix needed to achieve a 1 in the XOR result for the current bit.
     - If the opposite prefix is in seen, update max_xor.
     - Add the current prefix to seen.
   - Clear the seen set after processing each number.
3. **Return:** Return max_xor.

Program (Python Example):

```
def find_max_xor(nums):
    max_xor = 0
    for i in range(len(nums)):
        seen = set()
        for j in range(i, len(nums)):
            num1 = nums[i]
            num2 = nums[j]
            max_xor = max(max_xor, num1 ^ num2)
    return max_xor

def find_max_xor_optimized(nums):
    max_xor = 0
    for num1 in nums:
        for num2 in nums:
```

```
            max_xor = max(max_xor, num1 ^ num2)
    return max_xor

# Example
nums = [3, 10, 5, 25, 2, 8]
max_xor = find_max_xor_optimized(nums)
print(f"Maximum XOR: {max_xor}")  # Output: 28 (5 ^ 25)
```

Time and Space Complexities:

- **Time Complexity:** O(n^2), where n is the number of elements in the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array [3, 10, 5, 25, 2, 8], the maximum XOR is 28 (5 ^ 25).

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

Algorithm:

1. **Initialization:**
    - Initialize max_product, min_product, and result to the first element of the array.
2. **Iteration:**
    - Iterate through the array starting from the second element.
    - For each element:
        - If the current element is negative, swap max_product and min_product.
        - Update max_product to the maximum of the current element and max_product * current element.
        - Update min_product to the minimum of the current element and min_product * current element.
        - Update result to the maximum of result and max_product.
3. **Return:** Return result.

Program (Python Example):

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

```
    max_product = nums[0]
    min_product = nums[0]
    result = nums[0]

    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

# Example
nums = [2, 3, -2, 4]
max_product = max_product_subarray(nums)
print(f"Maximum product: {max_product}")  # Output: 6
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For the array [2, 3, -2, 4], the maximum product subarray is [2, 3], with a product of 6. For the array [-2, 0, -1], the maximum product is 0.

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 a suitable example.

Algorithm:

1. **Base Cases:**
    - If n is 0, return 1 (only 0 has unique digits).
    - If n is 1, return 10 (0 to 9).
2. **Iteration:**
    - Initialize count to 10 (for n = 1).
    - Initialize unique_digits to 9 (choices for the first digit, cannot be 0).
    - Iterate from i = 2 to n:
        - unique_digits = unique_digits * (10 - i + 1) // Choices decrease for subsequent digits.

        - count += unique_digits.
3. **Return:** Return count.

Program (Python Example):

```
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, n + 1):
        unique_digits = unique_digits * (10 - i + 1)
        count += unique_digits
    return count

# Example
n = 2
result = count_numbers_with_unique_digits(n)
print(f"Count for n = {n}: {result}")  # Output: 91
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the given number of digits.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

For n = 2, the numbers with unique digits are 0-9 (10 numbers), 10-19, 20-29,...90-98 (9 * 9 = 81 numbers). Total is 10 + 81 = 91.

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 a suitable example.

Algorithm:

1. **Initialization:** Create an array dp of size n + 1. dp[0] = 0.
2. **Iteration:** Iterate from i = 1 to n:
    - dp[i] = dp[i >> 1] + (i & 1). (Right shift by 1 and add the last bit).
3. **Calculate Total Count:** Sum all the values in the dp array.

4. **Return:** Return the total count.

Program (Python Example):

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

# Example
n = 5
result = count_ones(n)
print(f"Total 1s from 0 to {n}: {result}")  # Output: 7
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the given number.
- **Space Complexity:** O(n), to store the dp array.

Suitable Example:

For n = 5:

- 0: 000 - 0 ones
- 1: 001 - 1 one
- 2: 010 - 1 one
- 3: 011 - 2 ones
- 4: 100 - 1 one
- 5: 101 - 2 ones

Total ones: 0 + 1 + 1 + 2 + 1 + 2 = 7.

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 a suitable example.

Algorithm:

A number n is a power of two if it has only one bit set to 1 in its binary representation.

1. If n is less than or equal to 0, it's not a power of two. Return False.
2. Return the result of (n & (n - 1)) == 0. This operation removes the least significant 1 bit. If n is a power of 2, the result will be 0.

Program (Python Example):

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

# Example
numbers = [1, 2, 3, 4, 5, 8, 16, 24]
for num in numbers:
    print(f"{num} is a power of two: {is_power_of_two(num)}")
```

Time and Space Complexities:

- **Time Complexity:** O(1), constant time.
- **Space Complexity:** O(1), constant space.

Suitable Example:

- 8 (1000 in binary) is a power of two: 8 & 7 (1000 & 0111) = 0.
- 5 (0101 in binary) is not a power of two: 5 & 4 (0101 & 0100) = 4.

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 a suitable example.

This is a duplicate of question 18. Please refer to the answer provided for question 18.

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 individual bits of binary representations of numbers. Instead of working with numbers as abstract quantities, you manipulate the 0s and 1s that compose them.

Advantages in Algorithm Design:

- **Efficiency:** Bitwise operations are often much faster than traditional arithmetic operations. They can be executed directly by the CPU.
- **Space Efficiency:** Bit manipulation can allow you to store and process information in a very compact form. For example, you can use a single bit to represent a boolean flag.

- **Conciseness:** Some algorithms can be expressed more concisely and elegantly using bit manipulation.
- **Solving Specific Problems:** Bit manipulation is essential for solving certain types of problems, such as those involving:
    - Low-level hardware manipulation.
    - Error detection and correction.
    - Cryptography.
    - Compression.
    - Optimization problems.

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 a suitable example.

Algorithm:

1. **Initialization:**
    - Initialize an empty stack.
    - Initialize an output array result of the same size as the input array, filled with -1.
2. **Iteration:**
    - Iterate through the array.
    - For each element arr[i]:
        - While the stack is not empty and arr[i] is greater than the element at the top of the stack:
            - Pop the index from the stack.
            - Set result[popped_index] to arr[i].
        - Push the current index i onto the stack.
3. **Return:** Return the result array.

Program (Python Example):

```
def next_greater_element(arr):
    stack = []
    result = [-1] * len(arr)

    for i in range(len(arr)):
        while stack and arr[i] > arr[stack[-1]]:
            popped_index = stack.pop()
            result[popped_index] = arr[i]
        stack.append(i)
    return result
```

