Dictionary, List, Tuples and Math

Dictionaries in Python are mutable, unordered collections of key-value pairs. They are one of the most versatile data structures, often used for fast lookups, mapping, and storing structured data. Keys must be immutable (e.g., strings, integers, tuples), while values can be any type (e.g., lists, other dictionaries, functions). Dictionaries are defined using curly braces `{}` or the `dict()` constructor.

Here's a detailed breakdown of basic dictionary manipulations, with examples. I'll focus on simple, fundamental operations that are commonly tested in coding interviews.

```
#### 1. **Creating Dictionaries**
   - Empty dictionary: `my_dict = {}` or `my_dict = dict()`.
   - With key-value pairs:
       `my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}`.
   - Using `dict()` with pairs:
       `my_dict = dict([('name', 'Alice'), ('age', 30)])`.
   - From keys with default values:
       `my_dict = dict.fromkeys(['a', 'b'], 0)` → `{'a': 0, 'b': 0}`.
   - **Key points**: Keys are unique;
       duplicate keys overwrite the previous value.
       Dictionaries maintain insertion order since Python 3.7.

#### 2. **Accessing Elements**
   - By key: `value = my_dict['key']`.
       Raises `KeyError` if key doesn't exist.
   - Safely with `get()`: `value = my_dict.get('key', default_value)`.
       Returns `None` or `default_value` if key missing.
   - Check if key exists: `'key' in my_dict` → `True` or `False`.
   - **Example**:
     ```python
     person = {'name': 'Bob', 'age': 25}
     print(person['name'])  # Output: Bob
     print(person.get('city', 'Unknown'))  # Output: Unknown
```
```

```
    print('age' in person)  # Output: True
    ```

3. **Adding or Updating Elements**
   - Add/update with assignment: `my_dict['new_key'] = value`.
     If key exists, updates value; else adds.
   - Using `update()`: Merges another dict or iterable of pairs.
     Overwrites existing keys.
     - `my_dict.update({'key1': val1, 'key2': val2})`.
     - Or `my_dict.update([('key1', val1), ('key2', val2)])`.
   - **Example**:
     ```python
     fruits = {'apple': 5}
     fruits['banana'] = 3  # Adds new key
     fruits['apple'] = 10  # Updates existing
     fruits.update({'orange': 4, 'banana': 6})  # Updates banana, adds orange
     print(fruits)  # {'apple': 10, 'orange': 4, 'banana': 6}
     ```

4. **Removing Elements**
   - `del my_dict['key']`: Removes key-value pair.
     Raises `KeyError` if key missing.
   - `pop(key, default)`: Removes and returns value.
     Returns `default` if key missing.
   - `popitem()`: Removes and returns last inserted key-value pair (as tuple).
     Raises `KeyError` if empty.
   - `clear()`: Empties the dictionary.
   - **Example**:
     ```python
     colors = {'red': 1, 'green': 2, 'blue': 3}
     del colors['green']  # Removes 'green'
     popped = colors.pop('blue', None)  # popped = 3
     last = colors.popitem()  # last = ('red', 1)
     colors.clear()  # Now empty
     ```

5. **Iterating Over Dictionaries**
   - Over keys: `for key in my_dict:` or `for key in my_dict.keys():`.
   - Over values: `for value in my_dict.values():`.
   - Over pairs: `for key, value in my_dict.items():`.
   - **Example**:
     ```python
```

```
    scores = {'math': 90, 'english': 85}
    for subject in scores:  # Iterates keys: 'math', 'english'
        print(subject)
    for score in scores.values():  # 90, 85
        print(score)
    for sub, sc in scores.items():  # ('math', 90), ('english', 85)
        print(sub, sc)
    ```

6. **Common Methods and Operations**
   - `len(my_dict)`: Number of key-value pairs.
   - `keys()`: Returns view of keys
     (iterable, convertible to list: `list(my_dict.keys())`).
   - `values()`: Returns view of values.
   - `items()`: Returns view of (key, value) tuples.
   - Copying: `new_dict = my_dict.copy()` (shallow copy).
   - Merging (Python 3.9+): `merged = dict1 | dict2`.
   - Default values with `setdefault(key, default)`:
     Returns value if key exists; else adds key with default and returns it.
   - **Example**:
     ```python
     inventory = {'pens': 10}
     print(len(inventory))  # 1
     print(list(inventory.keys()))  # ['pens']
     print(inventory.setdefault('pencils', 5))  # 5 (adds key)
     print(inventory)  # {'pens': 10, 'pencils': 5}
     ```

7. **Common Patterns in Interviews**
   - Counting frequencies: Use dict to count occurrences (e.g., word counts).
   - Grouping: Dict of lists (e.g., group by key).
   - Nested dicts: Access like `my_dict['outer']['inner']`.
   - Error handling: Always handle missing keys to avoid `KeyError`.
   - Efficiency: O(1) average time for lookups, inserts, deletes.

Dictionaries are hash tables under the hood, so keys must be hashable.
Avoid mutating dict while iterating (use `.copy()` or
list comprehension if needed).

Now, to master these, here are 20 basic problems. Each includes:
- Problem statement
- Solution code (as a function for reusability)
```
```

```
- Detailed explanation
- Sample function calls with expected outputs

These problems cover creation, access, add/update,
remove, iteration, and common methods. Practice them step-by-step for interviews.
```

Problem 1: Create a Dictionary from Two Lists

Create a function that takes two lists (keys and values) and returns a dictionary.

```
def create_dict(keys, values):
    return dict(zip(keys, values))  # zip pairs them, dict converts
```

Explanation: `zip()` combines lists into pairs. `dict()` creates the dictionary. Handles equal-length lists; if unequal, truncates to shorter length.

Sample calls:

```
print(create_dict(['a', 'b'], [1, 2]))  # {'a': 1, 'b': 2}
print(create_dict(['name'], ['Alice']))  # {'name': 'Alice'}
```

Problem 2: Check if Key Exists

Write a function that checks if a key exists in a dictionary and returns True/False.

```
def key_exists(my_dict, key):
    return key in my_dict
```

Explanation: The `in` operator checks keys efficiently (O(1) time). Avoids `KeyError` unlike direct access.

Sample calls:

```
print(key_exists({'apple': 5}, 'apple'))  # True
print(key_exists({'apple': 5}, 'banana'))  # False
```

Problem 3: Get Value with Default

Function to get a value for a key, return 'Not Found' if missing.

```
def get_value(my_dict, key):
    return my_dict.get(key, 'Not Found')
```

Explanation: `get()` is safe; provides default without modifying dict.

Sample calls:

```
print(get_value({'color': 'red'}, 'color'))  # red
print(get_value({'color': 'red'}, 'size'))  # Not Found
```

Problem 4: Add a Key-Value Pair

Function to add a key-value if key doesn't exist, else update.

```
def add_or_update(my_dict, key, value):
    my_dict[key] = value
    return my_dict
```

Explanation: Assignment handles both add and update. Dict is mutable, so changes in-place.

Sample calls:

```
print(add_or_update({'a': 1}, 'b', 2))  # {'a': 1, 'b': 2}
print(add_or_update({'a': 1}, 'a', 3))  # {'a': 3}
```

Problem 5: Remove a Key

Function to remove a key if it exists, return the dict.

```
def remove_key(my_dict, key):
    if key in my_dict:
        del my_dict[key]
    return my_dict
```

Explanation: Check first to avoid `KeyError`. `del` removes in-place.

Sample calls:

```
print(remove_key({'x': 10, 'y': 20}, 'x'))  # {'y': 20}
print(remove_key({'x': 10}, 'z'))  # {'x': 10} (no change)
```

Problem 6: Pop a Key

Function to pop a key and return its value or None.

```
def pop_key(my_dict, key):
    return my_dict.pop(key, None)
```

Explanation: `pop()` removes and returns value; default prevents error.

Sample calls:

```
print(pop_key({'fruit': 'apple'}, 'fruit'))  # apple (dict now empty)
print(pop_key({'fruit': 'apple'}, 'color'))  # None
```

Problem 7: Merge Two Dictionaries

Function to merge dict2 into dict1 (overwrite conflicts).

```
def merge_dicts(dict1, dict2):
    dict1.update(dict2)
    return dict1
```

Explanation: `update()` adds/updates from dict2 in-place.

Sample calls:

```
print(merge_dicts({'a': 1}, {'b': 2}))  # {'a': 1, 'b': 2}
print(merge_dicts({'a': 1}, {'a': 3}))  # {'a': 3}
```

Problem 8: Get All Keys as List

Function to return sorted list of keys.

```python
def get_keys(my_dict):
    return sorted(my_dict.keys())
```

Explanation: `keys()` gives view; `list()` or `sorted()` converts. Sorting for consistency.

