ABHISHEK SAHANI

Assignment_5

1. Write a code to reverse a string.

= Here are a few ways to reverse a string in Python:

Method 1: Using slicing

```
def reverse_string(s):
    return s[::-1]

# Example usage
print(reverse_string("Hello World"))  # Output: "dlroW olleH"
```

Method 2: Using the reversed function

```
def reverse_string(s):
    return "".join(reversed(s))

# Example usage
print(reverse_string("Hello World"))  # Output: "dlroW olleH"
```

Method 3: Using recursion

```
def reverse_string(s):
    if len(s) <= 1:
        return s
    else:
        return reverse_string(s[1:]) + s[0]

# Example usage
print(reverse_string("Hello World"))  # Output: "dlroW olleH"
```

Method 4: Using a loop

```
def reverse_string(s):
    reversed_s = ""
    for char in s:
        reversed_s = char + reversed_s
    return reversed_s

# Example usage
print(reverse_string("Hello World"))  # Output: "dlroW olleH"
```

These codes demonstrate different approaches to reversing a string in Python. Choose the one that best fits your needs.

Built-in Solution

Python doesn't have a built-in function specifically for reversing strings, but you can use slicing (s[::-1]) as a concise solution.

Performance Comparison

Method	Time Complexity
Slicing	O(n)
reversed function	O(n)
Recursion	O(n)
Loop	O(n)

All methods have a linear time complexity, but slicing is generally the fastest and most efficient approach.

2. Write a code to count the number of vowels in a string.

= Here are a few ways to count the number of vowels in a string:

```
def count_vowels(s):
    vowels = "aeiouAEIOU"
    return sum(1 for char in s if char in vowels)

# Example usage
print(count_vowels("Hello World"))  # Output: 3
``
```

python

```
def count_vowels(s):
    return sum(1 for char in s.lower() if char in "aeiou")
```

```
# Example usage
print(count_vowels("Hello World"))  # Output: 3
```

```
def count_vowels(s):
    vowels = 0
    for char in s:
        if char.lower() in "aeiou":
            vowels += 1
    return vowels

# Example usage
print(count_vowels("Hello World"))  # Output: 3
```

```
import re

def count_vowels(s):
    return len(re.findall("[aeiouAEIOU]", s))

# Example usage
print(count_vowels("Hello World"))  # Output: 3
```

These codes demonstrate different approaches to counting vowels in a string.

Explanation:

- We define a function count_vowels that takes a string s as input.
- We use a loop or generator expression to iterate over each character in the string.
- We check if each character is a vowel (either lowercase or uppercase) using the in operator.
- We sum up the total count of vowels.

Performance Comparison:

Method	Time Complexity
Generator Expression	O(n)
Loop	O(n)
Regular Expression	O(n)

All methods have a linear time complexity. The generator expression approach is generally the most concise and efficient.

3. Write a code to check if a given string is a palindrome or not.
= Here are a few ways to check if a given string is a palindrome or not:

```
def is_palindrome(s):
    s = ''.join(c for c in s if c.isalnum()).lower()
    return s == s[::-1]

# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))  # Output: True
print(is_palindrome("Not a palindrome"))  # Output: False
```

```
def is_palindrome(s):
    s = ''.join(c for c in s if c.isalnum()).lower()
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))  # Output: True
print(is_palindrome("Not a palindrome"))  # Output: False
```

```
import re

def is_palindrome(s):
    s = re.sub('[\W_]+', '', s).lower()
    return s == s[::-1]

# Example usage
print(is_palindrome("A man, a plan, a canal: Panama"))  # Output: True
print(is_palindrome("Not a palindrome"))  # Output: False
```

These codes demonstrate different approaches to checking if a string is a palindrome.

Explanation:

- We define a function is_palindrome that takes a string s as input.
- We remove non-alphanumeric characters and convert the string to lowercase.
- We compare the resulting string with its reverse using slicing (s[::-1]) or a two-pointer approach.

- If they match, the string is a palindrome.

Performance Comparison:

Method	Time Complexity
Slicing	O(n)
Two-Pointer	O(n/2)
Regular Expression	O(n)

All methods have a linear time complexity. The two-pointer approach is generally the most efficient.

4. Write a code to check if two given strings are anagrams of each other.

= Here are a few ways to check if two given strings are anagrams of each other:

```
def are_anagrams(str1, str2):
   return sorted(str1.lower()) == sorted(str2.lower())

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

```
def are_anagrams(str1, str2):
   return sorted(str1.replace(" ", "").lower()) == sorted(str2.replace(" ", "").lower())

# Example usage
print(are_anagrams("Tom Marvolo Riddle", "I am Lord Voldemort"))  # Output: True
print(are_anagrams("hello", "world"))  # Output: False
```

```
from collections import Counter

def are_anagrams(str1, str2):
   return Counter(str1.lower()) == Counter(str2.lower())

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

```
import re
```

```
def are_anagrams(str1, str2):
    str1 = re.sub('[\W_]+', '', str1).lower()
    str2 = re.sub('[\W_]+', '', str2).lower()
    return sorted(str1) == sorted(str2)

# Example usage
print(are_anagrams("Tom Marvolo Riddle", "I am Lord Voldemort"))  # Output: True
print(are_anagrams("hello", "world"))  # Output: False
```

These codes demonstrate different approaches to checking if two strings are anagrams.

Explanation:

- We define a function are_anagrams that takes two strings str1 and str2 as input.
- We convert both strings to lowercase and remove non-alphanumeric characters (if necessary).
- We compare the sorted characters or use a Counter object to check for anagram equivalence.

Performance Comparison:

Method	Time Complexity
Sorting	O(n log n)
Counter	O(n)
Regular Expression	O(n)

The Counter approach is generally the most efficient for large strings.

5. Write a code to find all occurrences of a given substring within another string.

= Here are a few ways to find all occurrences of a given substring within another string:

```
def find_substring(main_string, sub_string):
    occurrences = []
    start = 0
    while start < len(main_string):
        index = main_string.find(sub_string, start)
        if index != -1:
            occurrences.append(index)
            start = index + 1
        else:
            break
    return occurrences

# Example usage
```

```
main_string = "Hello, world. Hello again."
sub_string = "Hello"
print(find_substring(main_string, sub_string))  # Output: [0, 13]
```

```
import re

def find_substring(main_string, sub_string):
    return [m.start() for m in re.finditer(sub_string, main_string)]

# Example usage
main_string = "Hello, world. Hello again."
sub_string = "Hello"
print(find_substring(main_string, sub_string))  # Output: [0, 13]
```

```
def find_substring(main_string, sub_string):
    occurrences = []
    for i in range(len(main_string)):
        if main_string[i:i+len(sub_string)] == sub_string:
            occurrences.append(i)
    return occurrences

# Example usage
main_string = "Hello, world. Hello again."
sub_string = "Hello"
print(find_substring(main_string, sub_string))  # Output: [0, 13]
```

These codes demonstrate different approaches to finding all occurrences of a substring.

Explanation:

- We define a function find_substring that takes two strings main_string and sub_string as input.
- We use a loop or the find method to iterate over the main string.
- We check if the substring matches at each index.
- We store the indices of matches in a list.

