Longest Common Prefix -

```python
def longest_common_prefix(strs):
    if not strs:
        return ""

    prefix = strs[0]  # Assume the first string is the prefix

    for string in strs[1:]:
        while not string.startswith(prefix):
            prefix = prefix[:-1]  # Remove the last character from prefix
            if not prefix:
                return ""  # If prefix becomes empty, return ""

    return prefix

# Test cases
print(longest_common_prefix(["flower", "flow", "flight"]))  # Output: "fl"
print(longest_common_prefix(["dog", "racecar", "car"]))     # Output: ""
print(longest_common_prefix(["interspecies", "interstellar", "interstate"]))  # Output: "int
print(longest_common_prefix([""]))                          # Output: ""
print(longest_common_prefix(["a"]))                         # Output: "a"
print(longest_common_prefix(["abc", "abcde", "abcdef"]))    # Output: "abc"
```

Largest Odd Number -

```python
# Write a function largest_odd_number(s) that takes a numeric string s and
# returns the largest odd-numbered substring that can be formed by removing some (or no)
# digits from the right. If no odd number exists, return an empty string.

Qodo Gen: Options | Test this function
def largest_odd_number(s):
    for i in range(len(s) - 1, -1, -1):  # Iterate from the last character to the first
        if int(s[i]) % 2 == 1:  # Check if the digit is odd
            return s[:i+1]  # Return the substring up to the last odd digit
    return ""  # Return empty string if no odd digit is found

# Test cases
print(largest_odd_number("51"))     # "51"
print(largest_odd_number("52"))     # "5"
print(largest_odd_number("35428"))  # "354"
print(largest_odd_number("2468"))   # "" (No odd digit)
print(largest_odd_number("13579"))  # "13579" (Already odd)
print(largest_odd_number("8642"))   # "" (No odd digit)
```

Remove Outermost Parenthesis -

```python
def remove_outermost_parenthesis(s):
    # Check if input is valid
    if not s or s[0] != '(' or s[-1] != ')' or s.count('(') != s.count(')'):
        return "Invalid input: Unbalanced or incorrectly formatted parentheses"

    result = []
    count = 0

    for char in s:
        if char == '(' and count > 0:
            result.append(char)
        elif char == ')' and count > 1:
            result.append(char)

        if char == '(':
            count += 1
        elif char == ')':
            count -= 1

    return ''.join(result)

# Test cases
print(remove_outermost_parenthesis("(()())"))   # "()()"
print(remove_outermost_parenthesis("(())"))     # "()"
```

Valid Anagrams - An **anagram** is a word or phrase formed by rearranging the letters of another word or phrase, using **all the original letters exactly once**.

1. Using Sorting (Simple Approach)

```
def is_anagram(s, t):
    return sorted(s) == sorted(t)

# Example usage:
print(is_anagram("listen", "silent"))  # Output: True
print(is_anagram("hello", "world"))    # Output: False
```

Explanation:

- Sorting both strings ensures they contain the same characters in the same frequency.
- Time Complexity: **O(n log n)** due to sorting.

2. Using HashMap (Efficient Approach)

```
from collections import Counter

def is_anagram(s, t):
    return Counter(s) == Counter(t)

# Example usage:
print(is_anagram("listen", "silent"))  # Output: True
print(is_anagram("hello", "world"))    # Output: False
```

Explanation:

- `Counter` counts the frequency of each character in both strings and compares them.
- Time Complexity: **O(n)** (faster than sorting).