1)
```
def minimumDeleteSum(s1, s2):
        m, n = len(s1), len(s2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(m-1, -1, -1):
        dp[i][n] = dp[i+1][n] + ord(s1[i])

        for j in range(n-1, -1, -1):
        dp[m][j] = dp[m][j+1] + ord(s2[j])

        for i in range(m-1, -1, -1):
        for j in range(n-1, -1, -1):
        if s1[i] == s2[j]:
                dp[i][j] = dp[i+1][j+1]
        else:
                dp[i][j] = min(dp[i+1][j] + ord(s1[i]), dp[i][j+1] + ord(s2[j]))

        return dp[0][0]

s1 = "sea"
s2 = "eat"
result = minimumDeleteSum(s1, s2)
print(result)  # Output: 231
```

2)
```
def isValid(s):
   stack = []

   for char in s:
      if char == '(' or char == '*':
         stack.append(char)
      else:
         if stack and stack[-1] == '(':
            stack.pop()
         elif stack and stack[-1] == '*':
            stack.pop()
            if stack and stack[-1] == '(':
               stack.pop()
         else:
            return False

   # Check for any remaining '(' characters in the stack
   while stack and stack[-1] == '(':
      stack.pop()

   return len(stack) == 0
```

```
s = "()"
result = isValid(s)
print(result)  # Output: True
```

3)

```
def minDistance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Fill the DP table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
word1 = "sea"
word2 = "eat"
result = minDistance(word1, word2)
print(result)  # Output: 2
```

4)

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def str2tree(s):
    if not s:
        return None

    # Find the index of the first opening parenthesis
    idx = s.find('(')

    if idx == -1:
        # No opening parenthesis, so the entire string is a value
        return TreeNode(int(s))

    # Extract the value from the substring before the opening parenthesis
    val = int(s[:idx])

    # Count the parentheses to find the substring for the left and right subtrees
    count = 0
    for i in range(idx, len(s)):
        if s[i] == '(':
            count += 1
        elif s[i] == ')':
            count -= 1

        if count == 0:
            # Recursive call to construct the left and right subtrees
            left = str2tree(s[idx + 1:i])
            right = str2tree(s[i + 2:-1])
            return TreeNode(val, left, right)

    # If no closing parenthesis is found, the entire string is a value
    return TreeNode(val)

s = "4(2(3)(1))(6(5))"
result = str2tree(s)

# Helper function to convert the binary tree to a list for easier visualization
def treeToList(node):
    if not node:
        return []
    return [node.val] + treeToList(node.left) + treeToList(node.right)

output = treeToList(result)
print(output)  # Output: [4, 2, 3, 1, 6, 5]
```

5)

```
def compress(chars):
        n = len(chars)
        write = 0
        count = 1

        for read in range(1, n+1):
        if read < n and chars[read] == chars[read - 1]:
        count += 1
        else:
        chars[write] = chars[read - 1]
        write += 1
        if count > 1:
                count_str = str(count)
                for digit in count_str:
                chars[write] = digit
                write += 1
```

```
    count = 1

    return write

chars = ["a","a","b","b","c","c","c"]
result = compress(chars)
print(result)  # Output: 6
print(chars[:result])  # Output: ['a', '2', 'b', '2', 'c', '3']
```

6)

```
from collections import Counter

def findAnagrams(s, p):
    result = []
    m, n = len(p), len(s)
    p_count = Counter(p)
    s_count = Counter(s[:m-1])

    for i in range(m-1, n):
    s_count[s[i]] += 1
    if s_count == p_count:
    result.append(i - m + 1)
    s_count[s[i - m + 1]] -= 1
    if s_count[s[i - m + 1]] == 0:
    del s_count[s[i - m + 1]]

    return result

s = "cbaebabacd"
p = "abc"
result = findAnagrams(s, p)
print(result)  # Output: [0, 6]
```

7)

```
def decodeString(s):
    stack = []
    curr_str = ""
    curr_num = 0

    for char in s:
    if char.isdigit():
    curr_num = curr_num * 10 + int(char)
    elif char == "[":
    stack.append(curr_str)
    stack.append(curr_num)
    curr_str = ""
    curr_num = 0
    elif char == "]":
    num = stack.pop()
    prev_str = stack.pop()
    curr_str = prev_str + num * curr_str
    else:
    curr_str += char

    return curr_str

s = "3[a]2[bc]"
result = decodeString(s)
print(result)  # Output: "aaabcbc"
```

8)

```
def buddyStrings(s, goal):
        if len(s) != len(goal):
        return False

        if s == goal:
        # Check if there are any duplicate characters in s
        seen = set()
        for char in s:
        if char in seen:
                return True
        seen.add(char)
        return False

        mismatched = []
        for i in range(len(s)):
        if s[i] != goal[i]:
        mismatched.append((s[i], goal[i]))

        return len(mismatched) == 2 and mismatched[0] == mismatched[1][::-1]

s = "ab"
goal = "ba"
result = buddyStrings(s, goal)
print(result)  # Output: True
```