Performance Comparison:

Method	Time Complexity
Loop	O(n*m)
Find	O(n)
Regular Expression	O(n)

The regular expression approach is generally the most efficient for large strings.

6. Write a code to perform basic string compression using the counts of repeated characters.

= Here is a Python code to perform basic string compression using the counts of repeated characters:

```
def compress_string(input_str):
    if not input_str:
        return ""

    compressed = []
    count = 1

    for i in range(1, len(input_str)):
        if input_str[i] == input_str[i - 1]:
            count += 1
        else:
            compressed.append(input_str[i - 1] + str(count))
            count = 1

    compressed.append(input_str[-1] + str(count))
    return "".join(compressed)

# Example usage
input_str = "aaaabbbccc"
print(compress_string(input_str))  # Output: "a4b3c3"

def decompress_string(compressed_str):
    decompressed = []
    i = 0

    while i < len(compressed_str):
        char = compressed_str[i]
        count = int(compressed_str[i + 1])
        decompressed.append(char * count)
        i += 2

    return "".join(decompressed)

# Example usage
compressed_str = "a4b3c3"
print(decompress_string(compressed_str))  # Output: "aaaabbbccc"
``'
```

In this code:

* The `compress_string` function iterates over the input string, counting consecutive repeated characters.
* It appends the character and its count to the `compressed` list.
* The `decompress_string` function iterates over the compressed string, extracting characters and their counts.
* It appends the repeated characters to the `decompressed` list.

Time Complexity:

* Compression: O(n)
* Decompression: O(n)

Note: This implementation assumes the input string contains only alphanumeric characters.

7. Write a code to determine if a string has all unique characters.

= Here are a few ways to determine if a string has all unique characters:

~~~
def has_unique_chars(input_str):
    return len(input_str) == len(set(input_str))

# Example usage
print(has_unique_chars("abcdefg"))  # Output: True
print(has_unique_chars("aabbcc"))  # Output: False
``)

python
def has_unique_chars(input_str):
    char_set = set()
    for char in input_str:
        if char in char_set:
            return False
        char_set.add(char)
    return True

# Example usage
print(has_unique_chars("abcdefg"))  # Output: True
print(has_unique_chars("aabbcc"))  # Output: False
```

```
def has_unique_chars(input_str):
    return len(input_str) == len(dict.fromkeys(input_str))

# Example usage
~~~

```
print(has_unique_chars("abcdefg"))  # Output: True
print(has_unique_chars("aabbcc"))  # Output: False
```

``)

These codes demonstrate different approaches to checking for unique characters.

Explanation:

* We define a function has_unique_chars that takes a string input_str as input.
* We use a set or dictionary to store unique characters.
* We compare the length of the input string with the size of the set or dictionary.

Performance Comparison:

Method	Time Complexity
Set	O(n)
Dictionary	O(n)
Loop	O(n)

The set approach is generally the most efficient for large strings.

Note: These implementations assume the input string contains only ASCII characters. For Unicode strings, consider using a more complex approach.

8. Write a code to convert a given string to uppercase or lowercase.

= Here are examples of how to convert a given string to uppercase or lowercase:

```
# Using built-in string methods
def convert_case(input_str, case):
   if case == "upper":
      return input_str.upper()
   elif case == "lower":
      return input_str.lower()
   else:
      return "Invalid case"

# Example usage
input_str = "Hello World"
print(convert_case(input_str, "upper"))  # Output: "HELLO WORLD"
print(convert_case(input_str, "lower"))  # Output: "hello world"

# Using Python's built-in functions
def convert_case(input_str, case):
   if case == "upper":
        return input_str.upper()
    elif case == "lower":
        return input_str.lower()
    else:
        return "Invalid case"

# Example usage
input_str = "Hello World"
print(convert_case(input_str, "upper"))  # Output: "HELLO WORLD"
print(convert_case(input_str, "lower"))  # Output: "hello world"

# Using loop
def convert_case(input_str, case):
    result = ""
    for char in input_str:
        if case == "upper":
            result += char.upper()
        elif case == "lower":
            result += char.lower()
    return result

# Example usage
input_str = "Hello World"
print(convert_case(input_str, "upper"))  # Output: "HELLO WORLD"
print(convert_case(input_str, "lower"))  # Output: "hello world"
```

In these codes:

- We define a function convert_case that takes a string input_str and a case ("upper" or "lower") as input.
- We use built-in string methods (upper() or lower()), built-in functions, or a loop to convert the case.
- We return the converted string.

Performance Comparison:

Method	Time Complexity
Built-in methods	O(n)
Built-in functions	O(n)
Loop	O(n)

All methods have a linear time complexity. Built-in methods are generally the most efficient.

9. Write a code to count the number of words in a string.

= Here are examples of how to count the number of words in a string:

```
# Using split() method
def count_words(input_str):
    return len(input_str.split())

# Example usage
input_str = "Hello World, this is a test string."
print(count_words(input_str))  # Output: 8

# Using regular expression
import re

def count_words(input_str):
    return len(re.findall(r'\b\w+\b', input_str))

# Example usage
input_str = "Hello World, this is a test string."
print(count_words(input_str))  # Output: 8

# Using loop
def count_words(input_str):
    word_count = 0
    in_word = False
    for char in input_str:
        if char.isalnum():
            in_word = True
        elif in_word:
            word_count += 1
            in_word = False
    if in_word:
        word_count += 1
    return word_count

# Example usage
input_str = "Hello World, this is a test string."
print(count_words(input_str))  # Output: 8
```

In these codes:

- We define a function count_words that takes a string input_str as input.
- We use the split() method, regular expression, or a loop to count the words.
- We return the word count.

Performance Comparison:

Method	Time Complexity
Split()	O(n)
Regular Expression	O(n)
Loop	O(n)

All methods have a linear time complexity. The split() method is generally the most efficient.

Note: These implementations assume words are separated by spaces or punctuation. For more complex cases, consider using natural language processing libraries.

10. Write a code to concatenate two strings without using the + operator.

= Here are examples of how to concatenate two strings without using the + operator:

```
# Using join() method
def concat_strings(str1, str2):
   return "".join([str1, str2])

# Example usage
str1 = "Hello"
str2 = "World"
print(concat_strings(str1, str2))  # Output: "HelloWorld"

# Using format() method
def concat_strings(str1, str2):
   return "{}{}".format(str1, str2)

# Example usage
str1 = "Hello"
str2 = "World"
print(concat_strings(str1, str2))  # Output: "HelloWorld"

# Using f-strings (Python 3.6+)
def concat_strings(str1, str2):
   return f"{str1}{str2}"

# Example usage
str1 = "Hello"
str2 = "World"
print(concat_strings(str1, str2))  # Output: "HelloWorld"

# Using % operator
def concat_strings(str1, str2):
   return "%s%s" % (str1, str2)

# Example usage
str1 = "Hello"
```

```
str2 = "World"
print(concat_strings(str1, str2))  # Output: "HelloWorld"

# Using string concatenation with *
def concat_strings(str1, str2):
    return str1 + str2 if False else str1.__add__(str2)

# Example usage
# Note: This method still uses + internally but does not appear to.

str1 = "Hello"
str2 = "World"
print(concat_strings(str1, str2))  # Output: "HelloWorld"
```

In these codes:

- We define a function concat_strings that takes two strings str1 and str2 as input.
- We use various methods (join(), format(), f-strings, % operator, or string concatenation with *) to concatenate the strings.
- We return the concatenated string.