Sample calls:

```python
print(get_keys({'b': 2, 'a': 1}))  # ['a', 'b']
print(get_keys({}))  # []
```

Problem 9: Get All Values as List

Function to return list of values (unsorted).

```python
def get_values(my_dict):
    return list(my_dict.values())
```

Explanation: `values()` is iterable; convert to list for return.

Sample calls:

```python
print(get_values({'one': 1, 'two': 2}))
# [1, 2] (order may vary, but since Python 3.7, insertion order)
print(get_values({}))  # []
```

Problem 10: Iterate and Sum Values

Function to sum all integer values in dict.

```python
def sum_values(my_dict):
    total = 0
    for value in my_dict.values():
        total += value
    return total
```

Explanation: Iterate values directly; assumes all values are ints.

Sample calls:

```
print(sum_values({'a': 10, 'b': 20}))  # 30
print(sum_values({}))  # 0
```

Problem 11: Count Frequency of Characters

Function that takes a string and returns dict of char counts.

```
def char_frequency(s):
    freq = {}
    for char in s:
        freq[char] = freq.get(char, 0) + 1
    return freq
```

Explanation: Use `get()` to initialize count if missing. Common counting pattern.

Sample calls:

```
print(char_frequency('hello'))  # {'h': 1, 'e': 1, 'l': 2, 'o': 1}
print(char_frequency(''))  # {}
```

Problem 12: Invert Dictionary (Values to Keys)

Function to invert dict (assume unique values).

```
def invert_dict(my_dict):
    return {value: key for key, value in my_dict.items()}
```

Explanation: Dict comprehension with `items()` for swapping.

Sample calls:

```
print(invert_dict({'a': 1, 'b': 2}))  # {1: 'a', 2: 'b'}
print(invert_dict({}))  # {}
```

Problem 13: Find Max Value Key

Function to return key with maximum value.

```
def max_value_key(my_dict):
    if not my_dict:
        return None
    return max(my_dict, key=my_dict.get)
```

Explanation: `max()` with `key` arg uses `get()` to compare values.

Sample calls:

```
print(max_value_key({'a': 10, 'b': 20}))  # b
print(max_value_key({}))  # None
```

Problem 14: Nested Dict Access

Function to get value from nested dict: my_dict[‘outer’][‘inner’].

```
def get_nested_value(my_dict, outer, inner):
    return my_dict.get(outer, {}).get(inner, None)
```

Explanation: Chain `get()` to safely access nested; default to {} then None.

Sample calls:

```
nested = {'person': {'name': 'Alice'}}
print(get_nested_value(nested, 'person', 'name'))  # Alice
print(get_nested_value(nested, 'person', 'age'))  # None
```

Problem 15: Group List by Length

Function that takes list of strings, returns dict grouped by length.

```
def group_by_length(words):
    groups = {}
    for word in words:
        length = len(word)
        if length not in groups:
            groups[length] = []
        groups[length].append(word)
    return groups
```

Explanation: Use length as key, append to list value. Common grouping.

Sample calls:

```
print(group_by_length(['hi', 'hello', 'a']))
# {2: ['hi'], 5: ['hello'], 1: ['a']}
print(group_by_length([]))  # {}
```

Problem 16: Use setdefault for Lists

Function to add item to list in dict, create list if missing.

```
def add_to_list(my_dict, key, item):
    my_dict.setdefault(key, []).append(item)
    return my_dict
```

Explanation: `setdefault()` adds default list if key missing, then append.

Sample calls:

```
print(add_to_list({}, 'fruits', 'apple'))  # {'fruits': ['apple']}
print(add_to_list({'fruits': ['apple']}, 'fruits', 'banana'))
# {'fruits': ['apple', 'banana']}
```

Problem 17: Clear Dictionary

Function to clear a dict and return it.

```
def clear_dict(my_dict):
    my_dict.clear()
    return my_dict
```

Explanation: `clear()` removes all pairs in-place.

Sample calls:

```
print(clear_dict({'a': 1, 'b': 2}))  # {}
print(clear_dict({}))  # {}
```

Problem 18: Copy Dictionary

Function to return a shallow copy of dict.

```
def copy_dict(my_dict):
    return my_dict.copy()
```

Explanation: `copy()` creates new dict; changes to copy don't affect original.

Sample calls:

```
original = {'key': 'value'}
copied = copy_dict(original)
copied['key'] = 'new'
print(original)  # {'key': 'value'} (unchanged)
print(copied)  # {'key': 'new'}
```

Problem 19: Filter Dict by Value

Function to return new dict with values > threshold.

```
def filter_by_value(my_dict, threshold):
    return {k: v for k, v in my_dict.items() if v > threshold}
```

Explanation: Dict comprehension filters with condition.

Sample calls:

```
print(filter_by_value({'a': 1, 'b': 3}, 2))  # {'b': 3}
print(filter_by_value({}, 0))  # {}
```

Problem 20: Merge Multiple Dicts

Function that takes list of dicts, merges them (rightmost wins conflicts).

```
def merge_multiple(dicts_list):
    merged = {}
    for d in dicts_list:
        merged.update(d)
    return merged
```

Explanation: Iterate and `update()` each; later dicts overwrite.

Sample calls:

```
print(merge_multiple([{'a': 1}, {'b': 2}]))  # {'a': 1, 'b': 2}
print(merge_multiple([{'a': 1}, {'a': 3}]))  # {'a': 3}
```

Python Lists and Tuples:

Lists and tuples are fundamental Python data structures used to store collections of items. Both are sequence types, but they differ in mutability and use cases. This explanation covers their properties, basic manipulations, and common interview patterns, followed by 20 problems with solutions, explanations, and sample function calls to master the concepts for coding interviews.

---

1. Python Lists

- **Definition**: Lists are ordered, mutable collections of items (of any type). Defined with square brackets `[]`.
- **Key Properties**:
  - **Mutable**: You can change elements (add, remove, modify) after creation.
  - **Ordered**: Maintains insertion order; accessible by index (0-based).
  - **Heterogeneous**: Can store mixed types (e.g., `[1, "hello", 3.14]`).
  - **Dynamic**: Can grow or shrink via methods like `append`, `pop`, etc.
- **Common Uses**: Storing dynamic data, stacks/queues, iteration, and manipulation.

Basic List Operations

1. **Creation**:
   - Empty: `my_list = []`
   - With elements: `my_list = [1, 2, 3]` or `my_list = list(range(3))`
2. **Accessing**:
   - By index: `my_list[0]` → first element. Negative indices: `my_list[-1]` → last.
   - Slicing: `my_list[start:end:step]` (e.g., `my_list[1:3]` → `[2, 3]`).
3. **Modifying**:

- Update: `my_list[0] = 10`
- Append: `my_list.append(4)` → adds to end.
- Insert: `my_list.insert(index, value)` → at specific index.
- Extend: `my_list.extend([5, 6])` → adds multiple items.

4. **Removing**:
   - By index: `my_list.pop(index)` → removes and returns (default: last).
   - By value: `my_list.remove(value)` → removes first occurrence.
   - Clear: `my_list.clear()` → empties list.
5. **Other Methods**:
   - `len(my_list)`: Length.
   - `my_list.count(x)`: Count occurrences of `x`.
   - `my_list.index(x)`: First index of `x`.
   - `my_list.sort()`: Sorts in-place.
   - `my_list.reverse()`: Reverses in-place.
6. **Iteration**:
   - `for item in my_list:`
   - With index: `for i, item in enumerate(my_list):`
7. **List Comprehensions**:
   - Concise creation: `[x*2 for x in range(5)]` → `[0, 2, 4, 6, 8]`
   - Conditional: `[x for x in my_list if x > 0]`

Example:

```
lst = [1, 2, 3]
lst.append(4)           # [1, 2, 3, 4]
lst[1] = 20             # [1, 20, 3, 4]
print(lst[1:3])         # [20, 3]
lst.remove(20)          # [1, 3, 4]
print(len(lst))         # 3
```

---

2. Python Tuples

- **Definition**: Tuples are ordered, immutable collections. Defined with parentheses `()` or commas.
- **Key Properties**:

  - **Immutable**: Cannot change elements after creation (no append, insert, etc.).
  - **Ordered**: Maintains insertion order; indexable like lists.
  - **Heterogeneous**: Can store mixed types.
  - **Lightweight**: Faster and less memory than lists due to immutability.
- **Common Uses**: Fixed data, dictionary keys, function return values, iteration.