```
# Example
arr = [1, 3, 2, 4]
result = next_greater_element(arr)
print(f"Next greater elements: {result}")  # Output: [3, 4, 4, -1]
```

Time and Space Complexities:

- **Time Complexity:** O(n), where n is the length of the array. Each element is pushed onto and popped from the stack at most once.
- **Space Complexity:** O(n), in the worst case, the stack might contain all the indices of the array.

Suitable Example:

For the array [1, 3, 2, 4]:

- 1's next greater element is 3.
- 3's next greater element is 4.
- 2's next greater element is 4.
- 4's next greater element is -1 (no greater element).

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 a suitable example.

Algorithm:

1. Use two pointers, slow and fast.
2. Move fast n nodes ahead.
3. Move both slow and fast until fast reaches the end of the list.
4. At this point, slow is pointing to the node before the one to be deleted.
5. Adjust the pointers to remove the node.

Program (Python Example):

```
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, head)  # Use a dummy node to handle the case where the head is removed.
    slow = dummy
```

```
    fast = head

    # Move fast n nodes ahead
    for _ in range(n):
        fast = fast.next

    # Move slow and fast until fast reaches the end
    while fast:
        slow = slow.next
        fast = fast.next

    # Remove the nth node from the end
    slow.next = slow.next.next

    return dummy.next

# Example
# Create linked list: 1->2->3->4->5
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
n = 2
new_head = remove_nth_from_end(head, n)

# Print the modified list: 1->2->3->5
while new_head:
    print(new_head.val, end="->")
    new_head = new_head.next
print("None")
```

Time and Space Complexities:

- **Time Complexity:** O(L), where L is the length of the linked list.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

Given the linked list 1->2->3->4->5 and n = 2, removing the 2nd node from the end (which is 4) results in the list 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 a suitable example.

Algorithm:

1. Find the lengths of the two linked lists, lenA and lenB.
2. Move the pointer of the longer list forward by the difference abs(lenA - lenB).
3. Move both pointers forward until they meet.
4. Return the meeting node (or None if they don't intersect).

Program (Python Example):

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

def get_intersection_node(headA, headB):
    lenA = 0
    lenB = 0
    currA = headA
    currB = headB

    # Find the lengths of the lists
    while currA:
        lenA += 1
        currA = currA.next
    while currB:
        lenB += 1
        currB = currB.next

    currA = headA
    currB = headB

    # Move the pointer of the longer list forward
    if lenA > lenB:
        for _ in range(lenA - lenB):
            currA = currA.next
    else:
        for _ in range(lenB - lenA):
            currB = currB.next

    # Move both pointers until they meet
    while currA and currB and currA != currB:
        currA = currA.next
```

```
        currB = currB.next

    return currA  # Or currB, they are the same at the intersection

# Example
# Create intersecting linked lists:
#    4->1->8->4->5
#  2->3->8->4->5
common = ListNode(8, ListNode(4, ListNode(5)))
headA = ListNode(4, ListNode(1, common))
headB = ListNode(2, ListNode(3, common))
intersection_node = get_intersection_node(headA, headB)
if intersection_node:
    print(f"Intersection node value: {intersection_node.val}")  # Output: 8
else:
    print("No intersection")
```

Time and Space Complexities:

- **Time Complexity:** O(m + n), where m and n are the lengths of the two lists.
- **Space Complexity:** O(1), as we use only a constant amount of extra space.

Suitable Example:

If two linked lists intersect at node with value 8, the function will return the node with value 8.

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

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. Maintain two pointers, top1 and top2, to track the top elements of the stacks.
4. For the first stack, push elements from left to right.
5. For the second stack, push elements from right to left.

Program (Python Example):

```
class TwoStacks:
    def __init__(self, n):
        self.array = [None] * n
```

```
        self.size = n
        self.top1 = -1
        self.top2 = n  # Start from the right end

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

    def push2(self, x):
        if self.top1 < self.top2 - 1:
            self.top2 -= 1
            self.array[self.top2] = x
        else:
            print("Stack 2 Overflow")

    def pop1(self):
        if self.top1 >= 0:
            x = self.array[self.top1]
            self.top1 -= 1
            return x
        else:
            print("Stack 1 Underflow")
            return None

    def pop2(self):
        if self.top2 < self.size:
            x = self.array[self.top2]
            self.top2 += 1
            return x
        else:
            print("Stack 2 Underflow")
            return None

    def peek1(self):
        if self.top1 >= 0:
            return self.array[self.top1]
        else:
```

```
            print("Stack 1 is empty")
            return None

    def peek2(self):
        if self.top2 < self.size:
            return self.array[self.top2]
        else:
            print("Stack 2 is empty")
            return None

# Example
stacks = TwoStacks(6)
stacks.push1(1)
stacks.push1(2)
stacks.push2(6)
stacks.push2(5)

print(f"Stack 1 top: {stacks.peek1()}")  # Output: 2
print(f"Stack 2 top: {stacks.peek2()}")  # Output: 5

print(f"Popped from Stack 1: {stacks.pop1()}")  # Output: 2
print(f"Popped from Stack 2: {stacks.pop2()}")  # Output: 5
```

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 a suitable example.

Algorithm:

1. Initialize num as the integer to be checked.
2. If num is negative, it's not a palindrome. Return False.
3. Initialize original_num = num and reversed_num = 0.
4. While num is greater than 0:
    - Extract the last digit: digit = num % 10.
    - Build the reversed number: reversed_num = reversed_num * 10 + digit.
    - Remove the last digit from num: num = num // 10.
5. If original_num is equal to reversed_num, return True.
6. Otherwise, return False.

Program (Python):

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

# Example
number = 121
if is_palindrome(number):
    print(f"{number} is a palindrome.")
else:
    print(f"{number} is not a palindrome.")