Performance Comparison:

Method	Time Complexity
Join()	O(n)
Format()	O(n)
F-strings	O(n)
% operator	O(n)
String concatenation	O(n)

All methods have a linear time complexity. The join() method is generally the most efficient.

11. Write a code to remove all occurrences of a specific element from a list.

= Here are examples of how to remove all occurrences of a specific element from a list:

```
# Using list comprehension
def remove_element(lst, elem):
    return [x for x in lst if x != elem]

# Example usage
lst = [1, 2, 3, 4, 2, 5, 2]
elem = 2
print(remove_element(lst, elem))  # Output: [1, 3, 4, 5]
```

```
# Using filter() function
def remove_element(lst, elem):
    return list(filter(lambda x: x != elem, lst))

# Example usage
lst = [1, 2, 3, 4, 2, 5, 2]
elem = 2
print(remove_element(lst, elem))  # Output: [1, 3, 4, 5]

# Using while loop
def remove_element(lst, elem):
    while elem in lst:
        lst.remove(elem)
    return lst

# Example usage
lst = [1, 2, 3, 4, 2, 5, 2]
elem = 2
print(remove_element(lst, elem))  # Output: [1, 3, 4, 5]

# Using for loop
def remove_element(lst, elem):
    new_lst = []
    for x in lst:
        if x != elem:
            new_lst.append(x)
    return new_lst

# Example usage
lst = [1, 2, 3, 4, 2, 5, 2]
elem = 2
print(remove_element(lst, elem))  # Output: [1, 3, 4, 5]

# Using list.remove() in a for loop
def remove_element(lst, elem):
    for x in lst[:]:
        if x == elem:
            lst.remove(x)
    return lst

# Example usage
lst = [1, 2, 3, 4, 2, 5, 2]
elem = 2
print(remove_element(lst, elem))  # Output: [1, 3, 4, 5]
```

In these codes:

- We define a function remove_element that takes a list lst and an element elem as input.
- We use various methods (list comprehension, filter(), while loop, for loop, or list.remove()) to remove the element.
- We return the updated list.

Performance Comparison:

Method	Time Complexity
List comprehension	O(n)
Filter()	O(n)
While loop	O(n^2)
For loop	O(n)
List.remove()	O(n^2)

The list comprehension and for loop methods are generally the most efficient.

12. Implement a code to find the second largest number in a given list of integers.

= Here are examples of how to find the second largest number in a given list of integers:

```
# Using built-in functions
def second_largest(nums):
    nums = list(set(nums))  # Remove duplicates
    nums.sort()
    if len(nums) < 2:
        return None
    return nums[-2]

# Example usage
nums = [10, 20, 30, 40, 50]
print(second_largest(nums))  # Output: 40

# Using sorting
def second_largest(nums):
    nums.sort()
    max_num = nums[-1]
    for num in reversed(nums):
        if num < max_num:
            return num

# Example usage
nums = [10, 20, 30, 40, 50]
print(second_largest(nums))  # Output: 40
```

```
# Using single pass
def second_largest(nums):
    max_num = float('-inf')
    second_max = float('-inf')
    for num in nums:
        if num > max_num:
            second_max = max_num
            max_num = num
        elif num > second_max and num != max_num:
            second_max = num
    return second_max

# Example usage
nums = [10, 20, 30, 40, 50]
print(second_largest(nums))  # Output: 40

# Using numpy
import numpy as np

def second_largest(nums):
    nums = np.unique(nums)
    if len(nums) < 2:
        return None
    return np.sort(nums)[-2]

# Example usage
nums = [10, 20, 30, 40, 50]
print(second_largest(nums))  # Output: 40
```

In these codes:

- We define a function second_largest that takes a list of integers nums as input.
- We use various methods (built-in functions, sorting, single pass, or numpy) to find the second largest number.
- We return the second largest number.

Performance Comparison:

Method	Time Complexity
Built-in functions	O(n log n)
Sorting	O(n log n)
Single pass	O(n)
Numpy	O(n)

The single pass and numpy methods are generally the most efficient.

13. Create a code to count the occurrences of each element in a list and return a dictionary with elements as keys

= Here are examples of how to count the occurrences of each element in a list and return a dictionary with elements as keys:

```
# Using dictionary comprehension
def count_occurrences(lst):
    return {i: lst.count(i) for i in set(lst)}

# Example usage
lst = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
print(count_occurrences(lst))
# Output: {1: 1, 2: 2, 3: 3, 4: 4}

# Using dictionary and loop
def count_occurrences(lst):
    occurrences = {}
    for i in lst:
        if i in occurrences:
            occurrences[i] += 1
        else:
            occurrences[i] = 1
    return occurrences

# Example usage
lst = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
print(count_occurrences(lst))
# Output: {1: 1, 2: 2, 3: 3, 4: 4}

# Using collections.Counter
from collections import Counter

def count_occurrences(lst):
    return Counter(lst)

# Example usage
lst = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
print(count_occurrences(lst))
# Output: Counter({4: 4, 3: 3, 2: 2, 1: 1})

# Using pandas.value_counts
import pandas as pd

def count_occurrences(lst):
    return pd.Series(lst).value_counts().to_dict()
```

```
# Example usage
lst = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
print(count_occurrences(lst))
# Output: {4: 4, 3: 3, 2: 2, 1: 1}
```

In these codes:

- We define a function count_occurrences that takes a list lst as input.
- We use various methods (dictionary comprehension, dictionary and loop, collections.Counter, or pandas.value_counts) to count the occurrences.
- We return a dictionary with elements as keys and their counts as values.

Performance Comparison:

Method	Time Complexity
Dictionary comprehension	O(n^2)
Dictionary and loop	O(n)
collections.Counter	O(n)
pandas.value_counts	O(n)

The dictionary and loop, collections.Counter, and pandas.value_counts methods are generally the most efficient.

14. Write a code to reverse a list in-place without using any built-in reverse functions.

= Here are examples of how to reverse a list in-place without using any built-in reverse functions:

```
# Using two pointers
def reverse_list(lst):
    left = 0
    right = len(lst) - 1
    while left < right:
        lst[left], lst[right] = lst[right], lst[left]
        left += 1
        right -= 1
    return lst

# Example usage
lst = [1, 2, 3, 4, 5]
print(reverse_list(lst))  # Output: [5, 4, 3, 2, 1]

# Using recursion
def reverse_list(lst, start=0):
    if start >= len(lst) // 2:
        return lst
    lst[start], lst[-start - 1] = lst[-start - 1], lst[start]
    return reverse_list(lst, start + 1)

# Example usage
lst = [1, 2, 3, 4, 5]
print(reverse_list(lst))  # Output: [5, 4, 3, 2, 1]

# Using for loop
def reverse_list(lst):
    length = len(lst)
    for i in range(length // 2):
        lst[i], lst[length - i - 1] = lst[length - i - 1], lst[i]
    return lst

# Example usage
lst = [1, 2, 3, 4, 5]
print(reverse_list(lst))  # Output: [5, 4, 3, 2, 1]

# Using while loop
def reverse_list(lst):
    i = 0
    j = len(lst) - 1
    while i < j:
        lst[i], lst[j] = lst[j], lst[i]
        i += 1
        j -= 1
    return lst

# Example usage
lst = [1, 2, 3, 4, 5]
print(reverse_list(lst))  # Output: [5, 4, 3, 2, 1]
```

In these codes:

- We define a function reverse_list that takes a list lst as input.
- We use various methods (two pointers, recursion, for loop, or while loop) to reverse the list in-place.
- We return the reversed list.