Basic Tuple Operations

1. **Creation**:
   - Empty: `my_tuple = ()` or `my_tuple = tuple()`
   - Single element: `my_tuple = (1,)` (comma required)
   - Multiple: `my_tuple = (1, 2, 3)` or `my_tuple = 1, 2, 3`
2. **Accessing**:
   - By index: `my_tuple[0]`, negative indices work.
   - Slicing: `my_tuple[1:3]`
3. **Operations**:
   - Concatenate: `(1, 2) + (3, 4)` → `(1, 2, 3, 4)`
   - Repeat: `(1, 2) * 2` → `(1, 2, 1, 2)`
   - Count: `my_tuple.count(x)`
   - Index: `my_tuple.index(x)`
4. **Iteration**:
   - Same as lists: `for item in my_tuple:`
5. **Unpacking**:
   - `a, b = my_tuple` assigns values.
   - Extended: `a, *rest, b = (1, 2, 3, 4)` → `a=1`, `rest=[2, 3]`, `b=4`.

Example:

```
tup = (1, 2, 3)
print(tup[0])          # 1
print(tup[1:])         # (2, 3)
print(tup + (4,))     # (1, 2, 3, 4)
a, b, c = tup          # a=1, b=2, c=3
print(tup.count(2))    # 1
```

Key Differences

Feature	List	Tuple
Syntax	`[]`	`()` or commas
Mutability	Mutable	Immutable
Performance	Slower (dynamic)	Faster (fixed)
Methods	Many (append, pop, etc.)	Few (count, index)
Use Case	Dynamic data	Fixed data, keys

Interview Tip: Use tuples for immutable data (e.g., coordinates, fixed records) or as dict keys. Use lists for mutable, dynamic collections.

---

20 Problems to Master Lists and Tuples

Below are 20 problems covering creation, access, modification (lists), iteration, and common patterns. Each includes a function, explanation, and sample calls with outputs. These are designed to be simple, fundamental, and aligned with coding interview questions.

Problem 1: Create a List from Range

Write a function to create a list of numbers from start to end (inclusive).

```
def create_list(start, end):
    return list(range(start, end + 1))
```

Explanation: `range(start, end+1)` generates numbers; `list()` converts to list.

Sample Calls:

```
print(create_list(1, 3))  # [1, 2, 3]
print(create_list(5, 5))  # [5]
```

Problem 2: Create a Tuple from List

Convert a list to a tuple.

```python
def list_to_tuple(lst):
    return tuple(lst)
```

Explanation: `tuple()` converts any iterable to a tuple.

Sample Calls:

```python
print(list_to_tuple([1, 2, 3]))  # (1, 2, 3)
print(list_to_tuple([]))         # ()
```

Problem 3: Access Middle Element

Return the middle element of a list (or first of two for even length).

```python
def middle_element(lst):
    if not lst:
        return None
    mid = len(lst) // 2
    return lst[mid]
```

Explanation: Use integer division to find middle index. Handle empty list.

Sample Calls:

```python
print(middle_element([1, 2, 3, 4]))  # 3
print(middle_element([1, 2]))        # 1
print(middle_element([]))             # None
```

Problem 4: Append to List

Add an item to the end of a list and return it.

```python
def append_item(lst, item):
    lst.append(item)
    return lst
```

Explanation: `append()` modifies in-place. Return for chaining.

Sample Calls:

```
print(append_item([1, 2], 3))  # [1, 2, 3]
print(append_item([], 1))      # [1]
```

Problem 5: Remove Last Element

Pop and return the last element of a list, or None if empty.

```
def pop_last(lst):
    return lst.pop() if lst else None
```

Explanation: `pop()` removes and returns last item; check for empty list.

Sample Calls:

```
print(pop_last([1, 2, 3]))  # 3 (list: [1, 2])
print(pop_last([]))         # None
```

Problem 6: Reverse a List

Reverse a list in-place and return it.

```
def reverse_list(lst):
    lst.reverse()
    return lst
```

Explanation: `reverse()` modifies in-place. Alternative: `lst[::-1]` for new list.

Sample Calls:

```
print(reverse_list([1, 2, 3]))  # [3, 2, 1]
print(reverse_list([]))          # []
```

Problem 7: Sort a List

Sort a list in ascending order and return it.

```
def sort_list(lst):
    lst.sort()
    return lst
```

Explanation: `sort()` modifies in-place. Use `sorted(lst)` for new list.

Sample Calls:

```
print(sort_list([3, 1, 2]))  # [1, 2, 3]
print(sort_list([]))         # []
```

Problem 8: Count Occurrences in List

Count how many times an item appears in a list.

```
def count_item(lst, item):
    return lst.count(item)
```

Explanation: `count()` returns occurrences of item.

Sample Calls:

```
print(count_item([1, 2, 1, 3], 1))  # 2
print(count_item([1, 2], 3))        # 0
```

Problem 9: Tuple Concatenation

Concatenate two tuples and return result.

```
def concat_tuples(tup1, tup2):
    return tup1 + tup2
```

Explanation: `+` creates new tuple by concatenating.

Sample Calls:

```
print(concat_tuples((1, 2), (3, 4)))  # (1, 2, 3, 4)
print(concat_tuples((), (1,)))        # (1,)
```

Problem 10: Check if List is Empty

Return True if list is empty, False otherwise.

```
def is_empty(lst):
    return len(lst) == 0
```

Explanation: Check length; `not lst` is equivalent.

Sample Calls:

```
print(is_empty([]))          # True
print(is_empty([1, 2]))     # False
```

Problem 11: List Comprehension Squares

Create a list of squares for numbers in a range.

```
def square_list(start, end):
    return [x * x for x in range(start, end + 1)]
```

Explanation: List comprehension generates squares efficiently.

Sample Calls:

```
print(square_list(1, 3))  # [1, 4, 9]
print(square_list(2, 2))  # [4]
```

Problem 12: Filter Even Numbers

Return a list of even numbers from input list.

```
def filter_evens(lst):
    return [x for x in lst if x % 2 == 0]
```

Explanation: Comprehension filters numbers divisible by 2.

Sample Calls:

```
print(filter_evens([1, 2, 3, 4]))  # [2, 4]
print(filter_evens([1, 3]))        # []
```

Problem 13: Tuple Unpacking

Return first and last elements of a tuple as a new tuple.

```
def first_last(tup):
    if not tup:
        return None
    return (tup[0], tup[-1])
```

Explanation: Use indexing; handle empty tuple.

Sample Calls:

```
print(first_last((1, 2, 3)))  # (1, 3)
print(first_last((1,)))       # (1, 1)
print(first_last(()))         # None
```

Problem 14: Remove Duplicates from List

Return a list with duplicates removed (keep first occurrence).

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

Explanation: Use set for O(1) lookup; maintain order with list.

Sample Calls:

```
print(remove_duplicates([1, 2, 2, 3]))  # [1, 2, 3]
print(remove_duplicates([]))            # []
```

Problem 15: Rotate List Left

Rotate list left by k positions.

```python
def rotate_left(lst, k):
    if not lst:
        return lst
    k = k % len(lst)  # Handle k > len
    return lst[k:] + lst[:k]
```

Explanation: Slice and concatenate; modulo handles large k.

Sample Calls:

```python
print(rotate_left([1, 2, 3, 4], 1))  # [2, 3, 4, 1]
print(rotate_left([1, 2], 3))        # [2, 1]
```

Problem 16: Find Index in Tuple

Find first index of item in tuple, or -1 if not found.

```python
def find_index(tup, item):
    return tup.index(item) if item in tup else -1
```

Explanation: `index()` returns first position; check existence first.

Sample Calls:

```python
print(find_index((1, 2, 3), 2))  # 1
print(find_index((1, 2), 4))     # -1
```

Problem 17: Merge Two Sorted Lists

Merge two sorted lists into a new sorted list.

```python
def merge_sorted(lst1, lst2):
    result = []
    i, j = 0, 0
    while i < len(lst1) and j < len(lst2):
        if lst1[i] <= lst2[j]:
            result.append(lst1[i])
            i += 1
        else:
            result.append(lst2[j])
            j += 1
```

```
    result.extend(lst1[i:])
    result.extend(lst2[j:])
    return result
```

Explanation: Compare and append smaller; add remaining elements.

Sample Calls:

```
print(merge_sorted([1, 3], [2, 4]))  # [1, 2, 3, 4]
print(merge_sorted([1], []))         # [1]
```

Problem 18: Split List into Chunks

Split a list into chunks of size n.

```
def split_chunks(lst, n):
    if n <= 0:
        return []
    return [lst[i:i + n] for i in range(0, len(lst), n)]
```

Explanation: Use slicing with step size n in comprehension.