number = 123
if is_palindrome(number):
    print(f"{number} is a palindrome.")
else:
    print(f"{number} is not a palindrome.")
```

Time Complexity: O(log10 n), where n is the input integer.

Space Complexity: O(1), as we are using a constant amount of extra space for variables.

Example:

Let's take the number 121.

1. num = 121, original_num = 121, reversed_num = 0.
2. Iteration 1: digit = 121 % 10 = 1, reversed_num = 0 * 10 + 1 = 1, num = 121 // 10 = 12.
3. Iteration 2: digit = 12 % 10 = 2, reversed_num = 1 * 10 + 2 = 12, num = 12 // 10 = 1.
4. Iteration 3: digit = 1 % 10 = 1, reversed_num = 12 * 10 + 1 = 121, num = 1 // 10 =

0.
5. The loop terminates. original_num (121) is equal to reversed_num (121). So, 121 is a palindrome.

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 not stored at contiguous memory locations. Instead, each element (called a node) contains two parts:

- **Data:** The actual value being stored.
- **Pointer (or Reference):** A link to the next node in the sequence.

The first node in a linked list is called the **head**, and the last node's pointer typically points to None (or NULL).

Types of Linked Lists:

- **Singly Linked List:** Each node points only to the next node.
- **Doubly Linked List:** Each node has two pointers: one to the next node and one to the previous node. This allows for traversal in both directions.
- **Circular Linked List:** The last node's pointer points back to the head node, forming a cycle.

Applications in Algorithm Design:

Linked lists are versatile and have several applications in algorithm design due to their dynamic nature and efficient insertion/deletion at arbitrary positions:

1. **Implementation of Abstract Data Types:**
   - **Stacks:** Linked lists can be used to implement stacks (LIFO - Last-In, First-Out) where insertion (push) and deletion (pop) occur at the head.
   - **Queues:** Linked lists can implement queues (FIFO - First-In, First-Out) where insertion (enqueue) happens at the tail and deletion (dequeue) happens at the head.
   - **Hash Tables:** Linked lists are often used for collision resolution in hash tables (separate chaining). When two keys hash to the same index, a linked list can store the colliding key-value pairs.
2. **Dynamic Memory Allocation:** Linked lists can efficiently manage dynamically allocated memory. Each block of free memory can be represented as a node in a linked list, allowing for easy allocation and deallocation.
3. **Polynomial Representation:** Polynomials can be represented using linked lists where each node stores a coefficient and an exponent. This allows for

efficient addition, subtraction, and multiplication of polynomials.

4. **Undo/Redo Functionality:** In text editors or other applications, a doubly linked list can store the history of operations, allowing for easy undoing and redoing of actions by moving backward and forward in the list.
5. **Graph Representation (Adjacency List):** Linked lists are commonly used in the adjacency list representation of graphs. For each vertex, a linked list stores its neighboring vertices.
6. **Music Playlists:** Linked lists are suitable for implementing music playlists where you can easily add, remove, or rearrange songs in the sequence.
7. **Operating System Tasks:** Linked lists are used in operating systems for various tasks like managing processes, memory, and files.

Advantages of Linked Lists:

- **Dynamic Size:** Linked lists can grow or shrink in size during runtime.
- **Efficient Insertion and Deletion:** Insertion and deletion of nodes at any position can be done in O(1) time if the position is known (after traversing to that position, which takes O(n) in the worst case for a singly linked list).
- **No Memory Wastage (Potentially):** Memory is allocated only when needed.

Disadvantages of Linked Lists:

- **Random Access Not Efficient:** Accessing an element at a specific index takes O(n) time as you need to traverse from the head.
- **Extra Memory Overhead:** Each node requires extra memory to store the pointer(s).

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 a suitable example.

Algorithm:

1. Initialize an empty deque D.
2. Iterate through the input array arr of size n.
3. For each element arr[i]:
    - Remove elements from the back of the deque that are smaller than or equal to the current element arr[i]. This ensures that the deque maintains elements in decreasing order from front to back.
    - Append the index i of the current element to the back of the deque.
    - If the index at the front of the deque is i - K, it means the element at the front is outside the current window of size K. Remove it from the front of the deque.
    - If i >= K - 1, it means a window of size K has been formed. The maximum

element in this window is the element at the index stored at the front of the deque (arr[D[0]]). Store this maximum.

4. Return the list of maximums for each window.

Program (Python):

```
from collections import deque

def find_max_sliding_window(arr, k):
    n = len(arr)
    if n == 0 or k <= 0 or k > n:
        return []
    D = deque()
    max_values = []
    for i in range(n):
        # Remove elements from back of deque that are smaller than current element
        while D and arr[D[-1]] <= arr[i]:
            D.pop()
        # Add current element's index to the back of deque
        D.append(i)
        # Remove elements from front of deque that are out of the current window
        if D[0] == i - k:
            D.popleft()
        # If window size is reached, the front of deque has the index of the maximum element
        if i >= k - 1:
            max_values.append(arr[D[0]])
    return max_values

# Example
arr = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
result = find_max_sliding_window(arr, k)
print(f"Maximums in sliding windows of size {k}: {result}")
# Expected output: [3, 3, 5, 5, 6, 7]
```

Time Complexity: O(n), where n is the size of the input array.

Space Complexity: O(k), where k is the size of the sliding window.

Example:

Let's consider the array arr = [1, 3, -1, -3, 5, 3, 6, 7] and window size k = 3.

Index (i)	Element (arr[i])	Deque (D)	Maximum in Window
0	1	[0]	
1	3	[1]	
2	-1	[1, 2]	3 (arr[1])
3	-3	[2, 3]	3 (arr[1])
4	5	[4]	5 (arr[4])
5	3	[4, 5]	5 (arr[4])
6	6	[6]	6 (arr[6])
7	7	[7]	7 (arr[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 a suitable example.

Algorithm:

1. Initialize an empty stack stack.
2. Initialize max_area = 0.
3. Iterate through the histogram heights array of size n, including a sentinel value of 0 at the end to handle remaining bars in the stack. Let the current index be i.
4. While the stack is not empty and the height of the bar at the top of the stack (heights[stack[-1]]) is greater than the height of the current bar (heights[i]):
    - Pop the top index top from the stack.
    - Calculate the width of the rectangle. If the stack is empty, the width is i. Otherwise, the width is i - stack[-1] - 1.
    - Calculate the area of the rectangle: area = heights[top] * width.
    - Update max_area = max(max_area, area).
5. Push the current index i onto the stack.
6. After the loop finishes, max_area will contain the largest rectangular area.

Program (Python):

```
def largest_rectangle_area(heights):
    stack = []
    max_area = 0
    n = len(heights)
    for i in range(n + 1):
        # Add a sentinel value at the end to process remaining bars
        current_height = heights[i] if i < n else 0
        while stack and heights[stack[-1]] > current_height:
            height = heights[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)
    return max_area