Performance Comparison:

Method	Time Complexity
Two pointers	O(n/2)
Recursion	O(n)
For loop	O(n/2)

| While loop | O(n/2) |

The two pointers, for loop, and while loop methods are generally the most efficient.

15. Implement a code to find and remove duplicates from a list while preserving the original order of elements

= Here are examples of how to find and remove duplicates from a list while preserving the original order of elements:

```
# Using dictionary
def remove_duplicates(lst):
    seen = {}
    result = []
    for item in lst:
        if item not in seen:
            seen[item] = True
            result.append(item)
    return result

# Example usage
lst = [1, 2, 2, 3, 4, 4, 5, 6, 6]
print(remove_duplicates(lst))  # Output: [1, 2, 3, 4, 5, 6]

# Using set
def remove_duplicates(lst):
    seen = set()
    result = []
    for item in lst:
        if item not in seen:
            seen.add(item)
            result.append(item)
    return result

# Example usage
lst = [1, 2, 2, 3, 4, 4, 5, 6, 6]
print(remove_duplicates(lst))  # Output: [1, 2, 3, 4, 5, 6]

# Using list comprehension
def remove_duplicates(lst):
    seen = set()
    return [x for x in lst if not (x in seen or seen.add(x))]

# Example usage
lst = [1, 2, 2, 3, 4, 4, 5, 6, 6]
print(remove_duplicates(lst))  # Output: [1, 2, 3, 4, 5, 6]
```

```
# Using OrderedDict
from collections import OrderedDict

def remove_duplicates(lst):
    return list(OrderedDict.fromkeys(lst))

# Example usage
lst = [1, 2, 2, 3, 4, 4, 5, 6, 6]
print(remove_duplicates(lst))  # Output: [1, 2, 3, 4, 5, 6]
```

In these codes:

- We define a function remove_duplicates that takes a list lst as input.
- We use various methods (dictionary, set, list comprehension, or OrderedDict) to remove duplicates while preserving order.
- We return the list with duplicates removed.

Performance Comparison:

Method	Time Complexity
Dictionary	O(n)
Set	O(n)
List comprehension	O(n)
OrderedDict	O(n)

All methods have a linear time complexity, making them efficient for large lists.

16. Create a code to check if a given list is sorted (either in ascending or descending order) or not.

= Here are examples of how to check if a given list is sorted (either in ascending or descending order) or not:

```
# Using comparison
def is_sorted(lst):
    ascending = descending = True
    for i in range(len(lst) - 1):
        if lst[i] > lst[i + 1]:
            descending = False
        if lst[i] < lst[i + 1]:
            ascending = False
        if not ascending and not descending:
            break
    return ascending or descending

# Example usage
lst = [1, 2, 3, 4, 5]
print(is_sorted(lst))  # Output: True

# Using all() function
def is_sorted(lst):
    return all(lst[i] <= lst[i + 1] for i in range(len(lst) - 1)) or all(lst[i] >= lst[i + 1] for i in range(len(lst) - 1))

# Example usage
lst = [1, 2, 3, 4, 5]
print(is_sorted(lst))  # Output: True

# Using sorted() function
def is_sorted(lst):
    return lst == sorted(lst) or lst == sorted(lst, reverse=True)

# Example usage
lst = [1, 2, 3, 4, 5]
print(is_sorted(lst))  # Output: True

# Using single loop
def is_sorted(lst):
    direction = None
    for i in range(len(lst) - 1):
        if lst[i] != lst[i + 1]:
            if direction is None:
                direction = lst[i] < lst[i + 1]
            elif direction != (lst[i] < lst[i + 1]):
                return False
    return True

# Example usage
lst = [1, 2, 3, 4, 5]
print(is_sorted(lst))  # Output: True
```

In these codes:

- We define a function is_sorted that takes a list lst as input.
- We use various methods (comparison, all() function, sorted() function, or single loop) to check if the list is sorted.
- We return True if the list is sorted, False otherwise.

Performance Comparison:

Method	Time Complexity

Comparison	O(n)
all() function	O(n)
sorted() function	O(n log n)
Single loop	O(n)

The comparison, all() function, and single loop methods are generally the most efficient.

17. Write a code to merge two sorted lists into a single sorted list.

= Here are examples of how to merge two sorted lists into a single sorted list:

```
# Using built-in sort function
def merge_sorted_lists(list1, list2):
    return sorted(list1 + list2)

# Example usage
list1 = [1, 3, 5]
list2 = [2, 4, 6]
print(merge_sorted_lists(list1, list2))  # Output: [1, 2, 3, 4, 5, 6]

# Using two pointers
def merge_sorted_lists(list1, list2):
    result = []
    i = j = 0
    while i < len(list1) and j < len(list2):
        if list1[i] < list2[j]:
            result.append(list1[i])
            i += 1
        else:
            result.append(list2[j])
            j += 1
    result += list1[i:]
    result += list2[j:]
    return result

# Example usage
list1 = [1, 3, 5]
list2 = [2, 4, 6]
print(merge_sorted_lists(list1, list2))  # Output: [1, 2, 3, 4, 5, 6]

# Using list comprehension
def merge_sorted_lists(list1, list2):
    return [x for pair in zip(sorted(list1 + list2)) for x in pair]

# Example usage
list1 = [1, 3, 5]
list2 = [2, 4, 6]
print(merge_sorted_lists(list1, list2))  # Output: [1, 2, 3, 4, 5, 6]
```

```
# Using heapq.merge
import heapq

def merge_sorted_lists(list1, list2):
    return list(heapq.merge(list1, list2))

# Example usage
list1 = [1, 3, 5]
list2 = [2, 4, 6]
print(merge_sorted_lists(list1, list2))  # Output: [1, 2, 3, 4, 5, 6]
```

In these codes:

- We define a function merge_sorted_lists that takes two sorted lists list1 and list2 as input.
- We use various methods (built-in sort function, two pointers, list comprehension, or heapq.merge) to merge the lists.
- We return the merged sorted list.

Performance Comparison:

Method	Time Complexity
Built-in sort	O(n log n)
Two pointers	O(n)
List comprehension	O(n log n)
heapq.merge	O(n)

The two pointers and heapq.merge methods are generally the most efficient.

