1. Counting Elements

Python
```
def count_elements(arr):
  """
  Counts the number of elements x in arr such that x + 1 is also in arr.

  Args:
      arr: A list of integers.

  Returns:
      The number of elements satisfying the condition.
  """
  seen = set(arr)
  return sum(x + 1 in seen for x in arr)
```

2. Perform String Shifts

Python
```
def perform_string_shifts(s, shift):
  """
  Performs left or right shifts on a string based on a shift matrix.

  Args:
      s: The original string.
      shift: A list of lists, where each sublist represents [direction,
amount].

  Returns:
      The shifted string.
  """
  amount = 0
  for direction, amt in shift:
    amount = (amount + amt) if direction else (-amount) % len(s)  # Handle
negative amounts

  return s[amount:] + s[:amount]
```

3. Leftmost Column with at Least a One

Python
```
def leftmost_column_with_one(binaryMatrix):
  """
  Finds the leftmost column with at least a 1 in a row-sorted binary
matrix.

  Args:
      binaryMatrix: A BinaryMatrix interface object.

  Returns:
      The index of the leftmost column (0-indexed) or -1 if not found.
  """
  cols = binaryMatrix.dimensions()[1]
  col = 0
  while col < cols and binaryMatrix.get(0, col) == 0:
    col += 1
  return col if col < cols else -1
```

4. First Unique Number

Python
```
from collections import OrderedDict

class FirstUnique:
  """
  Maintains a queue and keeps track of the first unique element.
  """
  def __init__(self, nums):
    self.nums = OrderedDict.fromkeys(nums)  # Maintain insertion order for
uniqueness

  def showFirstUnique(self):
    """
    Returns the first unique element or -1 if none exists.
    """
    for num, count in self.nums.items():
      if count == 1:
        return num
    return -1

  def add(self, value):
    """
    Adds a value to the queue, updating its count.
    """
    self.nums[value] = self.nums.get(value, 0) + 1  # Increment count
```

5. Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree
(This problem requires a custom tree data structure and logic to traverse it, which is beyond the scope of a general Python solution. However, here's a conceptual outline assuming you have a `TreeNode` class representing a binary tree node with `val` and `left`, `right` child references):

Python
```
def is_valid_sequence(root, arr):
  """
  Checks if a given sequence of integers can be formed by traversing a path
in the binary tree.

  Args:
      root: The root node of the binary tree.
      arr: The sequence of integers to check.

  Returns:
      True if the sequence is valid, False otherwise.
  """
  def dfs(node, i):
    if not node or i == len(arr):
      return node is None and i == len(arr)  # Check if both node and index
are exhausted
    return node.val == arr[i] and (dfs(node.left, i + 1) or dfs(node.right,
i + 1))

  return dfs(root, 0)
```

6. Kids With the Greatest Number of Candies

Python
```
def kids_with_canes(candies, extraCandies):
  """
  Determines which kids will have the greatest number of candies after
receiving extra candies.

  Args:
      candies: A list of integers representing the initial candies for each
kid.
      extraCandies: An integer representing the number of extra candies to
distribute.

  Returns:
      A list of booleans indicating if a kid has the greatest number of
candies (True) or not (False).
  """
  max_candies = max(candies)
  return [candy + extraCandies >= max_candies for candy in candies]
```

7. Max Difference You Can Get From Changing an Integer

Python
```
def max_diff(num):
  """
  Calculates the maximum difference achievable by replacing two digits in a
number.

  Args:
      num: An integer.

  Returns:
      The maximum difference.
  """
  def helper(n, d):
    res = 0
    for digit in str(n):
      for new_digit in '0123456789':
        if digit != new_digit:
          new_num = int(str(n).replace(digit, new_digit))
          res = max(res, abs(new_num - d))
    return res

  return helper(num, num)
```

8. Check If a String Can Break Another String

Python
```
def can_break(s1, s2):
  """
  Determines if one string can lexicographically "break" another string
(i.e., appear before it in a dictionary sorting).

  Args:
      s1: The first string.
      s2: The second string.

  Returns:
      True if s1 can break s2, False otherwise.
  """
```

```
  i = 0
  j = 0
  while i < len(s1) and j < len(s2):
    if s1[i] != s2[j]:
      return s1[i] < s2[j]  # Check which character comes first
alphabetically
    i += 1
    j += 1
  return i == len(s1) and j < len(s2)  # s1 is a prefix of s2
```

9. Reverse Words in a String

```python
def reverse_words(s):
  """
  Reverses the order of words in a string, preserving whitespace.

  Args:
      s: The string to reverse.

  Returns:
      The reversed string.
  """
  return ' '.join(s.strip().split()[::-1])  # Split, reverse order, join
back with spaces
```

10.**Longest Substring Without Repeating Characters**

Python

```
from collections import defaultdict

def longest_substring(s):
  """
  Finds the length of the longest substring without repeating characters.

  Args:
      s: The string to search.

  Returns:
      The length of the longest substring.
  """
  left, max_length = 0, 0
  char_freq = defaultdict(int)  # Track character frequencies

  for i, char in enumerate(s):
    char_freq[char] += 1
    while char_freq[char] > 1:  # Shrink the window if current char is not
unique
      char_freq[s[left]] -= 1
      left += 1
    max_length = max(max_length, i - left + 1)

  return max_length
```