# Example
histogram = [2, 1, 5, 6, 2, 3]
max_area = largest_rectangle_area(histogram)
print(f"The largest rectangular area in the histogram is: {max_area}")
```

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

Space Complexity: O(n) in the worst case, as the stack might store all the indices of the histogram bars if they are in increasing order.

Example:

Consider the histogram heights = [2, 1, 5, 6, 2, 3].

Index (i)	Height (current_height)	Stack	Calculation	max_area
0	2	[0]		0
1	1	[0]	heights[0] > 1: height = 2, width = 1 - (-1) - 1 = 1, area = 2 * 1 = 2, max_area = 2	2
		[1]		

2	5	[1, 2]		2
3	6	[1, 2, 3]		2
4	2	[1, 2, 3]	heights[3] > 2: height = 6, width = 4 - 2 - 1 = 1, area = 6 * 1 = 6, max_area = 6	6
		[1, 2]	heights[2] > 2: height = 5, width = 4 - 1 - 1 = 2, area = 5 * 2 = 10, max_area = 10	10
		[1, 4]		10
5	3	[1, 4, 5]		10
6 (sentinel)	0	[1, 4, 5]	heights[5] > 0: height = 3, width = 6 - 4 - 1 = 1, area = 3 * 1 = 3, max_area = 10	10
		[1, 4]	heights[4] > 0: height = 2, width = 6 - 1 - 1 = 4, area = 2 * 4 = 8, max_area = 10	10
		[1]	heights[1] > 0: height = 1, width = 6 - (-1) - 1 = 6, area = 1 * 6 = 6, max_area = 10	10
		[ ]		10

The largest rectangle has an area of 10, formed by the bars of height 5 and 6.

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

Concept of Sliding Window Technique:

The sliding window technique is a powerful algorithmic approach used to solve problems involving contiguous subarrays or substrings of a given size. Instead of repeatedly iterating through all possible subarrays/substrings, it maintains a "window" that slides through the data structure (usually an array or string).

The core idea is to:

1. **Define a window:** Determine the initial size of the window (fixed or variable).
2. **Slide the window:** Move the window one element (or a certain number of elements) at a time across the data structure.
3. **Maintain information:** Calculate and update the necessary information (sum, maximum, minimum, count, etc.) within the window as it slides.

Applications in Algorithm Problems:

The sliding window technique is used to solve a variety of problems, including:

- Finding the maximum or minimum sum of a subarray of a fixed size.
- Finding the longest subarray with a sum equal to a given target.
- Finding the longest substring without repeating characters.
- Counting the occurrences of a specific pattern in a string.
- Solving problems involving anagrams.
- Various optimization problems on arrays or strings.

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 a suitable example.

Algorithm:

1. Initialize a hash map (or dictionary) prefix_sum_map to store the cumulative sum up to each index and its frequency.
2. Initialize current_sum = 0 and count = 0.
3. Add an initial entry to prefix_sum_map: {0: 1} (representing a prefix sum of 0 before the start of the array).
4. Iterate through the input array arr of size n.
5. For each element num at index i:
    - Update current_sum = current_sum + num.
    - Check if current_sum - K exists as a key in prefix_sum_map. If it does, it means there was a previous prefix sum such that the subarray between

that prefix sum's index + 1 and the current index has a sum of K. Add the frequency of current_sum - K in prefix_sum_map to count.
    - Update the frequency of current_sum in prefix_sum_map. If current_sum is already a key, increment its value; otherwise, add it with a value of 1.
6. Return count.

Program (Python):

```
def subarray_sum_equals_k(arr, k):
    prefix_sum_map = {0: 1}
    current_sum = 0
    count = 0
    for num in arr:
        current_sum += num
        if current_sum - k in prefix_sum_map:
            count += prefix_sum_map[current_sum - k]
        prefix_sum_map[current_sum] = prefix_sum_map.get(current_sum, 0) + 1
    return count

# Example
arr = [1, 1, 1]
k = 2
result = subarray_sum_equals_k(arr, k)
print(f"Number of subarrays with sum {k}: {result}")

arr = [1, 2, 3]
k = 3
result = subarray_sum_equals_k(arr, k)
print(f"Number of subarrays with sum {k}: {result}")
```

Time Complexity: O(n), where n is the size of the array.

Space Complexity: O(n), in the worst case, the hash map might store the prefix sums for all n elements.

Example:

For arr = [1, 1, 1] and k = 2:

- Initialize prefix_sum_map = {0: 1}, current_sum = 0, count = 0.
- Element 1 (index 0): current_sum = 1. 1 - 2 = -1 not in map. prefix_sum_map = {0: 1, 1: 1}.

- Element 1 (index 1): current_sum = 2. 2 - 2 = 0 in map (frequency 1). count = 1. prefix_sum_map = {0: 1, 1: 1, 2: 1}.
- Element 1 (index 2): current_sum = 3. 3 - 2 = 1 in map (frequency 1). count = 1 + 1 = 2. prefix_sum_map = {0: 1, 1: 1, 2: 1, 3: 1}.
- Result: count = 2.

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 a suitable example.

Algorithm:

1. Count the frequency of each element in the array using a hash map (dictionary).
2. Create a min-heap (priority queue) of size *k*.
3. Iterate through the hash map:
    - For each element and its frequency, push the (frequency, element) pair into the min-heap.
    - If the size of the min-heap exceeds *k*, pop the smallest element (lowest frequency) from the heap.
4. The min-heap now contains the *k* most frequent elements. Extract them from the heap and return them.

Program (Python):

```
import heapq
from collections import Counter

def k_most_frequent(arr, k):
    """
    Finds the k most frequent elements in an array.

    Args:
        arr: The input array.
        k: The number of most frequent elements to find.

    Returns:
        A list of the k most frequent elements.
    """
    if not arr or k <= 0:
        return []
```

```
    # Count element frequencies
    counts = Counter(arr)

    # Create a min-heap
    heap = []
    for element, frequency in counts.items():
        heapq.heappush(heap, (frequency, element))
        if len(heap) > k:
            heapq.heappop(heap)

    # Extract the k most frequent elements from the heap
    result = [element for frequency, element in heap]
    return result

# Example
arr = [1, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 5]
k = 2
result = k_most_frequent(arr, k)
print(f"The {k} most frequent elements are: {result}")  # Output: [3, 1]
```

Time Complexity: O(N log k), where N is the length of the array. Counting frequencies is O(N). Heap operations (push and pop) are O(log k), and we do at most N of them.
Space Complexity: O(N + k). O(N) for the frequency map and O(k) for the heap.

Example:

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

1. Frequencies: {1: 3, 2: 2, 3: 4, 4: 2, 5: 1}
2. Heap:
    - Push (3, 1): [(3, 1)]
    - Push (2, 2): [(2, 2), (3, 1)]
    - Push (4, 3): [(2, 2), (3, 1), (4, 3)]
    - Pop (2, 2): [(3, 1), (4, 3)]
    - Push (2, 4): [(2, 4), (3, 1), (4, 3)]
    - Pop (2,4): [(3, 1), (4, 3)]
    - Push (1, 5): [(1, 5), (3, 1), (4, 3)]
    - Pop (1,5): [(3,1), (4,3)]
3. Result: [3, 1]

36. Generate all subsets of a given array. Write its algorithm, program. Find

its time and space complexities. Explain with a suitable example.

Algorithm:

We can use a backtracking approach:

1. Initialize an empty list subsets to store all generated subsets.
2. Define a recursive function generate_subsets(index, current_subset):
    - Add the current_subset to the subsets list.
    - Iterate from index to the end of the array:
        - Include the current element arr[i] in the current_subset.
        - Recursively call generate_subsets(i + 1, current_subset).
        - Exclude the current element arr[i] from the current_subset (backtrack).
3. Call generate_subsets(0, []) to start the process.
4. Return the subsets list.

Program (Python):

```
def get_all_subsets(arr):
  """
  Generates all subsets of a given array.