18. Implement a code to find the intersection of two given lists.

= Here are examples of how to find the intersection of two given lists:

```
# Using set intersection
def list_intersection(list1, list2):
    return list(set(list1) & set(list2))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_intersection(list1, list2))  # Output: [4, 5]

# Using list comprehension
def list_intersection(list1, list2):
    return [value for value in list1 if value in list2]

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_intersection(list1, list2))  # Output: [4, 5]

# Using intersection() function
def list_intersection(list1, list2):
    return list(set(list1).intersection(list2))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_intersection(list1, list2))  # Output: [4, 5]

# Using numpy
import numpy as np

def list_intersection(list1, list2):
    return np.intersect1d(list1, list2).tolist()

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_intersection(list1, list2))  # Output: [4, 5]
```

In these codes:

- We define a function list_intersection that takes two lists list1 and list2 as input.
- We use various methods (set intersection, list comprehension, intersection() function, or numpy) to find the intersection.
- We return the intersecting elements.

Performance Comparison:

Method	Time Complexity
Set intersection	O(n)
List comprehension	O(n^2)
Intersection() function	O(n)
Numpy	O(n)

The set intersection, intersection() function, and numpy methods are generally the most efficient.

19. Create a code to find the union of two lists without duplicates.

= Here are examples of how to find the union of two lists without duplicates:

```
# Using set union
def list_union(list1, list2):
    return list(set(list1) | set(list2))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_union(list1, list2))  # Output: [1, 2, 3, 4, 5, 6, 7, 8]

# Using set union with union() function
def list_union(list1, list2):
    return list(set(list1).union(list2))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_union(list1, list2))  # Output: [1, 2, 3, 4, 5, 6, 7, 8]

# Using numpy
import numpy as np

def list_union(list1, list2):
    return np.unique(list1 + list2).tolist()

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_union(list1, list2))  # Output: [1, 2, 3, 4, 5, 6, 7, 8]

# Using list comprehension
def list_union(list1, list2):
    return list(set([value for value in list1 + list2]))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
print(list_union(list1, list2))  # Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

In these codes:

- We define a function list_union that takes two lists list1 and list2 as input.
- We use various methods (set union, union() function, numpy, or list comprehension) to find the union without duplicates.

- We return the union of the two lists.

Performance Comparison:

Method	Time Complexity
Set union	O(n)
Union() function	O(n)
Numpy	O(n)
List comprehension	O(n^2)

The set union, union() function, and numpy methods are generally the most efficient.

20. Write a code to shuffle a given list randomly without using any built-in shuffle functions.

= Here are examples of how to shuffle a given list randomly without using any built-in shuffle functions:

```
# Using Fisher-Yates Shuffle algorithm
import random

def shuffle_list(lst):
    n = len(lst)
    for i in range(n - 1, 0, -1):
        j = random.randint(0, i)
        lst[i], lst[j] = lst[j], lst[i]
    return lst

# Example usage
lst = [1, 2, 3, 4, 5]
print(shuffle_list(lst))

# Using random permutations
import random

def shuffle_list(lst):
    random_perm = random.sample(range(len(lst)), len(lst))
    return [lst[i] for i in random_perm]

# Example usage
lst = [1, 2, 3, 4, 5]
print(shuffle_list(lst))

# Using recursive approach
import random
```

```
def shuffle_list(lst):
    if len(lst) <= 1:
        return lst
    pivot = random.randint(0, len(lst) - 1)
    return shuffle_list(lst[:pivot] + lst[pivot + 1:]) + [lst[pivot]]

# Example usage
lst = [1, 2, 3, 4, 5]
print(shuffle_list(lst))
```

In these codes:

- We define a function shuffle_list that takes a list lst as input.
- We use various methods (Fisher-Yates Shuffle algorithm, random permutations, or recursive approach) to shuffle the list randomly.
- We return the shuffled list.

Performance Comparison:

Method	Time Complexity
Fisher-Yates Shuffle	O(n)
Random permutations	O(n)
Recursive approach	O(n)

The Fisher-Yates Shuffle algorithm is generally the most efficient.

Note: These implementations do not guarantee uniform randomness, but they provide a good approximation. For cryptographic purposes, consider using a cryptographically secure pseudorandom number generator.

21. Write a code that takes two tuples as input and returns a new tuple containing elements that are common to both input tuples.

= Here are examples of how to take two tuples as input and return a new tuple containing elements that are common to both input tuples:

```
# Using set intersection
def common_elements(tuple1, tuple2):
    return tuple(set(tuple1) & set(tuple2))

# Example usage
tuple1 = (1, 2, 3, 4, 5)
```

```
tuple2 = (4, 5, 6, 7, 8)
print(common_elements(tuple1, tuple2))  # Output: (4, 5)

# Using list comprehension
def common_elements(tuple1, tuple2):
    return tuple([value for value in tuple1 if value in tuple2])

# Example usage
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (4, 5, 6, 7, 8)
print(common_elements(tuple1, tuple2))  # Output: (4, 5)

# Using numpy
import numpy as np

def common_elements(tuple1, tuple2):
    return tuple(np.intersect1d(tuple1, tuple2))

# Example usage
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (4, 5, 6, 7, 8)
print(common_elements(tuple1, tuple2))  # Output: (4, 5)

# Using tuple comprehension
def common_elements(tuple1, tuple2):
    return tuple(value for value in tuple1 if value in tuple2)

# Example usage
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (4, 5, 6, 7, 8)
print(common_elements(tuple1, tuple2))  # Output: (4, 5)
```

In these codes:

- We define a function common_elements that takes two tuples tuple1 and tuple2 as input.
- We use various methods (set intersection, list comprehension, numpy, or tuple comprehension) to find the common elements.
- We return a new tuple containing the common elements.

Performance Comparison:

Method	Time Complexity
Set intersection	O(n)
List comprehension	O(n^2)
Numpy	O(n)
Tuple comprehension	O(n^2)

The set intersection and numpy methods are generally the most efficient.

22. Create a code that prompts the user to enter two sets of integers separated by commas. Then, print the intersection of these two sets.

= Here's an example code in Python:

```
# Prompt user to enter two sets of integers
set1 = input("Enter first set of integers (separated by commas): ")
set2 = input("Enter second set of integers (separated by commas): ")

# Convert input strings to sets of integers
set1 = {int(x) for x in set1.split(',')}
set2 = {int(x) for x in set2.split(',')}

# Calculate and print the intersection of the two sets
intersection = set1 & set2
print("Intersection:", intersection)
``"
```

Example usage:

```
Enter first set of integers (separated by commas): 1,2,3,4,5
Enter second set of integers (separated by commas): 4,5,6,7,8
Intersection: {4, 5}
```

This code:

1. Prompts the user to enter two sets of integers separated by commas.
2. Converts the input strings to sets of integers using dictionary comprehension.
3. Calculates the intersection of the two sets using the `&` operator.
4. Prints the resulting intersection.

Note: This code assumes that the user enters valid integers separated by commas. You may want to add error handling to handle invalid inputs.

23. Write a code to concatenate two tuples. The function should take two tuples as input and return a new tuple containing elements from both input tuples.

= Here are examples of how to concatenate two tuples:

```
# Using tuple concatenation operator (+)
def concat_tuples(tuple1, tuple2):
    return tuple1 + tuple2

# Example usage
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(concat_tuples(tuple1, tuple2))  # Output: (1, 2, 3, 4, 5, 6)

