**Python String Manipulation Functions**

**1. remove\_outermost\_parenthesis(s)**

**🔹 Purpose:**

Removes the **outermost parentheses** from a valid parenthesis string.

**🔹 Why This Approach?**

* Instead of using a stack (which requires extra space), we use a **counting method** (open\_count).
* This allows us to determine when parentheses are at the outermost level and avoid adding them.

**🔹 How It Works:**

1. **Track opening brackets count (open\_count)**.
2. **Append characters to result[] only if they are not outermost**:
   * If char == '(' and open\_count > 0, add it (to skip the first ().
   * If char == ')' and open\_count > 1, add it (to skip the last )).
3. **Modify open\_count**:
   * Increment for (.
   * Decrement for ).

**🔹 Time Complexity:**

* **O(n)** - We iterate through the string once (n is the length of s).
* **O(n) space** - We store the modified string in a list.

**🔹 Code:**

def remove\_outermost\_parenthesis(s):

result = []

open\_count = 0

for char in s:

if char == '(' and open\_count > 0:

result.append(char)

elif char == ')' and open\_count > 1:

result.append(char)

if char == '(':

open\_count += 1

elif char == ')':

open\_count -= 1

return ''.join(result)

**🔹 Example Usage:**

print(remove\_outermost\_parenthesis("(()())")) # Output: "()()"

**2. reverse\_words(s)**

**🔹 Purpose:**

Reverses the order of words in a string.

**🔹 Why This Approach?**

* Instead of manually splitting and reversing words using a loop, we use Python's **split()** and **list slicing** ([::-1]), which is more efficient.

**🔹 How It Works:**

1. **Split the string** into words using .split() (splits on whitespace).
2. **Remove empty strings** (if any extra spaces exist).
3. **Reverse the list of words** using [::-1].
4. **Join the reversed list back** into a string using ' '.join().

**🔹 Time Complexity:**

* **O(n)** - We iterate over the string **once** for splitting, reversing, and joining.

**🔹 Code:**

def reverse\_words(s):

words = [word for word in s.split() if word]

return ' '.join(words[::-1])

**🔹 Example Usage:**

print(reverse\_words("my name is PK")) # Output: "PK is name my"

**3 . largest\_odd\_number(s)**

**🔹 Purpose:**

Finds the **largest odd number** that can be formed by trimming digits from the end.

**🔹 Why This Approach?**

* Instead of checking all possible numbers, we **scan from right to left** and return the substring as soon as we find an odd digit.

**🔹 How It Works:**

1. **Iterate from the last digit** to the first (range(len(s) - 1, -1, -1)).
2. If **digit is odd (int(s[i]) % 2 == 1)**, return s[:i+1].
3. If no odd digit is found, return an **empty string**.

**🔹 Time Complexity:**

* **O(n)** - In the worst case, we scan the entire string once.

**🔹 Code:**

def largest\_odd\_number(s):

for i in range(len(s)-1, -1, -1):

if int(s[i]) % 2 == 1:

return s[:i+1]

return ""

**🔹 Example Usage:**

print(largest\_odd\_number("354270")) # Output: "35427"

print(largest\_odd\_number("24680")) # Output: ""

**4 . longest\_common\_substring(words)**

**🔹 Purpose:**

Finds the **longest substring** that is common across all words.

**🔹 Why This Approach?**

* Instead of checking all possible substrings, we **start with the shortest word** and test substrings from it. This reduces unnecessary comparisons.

**🔹 Time Complexity:**

* **O(n \* m²)**, where:
  + n is the number of words.
  + m is the length of the shortest word.

**🔹 Code:**

def longest\_common\_substring(words):

if not words:

return ""

shortest = min(words, key=len)

longest\_substr = ""

for i in range(len(shortest)):

for j in range(i + 1, len(shortest) + 1):

substr = shortest[i:j]

if all(substr in word for word in words):

if len(substr) > len(longest\_substr):

longest\_substr = substr

else:

break

return longest\_substr

**🔹 Example Usage:**

words = ["flight", "lightning", "slight"]

print(longest\_common\_substring(words)) # Output: "light"

**5 . are\_rotations(s1, s2)**

**🔹 Purpose:**

Checks if two strings are **rotations of each other**.

**🔹 Why This Approach?**

* Instead of manually rotating the string, we **concatenate s1 + s1** and check if s2 exists inside.

**🔹 Time Complexity:**

* **O(n)** - String concatenation and substring search are linear.

**🔹 Code:**

def are\_rotations(s1, s2):

if len(s1) != len(s2):

return False

return s2 in (s1 + s1)

**🔹 Example Usage:**

print(are\_rotations("abcd", "cdab")) # Output: True

print(are\_rotations("abcd", "cdba")) # Output: False

**6. is\_anagram(s1, s2)**

**🔹 Purpose:**

Checks if two words are **anagrams**.

**🔹 Why This Approach?**

* We use **character frequency counting** instead of sorting (O(n log n)), making it **faster**.

**🔹 Time Complexity:**

* **O(n)** - Counting characters and comparing dictionaries.

**🔹 Code:**

def is\_anagram(s1, s2):

if len(s1) != len(s2):

return False

char\_count\_s1 = {}

char\_count\_s2 = {}

for char in s1:

char\_count\_s1[char] = char\_count\_s1.get(char, 0) + 1

for char in s2:

char\_count\_s2[char] = char\_count\_s2.get(char, 0) + 1

return char\_count\_s1 == char\_count\_s2

**🔹 Example Usage:**

print(is\_anagram("listen", "silent")) # Output: True

print(is\_anagram("hello", "world")) # Output: False

**Final Takeaway**

| **Function** | **Approach** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- | --- |
| remove\_outermost\_parenthesis() | Counting ( ) | **O(n)** | **O(n)** |
| reverse\_words() | Splitting & reversing | **O(n)** | **O(n)** |
| largest\_odd\_number() | Scanning from right | **O(n)** | **O(1)** |
| longest\_common\_substring() | Brute-force | **O(n \* m²)** | **O(1)** |
| are\_rotations() | s1 + s1 trick | **O(n)** | **O(n)** |
| is\_anagram() | Frequency dictionary | **O(n)** | **O(1)** |