Sample Calls:

```
print(split_chunks([1, 2, 3, 4], 2))  # [[1, 2], [3, 4]]
print(split_chunks([1, 2, 3], 5))     # [[1, 2, 3]]
```

Problem 19: Check if Tuple is Sorted

Return True if tuple is sorted in ascending order.

```
def is_sorted(tup):
    return all(tup[i] <= tup[i + 1] for i in range(len(tup) - 1))
```

Explanation: `all()` checks if each pair is in order; empty or single-element tuples are sorted.

Sample Calls:

```
print(is_sorted((1, 2, 3)))  # True
print(is_sorted((1, 3, 2)))  # False
print(is_sorted(()))         # True
```

Problem 20: Swap Elements in List

Swap elements at indices i and j in a list.

```
def swap_elements(lst, i, j):
    if 0 <= i < len(lst) and 0 <= j < len(lst):
        lst[i], lst[j] = lst[j], lst[i]
    return lst
```

Explanation: Use multiple assignment for swap; validate indices.

Sample Calls:

```
print(swap_elements([1, 2, 3], 0, 2))  # [3, 2, 1]
print(swap_elements([1, 2], 1, 5))     # [1, 2] (no swap, invalid index)
```

Tips for Coding Interviews

1. **Lists**:
   - Be cautious with index out-of-range errors; validate indices.
   - Use list comprehensions for concise code.
   - Know time complexities: `append` (O(1)), `pop` (O(1) from end, O(n) elsewhere), `sort` (O(n log n)).
   - Practice patterns: reverse, rotate, merge, filter, group.
2. **Tuples**:
   - Use for immutable data or as dict keys (e.g., `(x, y)` for coordinates).
   - Leverage unpacking for cleaner code.
   - Understand immutability limits (no modifications).
3. **Common Patterns**:
   - Two-pointer for merging or comparing.
   - Sliding window for chunks or sublists.
   - Set for deduplication (with lists).
   - Iteration with `enumerate` for index-value pairs.

Practice Strategy: - Run each problem's code and modify inputs to test edge cases (empty lists/tuples, single elements, duplicates). - Write test cases to verify correctness. - Simulate interview conditions: explain your logic aloud, handle edge cases, and optimize for clarity and efficiency.

These problems cover the basics and prepare you for common interview questions. If you want deeper challenges (e.g., advanced algorithms with lists/tuples), let me know!

Python List Comprehension and Dictionary Comprehension

List comprehensions and dictionary comprehensions are powerful, concise ways to create lists and dictionaries in Python. They are widely used in coding interviews for their elegance and efficiency in transforming, filtering, and generating data structures. Below, I'll explain both concepts in detail, covering syntax, use cases, and common patterns, followed by 20 problems (10 for list comprehensions, 10 for dictionary comprehensions) with solutions, explanations, and sample function calls to help you master them for coding interviews.

---

1. List Comprehension

- **Definition**: A list comprehension is a concise way to create a list by applying an expression to each item in an iterable, optionally with a condition. It replaces a `for` loop and is more readable.
- **Syntax**:

```
[expression for item in iterable if condition]
```

  - `expression`: The value or transformation for each item (e.g., `x*2`).
  - `item`: Variable representing each element in the iterable.
  - `iterable`: Source of elements (e.g., list, range, string).
  - `condition` (optional): Filters items (e.g., `x > 0`).

- **Key Properties**:
  - **Concise**: Reduces multi-line loops to one line.
  - **Readable**: Clear intent when used appropriately.
  - **Efficient**: Often faster than equivalent loops due to Python's internal optimization.
  - **Immutable**: Creates a new list; does not modify the source iterable.
- **Use Cases**:
  - Transform data (e.g., square numbers).
  - Filter elements (e.g., select evens).
  - Flatten or process nested structures (with nested comprehensions).

- **Example**:

```
# Square numbers from 1 to 5
squares = [x**2 for x in range(1, 6)]  # [1, 4, 9, 16, 25]
# Even numbers from a list
evens = [x for x in [1, 2, 3, 4] if x % 2 == 0]  # [2, 4]
```

Nested List Comprehension

- Used for nested loops or processing nested iterables (e.g., flattening a matrix).
- Syntax: `[expression for sublist in iterable for item in sublist]`
- Example:

```
matrix = [[1, 2], [3, 4]]
flattened = [num for row in matrix for num in row]  # [1, 2, 3, 4]
```

Limitations:

- Avoid overly complex comprehensions (e.g., multiple nested loops) for readability.
- No side effects (e.g., cannot modify external variables).

---

2. Dictionary Comprehension

- **Definition**: A dictionary comprehension creates a dictionary by generating key-value pairs from an iterable, optionally with a condition. It's similar to list comprehension but produces a dictionary.
- **Syntax**:

```
{key_expression: value_expression for item in iterable if condition}
```

  - `key_expression`: Generates the key.
  - `value_expression`: Generates the value.
  - `item`, `iterable`, `condition`: Same as list comprehension.

- **Key Properties**:

  - **Concise**: Replaces loops for dict creation.
  - **Flexible**: Can transform keys, values, or both.
  - **New dict**: Creates a new dictionary, doesn't modify existing ones.

- **Use Cases**:
  - Map values (e.g., string to length).
  - Filter key-value pairs (e.g., values above threshold).
  - Invert dictionaries or create from iterables.
- **Example**:

```
# Create dict of number to square
squares = {x: x**2 for x in range(1, 4)}  # {1: 1, 2: 4, 3: 9}
# Filter dict for values > 10
d = {'a': 5, 'b': 20, 'c': 15}
filtered = {k: v for k, v in d.items() if v > 10}  # {'b': 20, 'c': 15}
```

Nested Dictionary Comprehension:

- Less common but useful for nested dictionaries or complex mappings.
- Example:

```
# Create nested dict
nested = {x: {y: x*y for y in range(1, 3)} for x in range(1, 3)}
# {1: {1: 1, 2: 2}, 2: {1: 2, 2: 4}}
```

Limitations:

- Keys must be hashable (e.g., strings, numbers, tuples).
- Avoid complex logic to maintain readability.

Key Interview Patterns

- **List Comprehension**:
  - Filtering (e.g., select elements meeting a condition).
  - Mapping (e.g., transform each element).
  - Flattening nested lists or generating combinations.
- **Dictionary Comprehension**:
  - Counting frequencies (e.g., char counts in string).
  - Mapping values (e.g., word to length).
  - Filtering or transforming existing dictionaries.

- **Performance**: Both are optimized in Python, but avoid excessive nesting for clarity. Use sets or generators for memory efficiency if needed.

---

20 Problems to Master List and Dictionary Comprehensions

Below are 20 problems (10 list comprehensions, 10 dictionary comprehensions) with solutions, explanations, and sample calls. These are simple, fundamental problems tailored for coding interviews, covering creation, filtering, transformation, and common patterns.

List Comprehension Problems

Problem 1: Squares of Numbers

Create a list of squares for numbers from start to end (inclusive).

```
def square_numbers(start, end):
    return [x**2 for x in range(start, end + 1)]
```

Explanation: Use list comprehension to compute `x**2` for each number in range.

Sample Calls:

```
print(square_numbers(1, 3))  # [1, 4, 9]
print(square_numbers(2, 2))  # [4]
```

Problem 2: Filter Even Numbers

Return a list of even numbers from input list.

```
def filter_evens(lst):
    return [x for x in lst if x % 2 == 0]
```

Explanation: Filter with condition `x % 2 == 0`.

Sample Calls:

```
print(filter_evens([1, 2, 3, 4]))  # [2, 4]
print(filter_evens([1, 3]))        # []
```

Problem 3: Uppercase Strings

Convert all strings in a list to uppercase.

```
def to_uppercase(lst):
    return [s.upper() for s in lst]
```

Explanation: Apply `upper()` to each string in the list.

Sample Calls:

```
print(to_uppercase(['hello', 'world']))  # ['HELLO', 'WORLD']
print(to_uppercase([]))                  # []
```

Problem 4: Extract First Characters

Create a list of first characters from a list of strings.

```
def first_chars(strings):
    return [s[0] for s in strings if s]  # Avoid empty strings
```

Explanation: Use `s[0]` for first char; condition `if s` skips empty strings.

Sample Calls:

```
print(first_chars(['apple', 'banana', '']))  # ['a', 'b']
print(first_chars([]))                       # []
```

Problem 5: Flatten a Matrix

Flatten a 2D list into a 1D list.

```
def flatten_matrix(matrix):
    return [num for row in matrix for num in row]
```

Explanation: Nested comprehension iterates rows, then items in each row.