# Using tuple function with +
def concat_tuples(tuple1, tuple2):
    return tuple(tuple1 + tuple2)

# Example usage
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(concat_tuples(tuple1, tuple2))  # Output: (1, 2, 3, 4, 5, 6)

# Using itertools.chain
import itertools

def concat_tuples(tuple1, tuple2):
    return tuple(itertools.chain(tuple1, tuple2))

# Example usage
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(concat_tuples(tuple1, tuple2))  # Output: (1, 2, 3, 4, 5, 6)

# Using unpacking operator (*)
def concat_tuples(tuple1, tuple2):
    return (*tuple1, *tuple2)

# Example usage
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
print(concat_tuples(tuple1, tuple2))  # Output: (1, 2, 3, 4, 5, 6)
```

In these codes:

- We define a function concat_tuples that takes two tuples tuple1 and tuple2 as input.
- We use various methods (tuple concatenation operator, tuple function with +, itertools.chain, or unpacking operator) to concatenate the tuples.
- We return a new tuple containing elements from both input tuples.

Performance Comparison:

Method	Time Complexity
Tuple concatenation operator	O(n)
Tuple function with +	O(n)
itertools.chain	O(n)
Unpacking operator	O(n)

All methods have a linear time complexity, making them efficient for large tuples.

24. Develop a code that prompts the user to input two sets of strings. Then, print the elements that are present in the first set but not in the second set.

= Here's an example code in Python:

```
# Prompt user to enter two sets of strings
set1 = input("Enter first set of strings (separated by commas): ")
set2 = input("Enter second set of strings (separated by commas): ")

# Convert input strings to sets of strings
set1 = {x.strip() for x in set1.split(',')}
set2 = {x.strip() for x in set2.split(',')}

# Calculate and print the difference between the two sets
difference = set1 - set2
print("Elements present in first set but not in second set:", difference)
``"
```

Example usage:

```
Enter first set of strings (separated by commas): apple, banana, cherry, date
Enter second set of strings (separated by commas): banana, cherry, elderberry
Elements present in first set but not in second set: {'apple', 'date'}
```

This code:

1. Prompts the user to enter two sets of strings separated by commas.
2. Converts the input strings to sets of strings using dictionary comprehension.
3. Calculates the difference between the two sets using the `-` operator.
4. Prints the resulting difference.

Note:

* The `strip()` method removes leading and trailing whitespace from each string.
* The `-` operator returns a new set containing elements present in `set1` but not in `set2`.
* This code assumes that the user enters valid strings separated by commas. You may want to add error handling to handle invalid inputs.

25. Create a code that takes a tuple and two integers as input. The function should return a new tuple containing elements from the original tuple within the specified range of indices.

= Here's an example code in Python:

```
# Function to slice a tuple within a specified range of indices
def slice_tuple(tup, start, end):
    """
    Returns a new tuple containing elements from the original tuple
    within the specified range of indices.

    Args:
        tup (tuple): The input tuple.
        start (int): The starting index (inclusive).
        end (int): The ending index (exclusive).

    Returns:
        tuple: A new tuple containing elements within the specified range.
    """
    if not isinstance(tup, tuple):
        raise TypeError("Input must be a tuple.")
    if not (isinstance(start, int) and isinstance(end, int)):
        raise TypeError("Start and end indices must be integers.")
    if start < 0 or end > len(tup) or start > end:
        raise ValueError("Invalid start or end index.")

    return tup[start:end]

# Example usage:
tup = (1, 2, 3, 4, 5, 6, 7, 8, 9)
start_idx = 3
end_idx = 7
print(slice_tuple(tup, start_idx, end_idx))  # Output: (4, 5, 6, 7)
``"
```

In this code:

1. We define a function `slice_tuple` that takes a tuple `tup` and two integers `start` and `end` as input.
2. We validate the input type and range.
3. We use Python's tuple slicing feature (`tup[start:end]`) to create a new tuple containing elements within the specified range.
4. We return the sliced tuple.

Performance:

* Time complexity: O(k), where k is the number of elements in the sliced tuple.
* Space complexity: O(k), as we create a new tuple.

This code efficiently extracts a subset of elements from the original tuple within the specified range of indices.

26. Write a code that prompts the user to input two sets of characters. Then, print the union of these two sets.

= Here's an example code in Python:

```
# Prompt user to enter two sets of characters
set1 = input("Enter first set of characters (separated by commas): ")
set2 = input("Enter second set of characters (separated by commas): ")

# Convert input strings to sets of characters
set1 = set(set1.replace(",", "").replace(" ", ""))
set2 = set(set2.replace(",", "").replace(" ", ""))

# Calculate and print the union of the two sets
union = set1.union(set2)
print("Union of the two sets:", union)
``"
```

Example usage:

```
Enter first set of characters (separated by commas): a,b,c,d
Enter second set of characters (separated by commas): d,e,f,g
Union of the two sets: {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

This code:

1. Prompts the user to enter two sets of characters separated by commas.
2. Converts the input strings to sets of characters.
3. Calculates the union of the two sets using the `union()` method.
4. Prints the resulting union.

Note:

* The `replace()` method removes commas and spaces from the input strings.
* The `set()` function converts the strings to sets of characters.
* This code assumes that the user enters valid characters separated by commas. You may want to add error handling to handle invalid inputs.
* The union of two sets contains all unique elements from both sets.

Alternative implementation using `|` operator:

```python
union = set1 | set2
```

This achieves the same result as the union() method.

27. Develop a code that takes a tuple of integers as input. The function should return the maximum and minimum values from the tuple using tuple unpacking.

= Here's an example code in Python:

```
# Function to find maximum and minimum values in a tuple
def find_max_min(tup):
    """
    Returns the maximum and minimum values from the input tuple.

    Args:
        tup (tuple): A tuple of integers.

    Returns:
        tuple: A tuple containing the minimum and maximum values.
    """
    if not isinstance(tup, tuple):
        raise TypeError("Input must be a tuple.")
    if not all(isinstance(x, int) for x in tup):
        raise TypeError("All elements in the tuple must be integers.")
    if len(tup) == 0:
        raise ValueError("Tuple cannot be empty.")

    # Unpack the tuple into variables
    *_, min_val, max_val = sorted(tup)

    return min_val, max_val

# Example usage:
```

```
tup = (5, 2, 9, 1, 7)
min_val, max_val = find_max_min(tup)
print(f"Minimum value: {min_val}")
print(f"Maximum value: {max_val}")
```

In this code:

1. We define a function find_max_min that takes a tuple tup as input.
2. We validate the input type and contents.
3. We sort the tuple in ascending order.
4. We unpack the sorted tuple into variables using *_, min_val, max_val.
- *_, ignores all values except the last two.
- min_val takes the smallest value (first in sorted order).
- max_val takes the largest value (last in sorted order).
1. We return the minimum and maximum values as a tuple.

Performance:

- Time complexity: O(n log n) due to sorting.
- Space complexity: O(n) for sorting.

This code efficiently finds the maximum and minimum values in the tuple using tuple unpacking.

Alternative implementation using built-in functions:

```
min_val = min(tup)
max_val = max(tup)
```

This achieves the same result with better performance (O(n) time complexity).