  Args:
    arr: The input array.

  Returns:
    A list of all subsets (including the empty set).
  """
  subsets = []

  def generate_subsets(index, current_subset):
    subsets.append(current_subset[:])  # Add a copy to avoid modification

    for i in range(index, len(arr)):
      current_subset.append(arr[i])
      generate_subsets(i + 1, current_subset)
      current_subset.pop()  # Backtrack

  generate_subsets(0, [])
  return subsets
```

```
# Example
arr = [1, 2, 3]
result = get_all_subsets(arr)
print(f"All subsets of {arr} are: {result}")
# Output: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

Time Complexity: O(2N), where N is the length of the array. Each element can either be in a subset or not, leading to 2N possible combinations.
Space Complexity: O(N), the maximum depth of the recursion is N, and we store a subset of at most size N. The result list can have 2N subsets, but that's the size of the output, not the auxiliary space used by the algorithm.
Example:

arr = [1, 2, 3]

The subsets are:

- []
- [1]
- [1, 2]
- [1, 2, 3]
- [1, 3]
- [2]
- [2, 3]
- [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 a suitable example.

Algorithm:

We'll use a backtracking approach, similar to the subset problem, with some modifications to handle the target sum and uniqueness:

1. Sort the input array candidates. This helps in skipping duplicate combinations.
2. Initialize an empty list combinations to store the results.
3. Define a recursive function find_combinations(index, current_combination, current_sum):
    - If current_sum equals target, add current_combination to combinations.
    - If current_sum exceeds target, return (backtrack).
    - Iterate from index to the end of the candidates array:
        - If i > index and candidates[i] == candidates[i - 1], continue (skip

duplicates).
    - Include candidates[i] in current_combination.
    - Recursively call find_combinations(i + 1, current_combination, current_sum + candidates[i]).
    - Exclude candidates[i] from current_combination (backtrack).
4. Call find_combinations(0, [], 0) to start the process.
5. Return the combinations list.

Program (Python):

```
def combination_sum(candidates, target):
    """
    Finds all unique combinations of numbers that sum to a target.

    Args:
        candidates: A list of candidate numbers (without duplicates).
        target: The target sum.

    Returns:
        A list of lists, where each inner list is a unique combination.
    """
    candidates.sort()  # Sort to handle duplicates
    combinations = []

    def find_combinations(index, current_combination, current_sum):
        if current_sum == target:
            combinations.append(current_combination[:])
            return
        if current_sum > target:
            return

        for i in range(index, len(candidates)):
            if i > index and candidates[i] == candidates[i - 1]:
                continue  # Skip duplicates
            current_combination.append(candidates[i])
            find_combinations(i + 1, current_combination, current_sum + candidates[i])
            current_combination.pop()

    find_combinations(0, [], 0)
    return combinations
```

```
# Example
candidates = [2, 3, 6, 7]
target = 7
result = combination_sum(candidates, target)
print(f"Combinations that sum to {target}: {result}")
# Output: [[2, 2, 3], [7]]
```

Time Complexity: O(2N) in the worst case, where N is the number of candidates. In the worst case, it explores all possible combinations. However, the actual time complexity can be much better if the target is small or there are many duplicate candidates.
Space Complexity: O(N), the maximum depth of the recursion is N, and we store a combination of at most size N.

Example:

candidates = [2, 3, 6, 7], target = 7

The unique 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 a suitable example.

Algorithm:

We can use backtracking to generate all permutations:

1. Initialize an empty list permutations to store the results.
2. Define a recursive function generate_permutations(index, current_permutation):
    - If index reaches the length of the array, add current_permutation to permutations.
    - Iterate from index to the end of the array:
        - Swap current_permutation[index] and current_permutation[i].
        - Recursively call generate_permutations(index + 1, current_permutation).
        - Swap current_permutation[index] and current_permutation[i] (backtrack to restore the original order).
3. Call generate_permutations(0, arr.copy()) to start the process (pass a copy to avoid modifying the original array).
4. Return the permutations list.

Program (Python):

```
def get_all_permutations(arr):
    """
    Generates all permutations of a given array.

    Args:
        arr: The input array.

    Returns:
        A list of all permutations.
    """
    permutations = []

    def generate_permutations(index, current_permutation):
        if index == len(arr):
            permutations.append(current_permutation[:])  # Add a copy
            return

        for i in range(index, len(arr)):
            # Swap elements
            current_permutation[index], current_permutation[i] =
current_permutation[i], current_permutation[index]
            generate_permutations(index + 1, current_permutation)
            # Backtrack (swap back)
            current_permutation[index], current_permutation[i] =
current_permutation[i], current_permutation[index]

    generate_permutations(0, arr.copy())
    return permutations

# Example
arr = [1, 2, 3]
result = get_all_permutations(arr)
print(f"All permutations of {arr} are: {result}")
# 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 length of the array. There are N! possible permutations.
Space Complexity: O(N), the maximum depth of the recursion is N, and we store a permutation of size N. The result list has N! permutations, but that's output space.
Example:

arr = [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:

- A subset is a selection of elements from a set, where the order of the elements does not matter.
- A subset can contain any number of elements, from 0 (the empty set) to all the elements of the original set.
- Repetitions of elements are not considered in subsets. {1, 2} is the same subset as {2, 1}.

Example:

Let the set be S = {1, 2, 3}.

The subsets of S are:

- {} (empty set)
- {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 the elements matters.
- A permutation must contain a specific number of elements. We can talk about permutations of *k* elements from a set of *n* elements. If *k* = *n*, it's a permutation of the entire set.
- Repetitions of elements are not considered in permutations.

Example:

Let the set be S = {1, 2, 3}. We'll consider permutations of all 3 elements (k = 3).

The permutations of S are:

- (1, 2, 3)
- (1, 3, 2)
- (2, 1, 3)
- (2, 3, 1)
- (3, 1, 2)
- (3, 2, 1)

Key Differences Summarized:

Feature	Subset	Permutation
Order	Does not matter	Matters
Number of elements	Can be any number (0 to size of set)	Specific number (k)
Repetitions	Not considered	Not considered
Formula	2n (for all subsets of a set of size n)	n! / (n-k)! (for permutations of k elements from a set of size n)

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 a suitable example.

Algorithm:

1. Create a hash map (dictionary) frequency_map to store the frequency of each element in the array.
2. Iterate through the array:
    - For each element, increment its count in frequency_map.
3. Initialize max_frequency = 0 and max_element = None.
4. Iterate through the frequency_map:
    - If the frequency of the current element is greater than max_frequency:
        - Update max_frequency with the current element's frequency.
        - Update max_element with the current element.
5. Return max_element.

Program (Python):