Sample Calls:

```
print(flatten_matrix([[1, 2], [3, 4]]))  # [1, 2, 3, 4]
print(flatten_matrix([]))                 # []
```

Problem 6: Filter Positive Numbers

Return a list of positive numbers from input list.

```
def positive_numbers(lst):
    return [x for x in lst if x > 0]
```

Explanation: Filter with condition `x > 0`.

Sample Calls:

```
print(positive_numbers([-1, 0, 1, 2]))  # [1, 2]
print(positive_numbers([-1, -2]))       # []
```

Problem 7: Double Every Second Element

Double every second element in a list (0-based index 1, 3, ...).

```
def double_second(lst):
    return [x * 2 if i % 2 == 1 else x for i, x in enumerate(lst)]
```

Explanation: Use `enumerate` to check index; double if index is odd.

Sample Calls:

```
print(double_second([1, 2, 3, 4]))  # [1, 4, 3, 8]
print(double_second([]))             # []
```

Problem 8: List of Lengths

Create a list of lengths of strings in input list.

```
def string_lengths(strings):
    return [len(s) for s in strings]
```

Explanation: Apply `len()` to each string.

Sample Calls:

```
print(string_lengths(['cat', 'dog']))  # [3, 3]
print(string_lengths([]))              # []
```

Problem 9: Filter Non-Empty Strings

Return a list of non-empty strings from input list.

```
def non_empty_strings(strings):
    return [s for s in strings if s]
```

Explanation: Condition `if s` filters out empty strings.

Sample Calls:

```
print(non_empty_strings(['', 'hello', '']))  # ['hello']
print(non_empty_strings([]))                 # []
```

Problem 10: Generate Pairs

Create a list of tuples (i, j) for i in range(1, n+1) and j in range(1, m+1).

```
def generate_pairs(n, m):
    return [(i, j) for i in range(1, n + 1) for j in range(1, m + 1)]
```

Explanation: Nested comprehension creates all pairs using two ranges.

Sample Calls:

```
print(generate_pairs(2, 2))  # [(1, 1), (1, 2), (2, 1), (2, 2)]
print(generate_pairs(1, 1))  # [(1, 1)]
```

Dictionary Comprehension Problems

Problem 11: Number to Square Dict

Create a dict mapping numbers to their squares (1 to n).

```
def number_to_square(n):
    return {x: x**2 for x in range(1, n + 1)}
```

Explanation: Map each number to its square using range.

Sample Calls:

```
print(number_to_square(3))  # {1: 1, 2: 4, 3: 9}
print(number_to_square(0))  # {}
```

Problem 12: Character Frequency

Create a dict of character frequencies from a string.

```
def char_frequency(s):
    return {c: s.count(c) for c in s}
```

Explanation: Use `count()` for each unique char; note this is less efficient than a loop with `get()` for large strings.

Sample Calls:

```
print(char_frequency('hello'))  # {'h': 1, 'e': 1, 'l': 2, 'o': 1}
print(char_frequency(''))       # {}
```

Problem 13: Filter Dict by Value

Filter a dictionary to keep key-value pairs where value > threshold.

```
def filter_by_value(d, threshold):
    return {k: v for k, v in d.items() if v > threshold}
```

Explanation: Iterate `items()`; keep pairs where value exceeds threshold.

Sample Calls:

```
print(filter_by_value({'a': 1, 'b': 5}, 2))  # {'b': 5}
print(filter_by_value({}, 0))                # {}
```

Problem 14: Invert Dictionary

Invert a dictionary (swap keys and values), assuming unique values.

```
def invert_dict(d):
    return {v: k for k, v in d.items()}
```

Explanation: Swap key-value pairs using `items()`.

Sample Calls:

```
print(invert_dict({'a': 1, 'b': 2}))  # {1: 'a', 2: 'b'}
print(invert_dict({}))                # {}
```

Problem 15: String to Length Dict

Create a dict mapping strings to their lengths.

```
def string_to_length(strings):
    return {s: len(s) for s in strings}
```

Explanation: Map each string to its length.

Sample Calls:

```
print(string_to_length(['cat', 'dog']))  # {'cat': 3, 'dog': 3}
print(string_to_length([]))              # {}
```

Problem 16: Dict from Two Lists

Create a dict from two lists (keys and values).

```
def dict_from_lists(keys, values):
    return {k: v for k, v in zip(keys, values)}
```

Explanation: `zip()` pairs keys and values; truncates to shorter list.

Sample Calls:

```
print(dict_from_lists(['a', 'b'], [1, 2]))  # {'a': 1, 'b': 2}
print(dict_from_lists([], []))              # {}
```

Problem 17: Double Values in Dict

Create a dict with values doubled from input dict.

```
def double_values(d):
    return {k: v * 2 for k, v in d.items()}
```

Explanation: Multiply each value by 2; keep keys unchanged.

Sample Calls:

```
print(double_values({'x': 1, 'y': 2}))  # {'x': 2, 'y': 4}
print(double_values({}))                 # {}
```

Problem 18: Filter Keys by Prefix

Keep key-value pairs where key starts with given prefix.

```
def filter_prefix(d, prefix):
    return {k: v for k, v in d.items() if k.startswith(prefix)}
```

Explanation: Use `startswith()` to filter keys.

Sample Calls:

```
print(filter_prefix({'apple': 1, 'banana': 2}, 'a'))  # {'apple': 1}
print(filter_prefix({'x': 1}, 'y'))                   # {}
```

Problem 19: Dict of Even Numbers

Create a dict of even numbers to their squares (1 to n).

```
def even_squares(n):
    return {x: x**2 for x in range(1, n + 1) if x % 2 == 0}
```

Explanation: Filter range to include only even numbers.

Sample Calls:

```
print(even_squares(4))  # {2: 4, 4: 16}
print(even_squares(1))  # {}
```

Problem 20: Nested Dict Creation

Create a dict where keys are numbers (1 to n) and values are dicts {1: num*1, 2: num*2}.

```
def nested_dict(n):
    return {x: {y: x*y for y in range(1, 3)} for x in range(1, n + 1)}
```

Explanation: Outer comprehension creates keys; inner creates nested dicts.

Sample Calls:

```
print(nested_dict(2))  # {1: {1: 1, 2: 2}, 2: {1: 2, 2: 4}}
print(nested_dict(0))  # {}
```

Tips for Coding Interviews

1. **List Comprehension**:
   - Use for simple transformations or filtering to impress with concise code.
   - Avoid overcomplicating (e.g., multiple nested loops); prefer loops for complex logic.
   - Common patterns: filtering (evens, positives), mapping (squares, uppercase), flattening.
   - Time complexity: O(n) for single loop, O(n*m) for nested.
2. **Dictionary Comprehension**:
   - Ideal for mapping (e.g., string to length), filtering, or inverting dicts.
   - Ensure keys are hashable; handle duplicate keys carefully (last value wins).
   - Common patterns: frequency counting, key-value transformations, filtering by condition.
   - Time complexity: Similar to list comprehension, depends on iterable size.
3. **Best Practices**:
   - Write readable comprehensions; avoid nesting beyond two levels.
   - Test edge cases: empty inputs, single elements, invalid data.
   - Combine with other structures (e.g., sets for unique items, tuples for immutability).
4. **Interview Strategy**:
   - Explain your comprehension step-by-step (e.g., "This filters even numbers, then squares them").
   - Compare with loop-based solution if asked for clarity.
   - Practice converting loops to comprehensions and vice versa.

Practice Strategy: - Run each problem and test with edge cases (empty inputs, duplicates, large inputs). - Rewrite each comprehension as a loop to understand the logic deeply. - Simulate interview conditions: write code on paper or whiteboard, explain aloud.

These problems cover the essentials and prepare you for common interview scenarios. If you want more advanced problems (e.g., nested comprehensions with complex logic or performance optimization), let me know!

Math Problems

To prepare for coding interviews, math-related problems are excellent for practicing list, tuple, and dictionary manipulations, as they often involve numerical computations, data transformations, and pattern recognition. Below, I'll provide a detailed explanation of how lists, tuples, and dictionaries are used in math problems, followed by 20 math-focused problems that leverage these data structures. Each problem includes a solution, detailed explanation, and sample function calls with expected outputs. These problems are designed to be simple yet cover common interview patterns, focusing on basic math operations and manipulations using lists, tuples, list comprehensions, and dictionary comprehensions.