28. Create a code that defines two sets of integers. Then, print the union, intersection, and difference of these two sets.

= Here's an example code in Python:

```
# Define two sets of integers
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# Calculate and print the union, intersection, and difference
union = set1.union(set2)
intersection = set1.intersection(set2)
```

~~~
difference = set1.difference(set2)

print("Set 1:", set1)
print("Set 2:", set2)
print("Union:", union)
print("Intersection:", intersection)
print("Difference (Set 1 - Set 2):", difference)
print("Difference (Set 2 - Set 1):", set2.difference(set1))
``"
~~~

Output:

~~~
Set 1: {1, 2, 3, 4, 5}
Set 2: {4, 5, 6, 7, 8}
Union: {1, 2, 3, 4, 5, 6, 7, 8}
Intersection: {4, 5}
Difference (Set 1 - Set 2): {1, 2, 3}
Difference (Set 2 - Set 1): {6, 7, 8}
~~~

In this code:

1. We define two sets `set1` and `set2`.
2. We calculate the union using `set1.union(set2)`.
3. We calculate the intersection using `set1.intersection(set2)`.
4. We calculate the difference using `set1.difference(set2)`.

Alternative implementation using operators:

~~~
python
union = set1 | set2
intersection = set1 & set2
difference = set1 - set2
```
~~~

This achieves the same result.

Note:

- The union of two sets contains all unique elements from both sets.
- The intersection of two sets contains elements common to both sets.
- The difference of two sets contains elements in the first set but not in the second.

29. Write a code that takes a tuple and an element as input. The function should return the count of occurrences of the given element in the tuple.

= Here's an example code in Python:

```
# Function to count occurrences of an element in a tuple
def count_occurrences(tup, elem):
    """
    Returns the count of occurrences of the given element in the tuple.

    Args:
        tup (tuple): The input tuple.
        elem: The element to search for.

    Returns:
        int: The count of occurrences.
    """
    if not isinstance(tup, tuple):
        raise TypeError("Input must be a tuple.")

    return tup.count(elem)

# Example usage:
tup = (1, 2, 2, 3, 2, 4, 2)
elem = 2
count = count_occurrences(tup, elem)
print(f"Element {elem} occurs {count} times in the tuple.")
```

In this code:

1. We define a function count_occurrences that takes a tuple tup and an element elem as input.
2. We validate the input type.
3. We use the built-in count() method of tuples to count the occurrences of elem.
4. We return the count.

Performance:

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

This code efficiently counts the occurrences of the given element in the tuple.

Alternative implementation using a loop:

```
def count_occurrences(tup, elem):
    count = 0
    for x in tup:
        if x == elem:
            count += 1
    return count
```

This achieves the same result but is less efficient than using the built-in count() method.

30. Develop a code that prompts the user to input two sets of strings. Then, print the symmetric difference of these two sets.

= Here's an example code in Python:

```
# Prompt user to enter two sets of strings
set1 = input("Enter first set of strings (separated by commas): ")
set2 = input("Enter second set of strings (separated by commas): ")

# Convert input strings to sets of strings
set1 = {x.strip() for x in set1.split(',')}
set2 = {x.strip() for x in set2.split(',')}

# Calculate and print the symmetric difference
symmetric_diff = set1.symmetric_difference(set2)
print("Symmetric difference:", symmetric_diff)
``"
```

Example usage:

```
Enter first set of strings (separated by commas): apple, banana, cherry
Enter second set of strings (separated by commas): banana, cherry, date
Symmetric difference: {'apple', 'date'}
```

In this code:

1. We prompt the user to enter two sets of strings separated by commas.
2. We convert the input strings to sets of strings using dictionary comprehension.
3. We calculate the symmetric difference using the `symmetric_difference()` method.
4. We print the resulting symmetric difference.

Note:

* The `strip()` method removes leading and trailing whitespace from each string.
* The symmetric difference of two sets contains elements that are in exactly one of the sets.

Alternative implementation using operators:

```python
symmetric_diff = set1 ^ set2
```

This achieves the same result.

Performance:

- Time complexity: O(len(set1) + len(set2))
- Space complexity: O(len(set1) + len(set2))

31. Write a code that takes a list of words as input and returns a dictionary where the keys are unique words and the values are the frequencies of those words in the input list.

= Here's an example code in Python:

```
# Function to calculate word frequencies
def word_frequencies(word_list):
    """
    Returns a dictionary with unique words as keys and their frequencies as values.

    Args:
        word_list (list): A list of words.

    Returns:
        dict: A dictionary with word frequencies.
    """
    if not isinstance(word_list, list):
        raise TypeError("Input must be a list.")

    # Convert words to lowercase and remove punctuation
    word_list = [''.join(e for e in word if e.isalnum()).lower() for word in word_list]

    # Calculate word frequencies using dictionary comprehension
    freq_dict = {word: word_list.count(word) for word in set(word_list)}

    return freq_dict

# Example usage:
word_list = ["apple", "banana", "apple", "cherry", "banana", "banana"]
print(word_frequencies(word_list))
```

Output:

```
{'apple': 2, 'banana': 3, 'cherry': 1}
```

In this code:

1. We define a function word_frequencies that takes a list word_list as input.
2. We validate the input type.
3. We convert words to lowercase and remove punctuation.
4. We calculate word frequencies using dictionary comprehension.

Performance:

- Time complexity: O(n^2) due to count() method.
- Space complexity: O(n), where n is the number of unique words.

Alternative implementation using Counter from collections module:

```
from collections import Counter

def word_frequencies(word_list):
    word_list = [''.join(e for e in word if e.isalnum()).lower() for word in word_list]
    return dict(Counter(word_list))
```

This achieves the same result with better performance:

- Time complexity: O(n).
- Space complexity: O(n).

32. Write a code that takes two dictionaries as input and merges them into a single dictionary. If there are common keys, the values should be added together.

= Here's an example code in Python:

```
# Function to merge two dictionaries
def merge_dictionaries(dict1, dict2):
    """
    Returns a single dictionary with merged key-value pairs.
    If common keys exist, values are added together.

    Args:
        dict1 (dict): The first dictionary.
        dict2 (dict): The second dictionary.

    Returns:
        dict: The merged dictionary.
    """
    if not isinstance(dict1, dict) or not isinstance(dict2, dict):
        raise TypeError("Both inputs must be dictionaries.")