```
def find_max_frequency_element(arr):
    """
    Finds the element with the maximum frequency in an array.

    Args:
        arr: The input array.

    Returns:
        The element with the maximum frequency (or None if the array is empty).
    """
    if not arr:
        return None

    frequency_map = {}
    for element in arr:
        frequency_map[element] = frequency_map.get(element, 0) + 1

    max_frequency = 0
    max_element = None
    for element, frequency in frequency_map.items():
        if frequency > max_frequency:
            max_frequency = frequency
            max_element = element

    return max_element

# Example
arr = [1, 2, 3, 2, 1, 4, 2, 5]
result = find_max_frequency_element(arr)
print(f"The element with the maximum frequency is: {result}")  # Output: 2
```

Time Complexity: O(N), where N is the length of the array. We iterate through the array once to build the frequency map and once to find the maximum frequency.
Space Complexity: O(N), in the worst case, the hash map might store all unique elements of the array.

Example:

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

1. Frequency Map: {1: 2, 2: 3, 3: 1, 4: 1, 5: 1}
2. Max Frequency Element: 2 (frequency 3)

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

Algorithm:

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

1. Initialize max_so_far to negative infinity (or the first element of the array) and current_max to 0.
2. Iterate through the array:
    - Update current_max as the maximum of the current element and the sum of current_max and the current element.
    - Update max_so_far as the maximum of max_so_far and current_max.
3. Return max_so_far.

Program (Python):

```
import sys

def max_subarray_sum(arr):
    """
    Finds the maximum subarray sum using Kadane's algorithm.

    Args:
        arr: The input array of numbers.

    Returns:
        The maximum sum of any contiguous subarray within the input array.
    """
    max_so_far = -sys.maxsize  # Initialize with negative infinity
    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(arr)
```

```
print(f"Maximum subarray sum: {max_sum}")  # Output: 6
```

Time Complexity: O(N), where N is the length of the array. We iterate through the array only once.
Space Complexity: O(1), as we use only a constant amount of extra space.

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

Concept of Dynamic Programming:

Dynamic programming (DP) is an algorithmic technique for solving optimization problems by breaking them down into smaller overlapping subproblems, solving each subproblem only once, and storing the results (often in a table) to avoid redundant computations. It's applicable when a problem exhibits the properties of:

- **Overlapping Subproblems:** The problem can be divided into subproblems that are reused multiple times.
- **Optimal Substructure:** The optimal solution to the problem can be constructed from the optimal solutions to its subproblems.

Use of Dynamic Programming in Solving the Maximum Subarray Problem:

While Kadane's algorithm is the standard and most efficient way to solve the maximum subarray problem, dynamic programming principles can be applied. We can define a subproblem as finding the maximum subarray sum ending at index *i*.

Let max_so_far[i] be the maximum subarray sum ending at index *i*. Then we have the following recurrence relation:

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

This relation has optimal substructure because the maximum subarray sum ending at *i* depends on the maximum subarray sum ending at *i-1*. It also has overlapping subproblems because to calculate max_so_far[i], we need max_so_far[i-1], and so on.

Dynamic Programming Approach (Less Efficient than Kadane's):

1. Create an array max_so_far of the same size as the input array arr.
2. Initialize max_so_far[0] = arr[0].
3. Iterate from i = 1 to the end of the array:
    - Calculate max_so_far[i] = max(arr[i], max_so_far[i-1] + arr[i]).

4. The maximum subarray sum is the maximum value in the max_so_far array.

Why Kadane's is Preferred:

Kadane's algorithm is essentially an optimized version of this dynamic programming approach. It avoids the need to store the entire max_so_far array by keeping track of only the current maximum sum, making it more space-efficient.

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 a suitable example.

Algorithm:

1. Count the frequency of each element in the array using a hash map (dictionary).
2. Create a min-heap (priority queue) of size *k*.
3. Iterate through the hash map:
    - For each element and its frequency, push the (frequency, element) pair into the min-heap.
    - If the size of the min-heap exceeds *k*, pop the smallest element (lowest frequency) from the heap.
4. The min-heap now contains the *k* most frequent elements. Extract them from the heap and return them.

Program (Python):

```
import heapq
from collections import Counter

def k_most_frequent(arr, k):
    """
    Finds the k most frequent elements in an array.

    Args:
        arr: The input array.
        k: The number of most frequent elements to find.

    Returns:
        A list of the k most frequent elements.
    """
    if not arr or k <= 0:
```

```
        return []

    # Count element frequencies
    counts = Counter(arr)

    # Create a min-heap
    heap = []
    for element, frequency in counts.items():
        heapq.heappush(heap, (frequency, element))
        if len(heap) > k:
            heapq.heappop(heap)

    # Extract the k most frequent elements from the heap
    result = [element for frequency, element in heap]
    return result

# Example
arr = [1, 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 5]
k = 2
result = k_most_frequent(arr, k)
print(f"The {k} most frequent elements are: {result}")  # Output: [3, 1]
```

Time Complexity: O(N log k), where N is the length of the array. Counting frequencies is O(N). Heap operations (push and pop) are O(log k), and we do at most N of them.
Space Complexity: O(N + k). O(N) for the frequency map and O(k) for the heap.
Example:

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

1. Frequencies: {1: 3, 2: 2, 3: 4, 4: 2, 5: 1}
2. Heap:
    - Push (3, 1): [(3, 1)]
    - Push (2, 2): [(2, 2), (3, 1)]
    - Push (4, 3): [(2, 2), (3, 1), (4, 3)]
    - Pop (2, 2): [(3, 1), (4, 3)]
    - Push (2, 4): [(2, 4), (3, 1), (4, 3)]
    - Pop (2,4): [(3, 1), (4, 3)]
    - Push (1, 5): [(1, 5), (3, 1), (4, 3)]
    - Pop (1,5): [(3,1), (4,3)]
3. Result: [3, 1]

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 a suitable example.

Algorithm:

1. Create a hash set seen to store the numbers encountered so far.
2. Iterate through the array arr:
    - For each number num in arr, calculate the complement complement = target - num.
    - Check if the complement is present in the seen set.
        - If yes, return the pair (complement, num).
    - Add the current num to the seen set.
3. If no such pair is found, return None or an appropriate message.

Program (Python):

```
def find_pair_with_sum(arr, target):
    """
    Finds two numbers in an array that add up to a target using hashing.