Using Lists, Tuples, and Dictionaries in Math Problems

1. **Lists**:
    - **Use Case**: Store sequences of numbers (e.g., series, sequences, or arrays for calculations).
    - **Common Operations**: Append results, filter numbers, compute sums/products, or sort for ordering.
    - **Math Patterns**: Generating sequences (e.g., Fibonacci, primes), filtering (e.g., evens), or transforming (e.g., squares).
    - **Example**: `[x**2 for x in range(10)]` for squares, or `[x for x in lst if x % 2 == 0]` for even numbers.
2. **Tuples**:
    - **Use Case**: Store immutable numerical data (e.g., coordinates, fixed pairs like (numerator, denominator)).
    - **Common Operations**: Unpacking for calculations, returning multiple values, or using as dictionary keys.
    - **Math Patterns**: Represent points (e.g., `(x, y)`), store polynomial coefficients, or return multiple results (e.g., GCD and quotient).
    - **Example**: `(x, y) = (3, 4)` for Euclidean distance, or `tuple(range(5))` for fixed sequence.
3. **Dictionaries**:
    - **Use Case**: Map numbers to properties (e.g., number to frequency, value to index) or store results of computations.
    - **Common Operations**: Count frequencies, group numbers, or store intermediate results for dynamic programming.

- **Math Patterns**: Frequency of numbers, factor counts, or memoization for recursive math problems.
- **Example**: `{x: x**2 for x in range(5)}` for number-to-square mapping, or `{c: s.count(c) for c in s}` for digit counts.

4. **Comprehensions**:
   - **List Comprehension**: Generate sequences or filter numbers (e.g., `[x for x in range(100) if is_prime(x)]`).
   - **Dictionary Comprehension**: Map numbers to computed values or filter pairs (e.g., `{x: factorial(x) for x in range(5)}`).
   - **Math Relevance**: Concise for generating arithmetic sequences, filtering primes, or mapping to mathematical properties.

---

20 Math Problems with Solutions

These problems focus on math concepts (arithmetic, number theory, sequences, etc.) and use lists, tuples, and dictionaries with comprehensions where applicable. They are simple, fundamental, and aligned with coding interview questions. Each includes: - Problem statement - Solution as a Python function - Detailed explanation - Sample function calls with outputs

List-Based Math Problems

Problem 1: Sum of Even Numbers

Return the sum of even numbers in a list.

```
def sum_evens(lst):
    return sum([x for x in lst if x % 2 == 0])
```

Explanation: Use list comprehension to filter even numbers, then `sum()` to compute total. Empty lists return 0.

Sample Calls:

```
print(sum_evens([1, 2, 3, 4]))  # 6 (2 + 4)
print(sum_evens([1, 3, 5]))     # 0
```

Problem 2: Generate Arithmetic Sequence

Create a list of n terms starting at `start` with difference `d`.

```
def arithmetic_sequence(start, d, n):
    return [start + i * d for i in range(n)]
```

Explanation: List comprehension generates terms: `start`, `start + d`, `start + 2d`, ..., for `n` terms.

Sample Calls:

```
print(arithmetic_sequence(1, 2, 3))  # [1, 3, 5]
print(arithmetic_sequence(5, -1, 2)) # [5, 4]
```

Problem 3: Product of List Elements

Compute the product of all numbers in a list.

```
def list_product(lst):
    result = 1
    for x in lst:
        result *= x
    return result
```

Explanation: Iterate and multiply each element. Return 1 for empty list (identity for multiplication).

Sample Calls:

```
print(list_product([2, 3, 4]))  # 24
print(list_product([]))          # 1
```

Problem 4: Find Prime Numbers

Return a list of prime numbers up to n (inclusive).

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
```

```
            return False
    return True

def primes_up_to_n(n):
    return [x for x in range(2, n + 1) if is_prime(x)]
```

Explanation: `is_prime` checks divisibility up to square root. List comprehension filters primes.

Sample Calls:

```
print(primes_up_to_n(10))  # [2, 3, 5, 7]
print(primes_up_to_n(1))   # []
```

Problem 5: Fibonacci Sequence

Generate a list of first n Fibonacci numbers (starting with 0, 1).

```
def fibonacci(n):
    fib = [0, 1]
    while len(fib) < n:
        fib.append(fib[-1] + fib[-2])
    return fib[:n]
```

Explanation: Initialize with `[0, 1]`, append sum of last two numbers. Slice to handle n 2.

Sample Calls:

```
print(fibonacci(5))  # [0, 1, 1, 2, 3]
print(fibonacci(1))  # [0]
```

Problem 6: Sum of Squares

Return the sum of squares of numbers in a list.

```
def sum_of_squares(lst):
    return sum([x**2 for x in lst])
```

Explanation: Square each number with comprehension, then sum.

Sample Calls:

```
print(sum_of_squares([1, 2, 3]))  # 14 (1 + 4 + 9)
print(sum_of_squares([]))          # 0
```

Problem 7: Filter Multiples of k

Return a list of numbers from input list divisible by k.

```
def multiples_of_k(lst, k):
    return [x for x in lst if x % k == 0]
```

Explanation: Filter numbers where `x % k == 0`.

Sample Calls:

```
print(multiples_of_k([1, 2, 3, 4, 6], 2))  # [2, 4, 6]
print(multiples_of_k([1, 3, 5], 2))        # []
```

Problem 8: Maximum Difference

Find the maximum difference between any two elements in a list (max - min).

```
def max_difference(lst):
    return max(lst) - min(lst) if lst else 0
```

Explanation: Use `max()` and `min()`; handle empty list with 0.

Sample Calls:

```
print(max_difference([1, 5, 3]))  # 4 (5 - 1)
print(max_difference([]))          # 0
```

Problem 9: Generate Factorials

Create a list of factorials for numbers 0 to n.

```
def factorials(n):
    def factorial(x):
        if x == 0:
            return 1
        return x * factorial(x - 1)
    return [factorial(i) for i in range(n + 1)]
```

Explanation: Recursive `factorial` function; comprehension applies to range.

Sample Calls:

```
print(factorials(3))  # [1, 1, 2, 6]
print(factorials(0))  # [1]
```

Problem 10: Absolute Differences

Return a list of absolute differences between consecutive elements.

```
def abs_differences(lst):
    return [abs(lst[i + 1] - lst[i]) for i in range(len(lst) - 1)]
```

Explanation: Comprehension computes `|lst[i+1] - lst[i]|` for consecutive pairs.

Sample Calls:

```
print(abs_differences([1, 4, 2]))  # [3, 2]
print(abs_differences([1]))        # []
```

Tuple-Based Math Problems

Problem 11: Euclidean Distance

Compute the Euclidean distance between two points given as tuples (x1, y1) and (x2, y2).

```
def euclidean_distance(p1, p2):
    return ((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2) ** 0.5
```

Explanation: Use tuple indexing to compute `sqrt((x2-x1)^2 + (y2-y1)^2)`.

Sample Calls:

```
print(euclidean_distance((0, 0), (3, 4)))  # 5.0
print(euclidean_distance((1, 1), (1, 1)))  # 0.0
```

Problem 12: Tuple of Divisors

Return a tuple of all divisors of a number n.

```
def divisors(n):
    return tuple([i for i in range(1, n + 1) if n % i == 0])
```

Explanation: List comprehension finds divisors; convert to tuple.

Sample Calls:

```
print(divisors(6))   # (1, 2, 3, 6)
print(divisors(1))   # (1,)
```

Problem 13: Sum of Tuple Elements

Return the sum of all elements in a tuple.

```
def tuple_sum(tup):
    return sum(tup)
```

Explanation: `sum()` works directly on tuples; returns 0 for empty.

Sample Calls:

```
print(tuple_sum((1, 2, 3)))  # 6
print(tuple_sum(()))         # 0
```

Problem 14: Pairwise Product Tuple

Return a tuple of products of corresponding elements from two tuples.

```
def pairwise_product(tup1, tup2):
    return tuple(a * b for a, b in zip(tup1, tup2))
```

Explanation: `zip()` pairs elements; comprehension multiplies; convert to tuple.

Sample Calls:

```
print(pairwise_product((1, 2), (3, 4)))  # (3, 8)
print(pairwise_product((), ()))          # ()
```

Problem 15: GCD of Two Numbers

Return a tuple of (GCD, LCM) for two numbers using tuples.

```
def gcd_lcm(a, b):
    def gcd(x, y):
        while y:
            x, y = y, x % y
        return x
    g = gcd(a, b)
    return (g, (a * b) // g)
```

Explanation: Euclidean algorithm for GCD; LCM = (a * b) / GCD. Return as tuple.

Sample Calls:

```
print(gcd_lcm(12, 18))  # (6, 36)
print(gcd_lcm(5, 7))    # (1, 35)
```

Dictionary-Based Math Problems

Problem 16: Frequency of Digits

Return a dictionary of digit frequencies in a number.