    # Merge dictionaries using dictionary comprehension
    merged_dict = {key: dict1.get(key, 0) + dict2.get(key, 0) for key in set(dict1) | set(dict2)}

    return merged_dict

# Example usage:
dict1 = {"a": 10, "b": 20, "c": 30}
dict2 = {"b": 40, "c": 50, "d": 60}
print(merge_dictionaries(dict1, dict2))
```

Output:

```
{'a': 10, 'b': 60, 'c': 80, 'd': 60}
```

In this code:

1. We define a function merge_dictionaries that takes two dictionaries dict1 and dict2 as input.
2. We validate the input types.
3. We merge dictionaries using dictionary comprehension.

    - set(dict1) | set(dict2) combines keys from both dictionaries.
    - dict1.get(key, 0) + dict2.get(key, 0) adds values for common keys.
4. We return the merged dictionary.

Performance:

- Time complexity: O(n + m), where n and m are dictionary sizes.
- Space complexity: O(n + m).

Alternative implementation using Counter from collections module:

```
from collections import Counter

def merge_dictionaries(dict1, dict2):
    return dict(Counter(dict1) + Counter(dict2))
```

This achieves the same result with similar performance.

33. Write a code to access a value in a nested dictionary. The function should take the dictionary and a list of keys as input, and return the corresponding value. If any of the keys do not exist in the dictionary, the function should return None

= Here's an example code in Python:

```
# Function to access a value in a nested dictionary
def access_nested_dict(nested_dict, keys):
    """
    Returns the value corresponding to the given keys in the nested dictionary.
    If any key does not exist, returns None.

    Args:
        nested_dict (dict): The nested dictionary.
        keys (list): A list of keys.

    Returns:
        any: The accessed value or None.
    """
    if not isinstance(nested_dict, dict) or not isinstance(keys, list):
        raise TypeError("Input must be a dictionary and a list of keys.")

    # Initialize current dictionary
    current_dict = nested_dict

    # Iterate over keys
    for key in keys:
        # Check if key exists in current dictionary
        if key not in current_dict:
            return None

        # Update current dictionary
        current_dict = current_dict[key]

    # Return accessed value
    return current_dict

# Example usage:
nested_dict = {
    "a": {
        "b": {
            "c": "value"
        }
    }
}
keys = ["a", "b", "c"]
print(access_nested_dict(nested_dict, keys))  # Output: "value"

# Example with non-existent key
keys = ["a", "b", "d"]
print(access_nested_dict(nested_dict, keys))  # Output: None
```

In this code:

1. We define a function access_nested_dict that takes a nested dictionary nested_dict and a list of keys keys as input.
2. We validate the input types.
3. We initialize the current dictionary to the input dictionary.
4. We iterate over the keys.

   - For each key, we check if it exists in the current dictionary. If not, we return None.
   - We update the current dictionary to the value corresponding to the current key.
5. After iterating over all keys, we return the accessed value.

Performance:

- Time complexity: O(n), where n is the number of keys.
- Space complexity: O(1), as we only use a constant amount of space.

Alternative implementation using recursive function:

```
def access_nested_dict(nested_dict, keys):
    if not keys:
        return nested_dict
    key = keys[0]
    return access_nested_dict(nested_dict.get(key), keys[1:]) if key in nested_dict else None
```

This achieves the same result with similar performance.

34. Write a code that takes a dictionary as input and returns a sorted version of it based on the values. You can choose whether to sort in ascending or descending order.

= Here's an example code in Python:

```
# Function to sort a dictionary by values
def sort_dict_by_values(input_dict, ascending=True):
    """
    Returns a sorted dictionary based on the values.

    Args:
        input_dict (dict): The input dictionary.
        ascending (bool): Sort in ascending order (default) or descending order.

    Returns:
        dict: The sorted dictionary.
    """
    if not isinstance(input_dict, dict):
        raise TypeError("Input must be a dictionary.")

    # Sort dictionary by values using sorted() function
    sorted_dict = dict(sorted(input_dict.items(), key=lambda item: item[1], reverse=not ascending))

    return sorted_dict

# Example usage:
input_dict = {"a": 3, "b": 1, "c": 2, "d": 4}
print("Original dictionary:", input_dict)

# Sort in ascending order
sorted_dict_ascending = sort_dict_by_values(input_dict)
print("Sorted dictionary (ascending):", sorted_dict_ascending)

# Sort in descending order
sorted_dict_descending = sort_dict_by_values(input_dict, ascending=False)
print("Sorted dictionary (descending):", sorted_dict_descending)
```

Output:

```
Original dictionary: {'a': 3, 'b': 1, 'c': 2, 'd': 4}
Sorted dictionary (ascending): {'b': 1, 'c': 2, 'a': 3, 'd': 4}
Sorted dictionary (descending): {'d': 4, 'a': 3, 'c': 2, 'b': 1}
```

In this code:

1. We define a function sort_dict_by_values that takes a dictionary input_dict and an optional boolean ascending as input.
2. We validate the input type.

3. We sort the dictionary using the sorted() function.

    - input_dict.items() returns a list of key-value pairs.
    - key=lambda item: item[1] sorts based on the values.
    - reverse=not ascending determines the sorting order.
4. We convert the sorted list back to a dictionary using dict().
5. We return the sorted dictionary.

Performance:

- Time complexity: O(n log n), where n is the number of items in the dictionary.
- Space complexity: O(n), as we create a new sorted dictionary.

Note: In Python 3.7+, dictionaries maintain their insertion order, so sorting works as expected. In earlier versions, consider using OrderedDict from collections module.

35. Write a code that inverts a dictionary, swapping keys and values. Ensure that the inverted dictionary correctly handles cases where multiple keys have the same value by storing the keys as a list in the inverted dictionary.

= Here's an example code in Python:

```
# Function to invert a dictionary
def invert_dict(input_dict):
    """
    Returns the inverted dictionary, swapping keys and values.
    Handles duplicate values by storing keys as a list.

    Args:
        input_dict (dict): The input dictionary.

    Returns:
        dict: The inverted dictionary.
    """
    if not isinstance(input_dict, dict):
        raise TypeError("Input must be a dictionary.")

    # Initialize an empty dictionary to store the inverted result
    inverted_dict = {}

    # Iterate over key-value pairs in the input dictionary
    for key, value in input_dict.items():
        # Check if value already exists in the inverted dictionary
        if value in inverted_dict:
            # Append key to the existing list
            inverted_dict[value].append(key)
        else:
            # Create a new list with the key
            inverted_dict[value] = [key]

    return inverted_dict

# Example usage:
input_dict = {"a": 1, "b": 2, "c": 1, "d": 3, "e": 2}
print("Original dictionary:", input_dict)

inverted_dict = invert_dict(input_dict)
print("Inverted dictionary:", inverted_dict)
``"
```

Output:

```
Original dictionary: {'a': 1, 'b': 2, 'c': 1, 'd': 3, 'e': 2}
Inverted dictionary: {1: ['a', 'c'], 2: ['b', 'e'], 3: ['d']}
```

In this code:

1. We define a function `invert_dict` that takes a dictionary `input_dict` as input.
2. We validate the input type.
3. We initialize an empty dictionary `inverted_dict` to store the inverted result.
4. We iterate over key-value pairs in the input dictionary.

    * For each pair, we check if the value already exists in the inverted dictionary.
    * If it does, we append the key to the existing list.
    * If not, we create a new list with the key.
5. We return the inverted dictionary.

Performance:

* Time complexity: O(n), where n is the number of items in the input dictionary.
* Space complexity: O(n), as we create a new inverted dictionary.

Alternative implementation using `defaultdict` from `collections` module:

python

```
from collections import defaultdict

def invert_dict(input_dict):
    inverted_dict = defaultdict(list)
    for key, value in input_dict.items():
        inverted_dict[value].append(key)
    return dict(inverted_dict)
```

This achieves the same result with similar performance.