    Args:
        arr: The input array.
        target: The target sum.

    Returns:
        A tuple containing the two numbers, or None if no such pair exists.
    """
    seen = set()
    for num in arr:
        complement = target - num
        if complement in seen:
            return (complement, num)
        seen.add(num)
    return None

# Example
arr = [2, 7, 11, 15]
target = 9
result = find_pair_with_sum(arr, target)
print(f"Pair with sum {target}: {result}")  # Output: (2, 7)
```

```
arr = [2, 7, 11, 15]
target = 10
result = find_pair_with_sum(arr, target)
print(f"Pair with sum {target}: {result}")
```

Time Complexity: O(N), where N is the length of the array. We iterate through the array only once. Hash set operations (add and check) take O(1) on average.
Space Complexity: O(N), in the worst case, we might store all the elements of the array in the seen set.
Example:

arr = [2, 7, 11, 15], target = 9

1. seen = {}
2. num = 2, complement = 9 - 2 = 7. 7 not in seen. seen = {2}.
3. num = 7, complement = 9 - 7 = 2. 2 is in seen. Return (2, 7).

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 similar to a regular queue, but with an added feature: each element has an associated priority. While a regular queue follows the First-In, First-Out (FIFO) principle, a priority queue dequeues elements based on their priority.

Key properties:

- Elements are associated with a priority (usually a numerical value).
- The element with the highest priority (or lowest, depending on the implementation) is always at the front of the queue.
- Elements with the same priority are typically dequeued in FIFO order.

Common implementations:

- **Heap:** Binary heap (min-heap or max-heap) is the most common and efficient implementation.
- **Sorted Array:** Simple but less efficient for insertion and deletion.
- **Linked List:** Can be used, but less efficient than heaps.

Applications in Algorithm Design:

Priority queues are used in various algorithms and applications:

1. **Scheduling:**

    - **Job Scheduling:** Processes with higher priority are executed before lower-priority ones.
    - **Task Scheduling:** Tasks in an operating system or real-time system can be scheduled based on their importance.
2. **Graph Algorithms:**
    - **Dijkstra's Algorithm:** Finding the shortest path in a graph. A min-priority queue is used to select the vertex with the smallest distance.
    - **Prim's Algorithm:** Finding the minimum spanning tree in a graph. A min-priority queue is used to select the edge with the smallest weight.
3. **Data Compression:**
    - **Huffman Coding:** Building a Huffman tree for data compression. A min-priority queue is used to repeatedly merge the nodes with the smallest frequencies.
4. **Event-Driven Simulation:** Processing events in order of their occurrence time.
5. **Heap Sort:** An efficient sorting algorithm that uses a heap.
6. **Bandwidth Management:** Prioritizing network traffic based on the type of data.
7. **Best-First Search:** In artificial intelligence, priority queues are used in search algorithms like A* search.

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 a suitable example.

Algorithm:

We can use the "expand around center" approach:

1. Initialize longest_palindrome to an empty string.
2. Iterate through the string s:
    - For each character s[i], consider it as the center of a potential palindrome.
    - Expand around the center for both odd-length and even-length palindromes:
        - Odd length: left = i, right = i.
        - Even length: left = i, right = i + 1.
    - While left >= 0 and right < len(s) and s[left] == s[right]:
        - Expand: left--, right++.
    - Calculate the current palindrome: current_palindrome = s[left + 1:right].
    - If the length of current_palindrome is greater than the length of longest_palindrome, update longest_palindrome.
3. Return longest_palindrome.

Program (Python):

```
def longest_palindromic_substring(s):
    """
    Finds the longest palindromic substring in a given string.

    Args:
        s: The input string.

    Returns:
        The longest palindromic substring.
    """
    longest_palindrome = ""

    for i in range(len(s)):
        # Odd length palindromes
        left = i
        right = i
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        current_palindrome = s[left + 1:right]
        if len(current_palindrome) > len(longest_palindrome):
            longest_palindrome = current_palindrome

        # Even length palindromes
        left = i
        right = i + 1
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        current_palindrome = s[left + 1:right]
        if len(current_palindrome) > len(longest_palindrome):
            longest_palindrome = current_palindrome

    return longest_palindrome

# Example
string = "babad"
result = longest_palindromic_substring(string)
```

```
print(f"Longest palindromic substring: {result}")  # Output: "bab" or "aba"

string = "cbbd"
result = longest_palindromic_substring(string)
print(f"Longest palindromic substring: {result}")
```

Time Complexity: O(N2), where N is the length of the string. We expand around each character, and in the worst case, the expansion can take O(N) time.
Space Complexity: O(1), as we use only a constant amount of extra space.
Example:

s = "babad"

The algorithm expands around each character:

- b: "b"
- a: "bab", "a"
- b: "bab", "b"
- a: "a"
- d: "d"

Longest palindrome: "bab" or "aba"

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

Concept of Histogram Problems:

A histogram is a graphical representation of the distribution of data. In the context of algorithm problems, a histogram is often represented as an array of integers, where each integer represents the height of a bar and the width of each bar is assumed to be 1.

Histogram problems involve finding certain properties or solving optimization problems related to these bar representations. The key idea is to efficiently process the array of heights to extract meaningful information.

Applications in Algorithm Design:

Histogram problems appear in various applications and serve as a basis for solving other computational challenges:

1. **Largest Rectangular Area in a Histogram:** Finding the largest rectangle that can be inscribed within a histogram. This problem has applications in:

    - **Image Processing:** Finding the largest homogeneous region in an image.
    - **Data Analysis:** Finding the maximum area under a curve.
    - **Resource Allocation:** Optimizing the allocation of resources.
2. **Water Trapping Problem:** Calculating the amount of water that can be trapped between bars of a histogram after rainfall. This problem is relevant to:
    - **Civil Engineering:** Designing dams and reservoirs.
    - **Environmental Science:** Modeling water retention in landscapes.
3. **Finding the Nearest Smaller Elements:** For each bar in a histogram, finding the nearest smaller bar to its left and right. This can be useful in:
    - **Stock Market Analysis:** Finding price support and resistance levels.
    - **Compiler Design:** Analyzing code blocks.
4. **Maximum Area Submatrix in a Binary Matrix:** This problem can be solved using histogram techniques. Given a binary matrix, find the largest rectangular submatrix containing only 1s.
5. **Simplified Bar Charts:** Many real-world scenarios can be modeled as histograms, allowing us to apply histogram problem-solving techniques.

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 a suitable example.

Algorithm:

1. Find the largest index i such that arr[i] < arr[i + 1]. If no such i exists, the array is in descending order, and it's the last permutation. In this case, reverse the entire array and return.
2. Find the largest index j > i such that arr[j] > arr[i].
3. Swap arr[i] and arr[j].
4. Reverse the subarray arr[i + 1 ... n - 1].

Program (Python):