```
def digit_frequency(n):
    return {int(d): str(n).count(d) for d in str(n)}
```

Explanation: Convert number to string; count each digit using comprehension.

Sample Calls:

```
print(digit_frequency(1223))  # {1: 1, 2: 2, 3: 1}
print(digit_frequency(0))     # {0: 1}
```

Problem 17: Number to Factors Dict

Create a dict mapping numbers 1 to n to their number of factors.

```
def factor_count(n):
    def count_factors(x):
        return sum(1 for i in range(1, x + 1) if x % i == 0)
    return {x: count_factors(x) for x in range(1, n + 1)}
```

Explanation: For each number, count divisors; map in dict comprehension.

Sample Calls:

```
print(factor_count(4))  # {1: 1, 2: 2, 3: 2, 4: 3}
print(factor_count(1))  # {1: 1}
```

Problem 18: Sum of Values by Parity

Return a dict with keys 'even' and 'odd' mapping to sum of even/odd numbers in a list.

```
def sum_by_parity(lst):
    return {
        'even': sum(x for x in lst if x % 2 == 0),
        'odd': sum(x for x in lst if x % 2 != 0)
    }
```

Explanation: Use comprehensions to sum even and odd numbers separately.

Sample Calls:

```
print(sum_by_parity([1, 2, 3, 4]))  # {'even': 6, 'odd': 4}
print(sum_by_parity([]))             # {'even': 0, 'odd': 0}
```

Problem 19: Map to Squares

Create a dict mapping numbers from a list to their squares, excluding duplicates.

```
def unique_squares(lst):
    return {x: x**2 for x in sorted(set(lst))}
```

Explanation: Convert to `set` to remove duplicates; sort for consistent order; map to squares.

Sample Calls:

```
print(unique_squares([1, 2, 2, 3]))  # {1: 1, 2: 4, 3: 9}
print(unique_squares([]))             # {}
```

Problem 20: Prime Factorization

Return a dict of prime factors and their counts for a number.

```
def prime_factorization(n):
    factors = {}
    d = 2
    while n > 1:
        while n % d == 0:
            factors[d] = factors.get(d, 0) + 1
            n //= d
        d += 1 if d == 2 else 2
        if d * d > n and n > 1:
            factors[n] = 1
            break
    return factors
```

Explanation: Divide by smallest prime (start with 2, then odd numbers); track counts in dict.

Sample Calls:

```
print(prime_factorization(12))   # {2: 2, 3: 1}
print(prime_factorization(7))    # {7: 1}
```

Tips for Coding Interviews

1. **Lists**:
   - Use list comprehensions for concise sequence generation or filtering (e.g., primes, squares).
   - Handle edge cases: empty lists, single elements, negative numbers.
   - Time complexity: O(n) for iteration, O(n log n) for sorting, O(n) for sum/product.
2. **Tuples**:
   - Use for immutable results (e.g., returning GCD and LCM).
   - Leverage unpacking for cleaner code in math problems (e.g., point coordinates).
   - Time complexity: O(1) for access, O(n) for iteration.
3. **Dictionaries**:
   - Ideal for counting (e.g., digit frequencies, factor counts).

- Use comprehensions for mapping numbers to properties (e.g., number to square).
- Time complexity: O(1) average for lookups/inserts, O(n) for iteration.

4. **Comprehensions**:

   - Keep them simple to maintain readability.
   - Use for generating sequences or mappings in one line.
   - Avoid for complex logic; use loops instead.

5. **Math Patterns**:

   - **Sequences**: Arithmetic, geometric, Fibonacci.
   - **Number Theory**: Primes, divisors, GCD/LCM, factorials.
   - **Aggregations**: Sum, product, max/min differences.
   - **Counting**: Frequencies, factor counts, grouping by property.

Practice Strategy: - Run each problem and test edge cases (empty inputs, negative numbers, large inputs). - Convert comprehensions to loops and vice versa to deepen understanding. - Simulate interview conditions: explain logic aloud, write on paper/whiteboard, and optimize for clarity. - Verify outputs for correctness using additional test cases.

These problems cover fundamental math concepts and data structure manipulations, preparing you for coding interviews. If you want more advanced math problems (e.g., modular arithmetic, matrix operations, or dynamic programming), let me know!

Armstrong Number Checker

Problem Statement

An Armstrong number is a number equal to the sum of its digits raised to the power of the number of digits. Implement a function to check if a given positive integer is an Armstrong number.

- Convert number to string for digit access.
- Compute sum of digits**power.
- Compare sum to original number.

Sample Input and Output

Input (`number`)	Output
`153`	`True`
`9474`	`True`
`9475`	`False`

Code with Detailed Comments

```python
def is_armstrong(number: int) -> bool:
    """
    Checks if number is an Armstrong number.
    Uses string conversion for digit iteration.
    """
    # Convert to string for digit count/access
    digits = str(number)
    # Get number of digits
    power = len(digits)
    # Compute sum of digit**power
    total = 0
    for digit in digits:
        digit_int = int(digit)
        total += digit_int ** power
    # Check equality
    return total == number
```

Key Insight: String conversion simplifies digit extraction without modulo operations.

Code Example Walkthrough

For `number = 153` (3 digits):

- `digits = '153'`, `power = 3`
- Sum: `1**3 + 5**3 + 3**3 = 1 + 125 + 27 = 153`
- `153 == 153` → `True`

For `9475` (4 digits):

- Sum: `9**4 + 4**4 + 7**4 + 5**4 = 6561 + 256 + 2401 + 625 = 9843`
- `9843 != 9475` → `False`

Time and Space Complexity

- **Time Complexity**: **O(d)**
  `d` = number of digits (log10(n) + 1).
- **Space Complexity**: **O(d)**
  Stores string of digits.

  Technique: Exponentiation and summation with string-based digit iteration.

Test Cases

```python
# Test Case 1
assert is_armstrong(153) == True

# Test Case 2
assert is_armstrong(9474) == True

# Test Case 3
assert is_armstrong(9475) == False

print("All test cases passed!")
```

Number Reversal

Problem Statement

Implement a function to reverse the digits of an integer, handling negative numbers by preserving the sign.

- Use modulo/division to extract digits.
- Build reversed number iteratively.
- Restore sign if negative.

Sample Input and Output

Input (n)	Output
12345	54321
-12345	-54321
100	1

Code with Detailed Comments

```python
def reverse_number(n: int) -> int:
    """
    Reverses digits of n, preserving sign.
    Handles negatives separately.
    """
    # Check if negative
    negative = n < 0
    if negative:
        n = -n
    # Initialize reversed
    reversed_num = 0
    while n > 0:
        # Get last digit
        digit = n % 10
        # Append to reversed
        reversed_num *= 10
        reversed_num += digit
        # Remove last digit
        n //= 10
    # Restore sign
    if negative:
        reversed_num = -reversed_num
    return reversed_num
```

Key Insight: Modulo extracts digits; multiplication shifts left for appending.

Code Example Walkthrough

For `n = 12345`:

- `negative = False`, `n = 12345`
- Loop:
  - digit=5, reversed=5, n=1234
  - digit=4, reversed=54, n=123
  - digit=3, reversed=543, n=12
  - digit=2, reversed=5432, n=1
  - digit=1, reversed=54321, n=0
- Result: `54321`

For `n = -123`:

- `negative = True`, `n = 123`
- Reversed: `321`
- Result: `-321`

Time and Space Complexity

- **Time Complexity**: **O(d)**
  `d` = number of digits.
- **Space Complexity**: **O(1)**
  Constant extra space.

  Technique: Iterative digit extraction with modulo and division.

Test Cases

```
# Test Case 1
assert reverse_number(12345) == 54321

# Test Case 2
assert reverse_number(-12345) == -54321
```

```python
# Test Case 3
assert reverse_number(100) == 1

print("All test cases passed!")
```

Factorial (Iterative)

Problem Statement

Compute factorial of non-negative integer `n` iteratively. Factorial is product of integers from 1 to `n`. Return string for negatives.

- Loop from 1 to `n`, multiplying.
- Handle base cases ($0!/1! = 1$).

Sample Input and Output

Input (`n`)	Output
5	120
0	1
-1	"Undefined for negative numbers"

Code with Detailed Comments

```python
def factorial_iterative(n: int) -> int | str:
    """
    Computes factorial iteratively.
    Returns string for negatives.
    """
    if n < 0:
        return "Undefined for negative numbers"
    # Initialize result
    result = 1
    # Multiply from 1 to n
```