```
def next_permutation(arr):
    """
    Finds the next lexicographically greater permutation of a given array.

    Args:
        arr: The input array of integers.

    Returns:
        The next permutation of the array.  Modifies the array in place.
    """
    n = len(arr)
```

```
    # Step 1: Find the largest i such that arr[i] < arr[i + 1]
    i = n - 2
    while i >= 0 and arr[i] >= arr[i + 1]:
        i -= 1

    if i == -1:  # Array is in descending order (last permutation)
        arr.reverse()
        return

    # Step 2: Find the largest j > i such that arr[j] > arr[i]
    j = n - 1
    while arr[j] <= arr[i]:
        j -= 1

    # Step 3: Swap arr[i] and arr[j]
    arr[i], arr[j] = arr[j], arr[i]

    # Step 4: Reverse the subarray arr[i + 1 ... n - 1]
    left = i + 1
    right = n - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1

# Example
arr = [1, 2, 3]
next_permutation(arr)
print(f"Next permutation: {arr}")  # Output: [1, 3, 2]

arr = [3, 2, 1]
next_permutation(arr)
print(f"Next permutation: {arr}")  # Output: [1, 2, 3]
```

Time Complexity: O(N), where N is the length of the array. Each step takes at most O(N) time.
Space Complexity: O(1), as we modify the array in place and use only a constant amount of extra space.

Example:

arr = [1, 2, 3]

1. i = 1 (because arr[1] < arr[2], i.e., 2 < 3)
2. j = 2 (because arr[2] > arr[1], i.e., 3 > 2)
3. Swap arr[1] and arr[2]: arr becomes [1, 3, 2]
4. Reverse arr[2...2] (which is just arr[2]): arr remains [1, 3, 2]

Next permutation: [1, 3, 2]

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.

Algorithm:

1. Find the lengths of both linked lists, len1 and len2.
2. Find the difference in lengths: diff = abs(len1 - len2).
3. Move the pointer of the longer list forward by diff nodes. Now, both pointers are at the same distance from the end of their respective lists.
4. Iterate through both lists simultaneously, comparing the nodes.
5. If the nodes are the same, return the intersecting node.
6. If either pointer reaches the end of its list without finding an intersection, return None.

Program (Python):

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

def get_intersection_node(head1, head2):
    """
    Finds the intersection node of two linked lists.

    Args:
        head1: The head of the first linked list.
        head2: The head of the second linked list.

    Returns:
        The intersection node, or None if there is no intersection.
    """
    len1 = 0
    len2 = 0
```

```
    curr1 = head1
    curr2 = head2

    # Calculate lengths
    while curr1:
        len1 += 1
        curr1 = curr1.next
    while curr2:
        len2 += 1
        curr2 = curr2.next

    curr1 = head1
    curr2 = head2
    diff = abs(len1 - len2)

    # Move the pointer of the longer list forward
    if len1 > len2:
        for _ in range(diff):
            curr1 = curr1.next
    else:
        for _ in range(diff):
            curr2 = curr2.next

    # Iterate and compare nodes
    while curr1 and curr2:
        if curr1 == curr2:
            return curr1
        curr1 = curr1.next
        curr2 = curr2.next

    return None

# Example
# Create the linked lists:
# 1 -> 2 -> 3 -> 4 -> 5
#       ^
#       |
#    6 -> 7
node1 = ListNode(1)
node2 = ListNode(2)
```

```
node3 = ListNode(3)
node4 = ListNode(4)
node5 = ListNode(5)
node6 = ListNode(6)
node7 = ListNode(7)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

node6.next = node7
node7.next = node3  # Intersection at node 3

head1 = node1
head2 = node6

intersection_node = get_intersection_node(head1, head2)
if intersection_node:
    print(f"Intersection at node with value: {intersection_node.val}")  # Output: 3
else:
    print("No intersection.")
```

Time Complexity: O(M + N), where M and N are the lengths of the two linked lists.
Space Complexity: O(1), as we use only a constant amount of extra space.

Example:

List 1: 1 -> 2 -> 3 -> 4 -> 5
List 2: 6 -> 7 -> 3 -> 4 -> 5
The intersection node is 3.

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

Concept of Equilibrium Index:

An equilibrium index in an array is an index such that the sum of elements at indices lower than it is equal to the sum of elements at indices greater than it.

For an array arr of size n, an index i is an equilibrium index if:

arr[0] + arr[1] + ... + arr[i-1] = arr[i+1] + arr[i+2] + ... + arr[n-1]

If i = 0, the sum of the left side is considered 0. Similarly, if i = n - 1, the sum of the right side is considered 0.

Applications in Array Problems:

The concept of equilibrium index can be applied to various array problems:

1. **Array Balancing:** Finding an equilibrium index helps identify a point in the array where the "weight" of the elements on the left is balanced with the "weight" of the elements on the right.
2. **Prefix Sum Problems:** The equilibrium index problem is related to prefix sum calculations. Efficiently computing prefix sums can help in finding equilibrium indices.
3. **Divide and Conquer Algorithms:** The concept can be used in divide-and-conquer strategies for solving certain array-related problems.
4. **Load Balancing:** In distributed systems, the idea of equilibrium can be extended to balance the load across different nodes.
5. **Game Theory:** Equilibrium points in certain games can be related to the concept of equilibrium indices in arrays.