```
    for i in range(1, n + 1):
        result *= i
    return result
```

Key Insight: Iterative multiplication avoids recursion depth issues for large n.

Code Example Walkthrough

For `n = 5`:

- `result = 1`
- i=1: `1*1=1`
- i=2: `1*2=2`
- i=3: `2*3=6`
- i=4: `6*4=24`
- i=5: `24*5=120`
- Result: `120`

For `n = -1`: Returns string message.

Time and Space Complexity

- **Time Complexity**: **O(n)**
  Loop runs n times.
- **Space Complexity**: **O(1)**
  Constant space.

  Technique: Simple loop accumulation.

Test Cases

```python
# Test Case 1
assert factorial_iterative(5) == 120

# Test Case 2
assert factorial_iterative(0) == 1

# Test Case 3
assert factorial_iterative(-1) == "Undefined for negative numbers"

print("All test cases passed!")
```

Factorial (Recursive)

Problem Statement

Compute factorial of non-negative integer `n` recursively. Factorial is product of integers from 1 to `n`. Return string for negatives.

- Base: 0 or 1 → 1
- Recursive: n * factorial(n-1)

Sample Input and Output

Input (`n`)	Output
5	120
0	1
-1	"Undefined for negative numbers"

Code with Detailed Comments

```python
def factorial_recursive(n: int) -> int | str:
    """
    Computes factorial recursively.
    Returns string for negatives.
    """
    if n < 0:
        return "Undefined for negative numbers"
    elif n == 0 or n == 1:
        return 1  # Base case
    else:
        # Recursive multiplication
        return n * factorial_recursive(n - 1)
```

Key Insight: Recursion mirrors mathematical definition but risks stack overflow for large n.

Code Example Walkthrough

For `n = 5`:

- 5 * factorial(4)
- 4 * factorial(3)
- 3 * factorial(2)
- 2 * factorial(1) → 2 * 1 = 2
- Unwind: 3*2=6, 4*6=24, 5*24=120

For `n = 0`: Base case → 1

Time and Space Complexity

- **Time Complexity**: **O(n)**
  n recursive calls.
- **Space Complexity**: **O(n)**
  Recursion stack depth n.

  Technique: Recursive decomposition.

Test Cases

```
# Test Case 1
assert factorial_recursive(5) == 120

# Test Case 2
assert factorial_recursive(0) == 1

# Test Case 3
assert factorial_recursive(-1) == "Undefined for negative numbers"

print("All test cases passed!")
```

Fibonacci (Iterative)

Problem Statement

Compute the nth Fibonacci number iteratively,
where F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

- Use loop to build sequence.
- Handle n <= 0 as 0.

Sample Input and Output

Input (`n`)	Output
10	55
0	0
1	1

Code with Detailed Comments

```python
def fibonacci_iterative(n: int) -> int:
    """
    Computes nth Fibonacci iteratively.
    Efficient with O(1) space.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    # Initialize first two
    a, b = 0, 1
    # Iterate to nth
    for i in range(2, n + 1):
        temp = a + b
        a = b
        b = temp
    return b
```

Key Insight: Two variables track last two numbers, avoiding array storage.

Code Example Walkthrough

For `n = 5`:

- a=0, b=1
- i=2: temp=1, a=1, b=1
- i=3: temp=2, a=1, b=2
- i=4: temp=3, a=2, b=3
- i=5: temp=5, a=3, b=5
- Result: 5

Time and Space Complexity

- **Time Complexity**: **O(n)**
  Loop runs n-1 times.

- **Space Complexity**: **O(1)**
  Constant variables.

  Technique: Iterative sequence building.

Test Cases

```
# Test Case 1
assert fibonacci_iterative(10) == 55

# Test Case 2
assert fibonacci_iterative(0) == 0

# Test Case 3
assert fibonacci_iterative(1) == 1

print("All test cases passed!")
```

Fibonacci (Recursive)

Problem Statement

Compute the nth Fibonacci number recursively,
where F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

- Base cases for 0 and 1.
- Recursive summation.

Sample Input and Output

Input (`n`)	Output
10	55
0	0
1	1

Code with Detailed Comments

```python
def fibonacci_recursive(n: int) -> int:
    """
    Computes nth Fibonacci recursively.
    Exponential time due to overlaps.
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    # Recursive calls
    prev1 = fibonacci_recursive(n - 1)
    prev2 = fibonacci_recursive(n - 2)
    return prev1 + prev2
```

Key Insight: Simple but inefficient due to redundant computations (tree recursion).

Code Example Walkthrough

For `n = 4`:

- $F(4) = F(3) + F(2)$
- $F(3) = F(2) + F(1) = (F(1)+F(0)) + 1 = (1+0)+1=2$
- $F(2) = F(1) + F(0) = 1 + 0 = 1$
- $F(4) = 2 + 1 = 3$

Many overlapping calls.

Time and Space Complexity

- **Time Complexity**: **O(2^n)**
  Exponential due to branching.
- **Space Complexity**: **O(n)**
  Recursion depth n.

  Technique: Recursive tree decomposition.

Test Cases

```
# Test Case 1
assert fibonacci_recursive(10) == 55

# Test Case 2
assert fibonacci_recursive(0) == 0

# Test Case 3
assert fibonacci_recursive(1) == 1

print("All test cases passed!")
```

Fibonacci (Golden Ratio)

Problem Statement

Approximate nth Fibonacci using Binet's formula:
F(n) (phi^n) / sqrt(5), where phi = (1 + sqrt(5))/2.
Round to nearest integer.

- Use math library for sqrt.
- Handles large n without iteration.

Sample Input and Output

Input (`n`)	Output
10	55
0	0
1	1

Code with Detailed Comments

```python
import math

def fibonacci_golden_ratio(n: int) -> int:
    """
    Approximates nth Fibonacci using Binet's formula.
    Rounds to nearest integer.
    """
    if n <= 0:
        return 0
    # Golden ratio
    phi = (1 + math.sqrt(5)) / 2
    # Binet's approximation
    approx = (phi ** n) / math.sqrt(5)
    # Round to int
    return round(approx)
```

Key Insight: Mathematical closed-form expression; accurate for n $<$ 70 due to floats.

Code Example Walkthrough

For `n = 10`:

- phi 1.6180339887
- phi^10 122.991869
- / sqrt(5) 122.991869 / 2.236 55.003
- round(55.003) = 55

Time and Space Complexity

- **Time Complexity**: **O(1)**
  Constant operations.
- **Space Complexity**: **O(1)**
  Constant space.

  Technique: Closed-form approximation with exponentiation.

Test Cases

```
# Test Case 1
assert fibonacci_golden_ratio(10) == 55

# Test Case 2
assert fibonacci_golden_ratio(0) == 0

# Test Case 3
assert fibonacci_golden_ratio(1) == 1

print("All test cases passed!")
```

Fibonacci (Golden Ratio Approx)

Problem Statement

Approximate nth Fibonacci using rough phi=1.618.
F(n) (phi^n) / sqrt(5). Round to nearest int.

- Similar to exact but uses approximate phi.
- Less accurate for large n.

Sample Input and Output

Input (`n`)	Output
10	55
0	0
1	1

Code with Detailed Comments

```python
import math

def fibonacci_golden_ratio_approx(n: int) -> int:
    """
    Approximates nth Fibonacci with rough phi.
    Rounds to nearest integer.
    """
    if n <= 0:
        return 0
    # Approx golden ratio
    phi_approx = 1.618
    # Approximation
    approx = (phi_approx ** n) / math.sqrt(5)
    # Round to int
    return round(approx)
```

Key Insight: Even rough phi gives good results for small n; deviates for larger.

Code Example Walkthrough

For `n = 10`:

- phi_approx=1.618
- 1.618^10 122.705
- / sqrt(5) 122.705 / 2.236 54.86
- round(54.86) = 55

Time and Space Complexity

- **Time Complexity**: **O(1)**
  Constant operations.
- **Space Complexity**: **O(1)**
  Constant space.

  Technique: Simplified closed-form with approximation.

Test Cases

```
# Test Case 1
assert fibonacci_golden_ratio_approx(10) == 55

# Test Case 2
assert fibonacci_golden_ratio_approx(0) == 0

# Test Case 3
assert fibonacci_golden_ratio_approx(1) == 1

print("All test cases passed!")
```

Pascal's Triangle Generator

Problem Statement

Given a positive integer `numRows`, generate the first `numRows` of **Pascal's Triangle**.

- Each row `r` (0-indexed) contains `r + 1` integers.
- The first and last elements of every row are always `1`.
- Any interior element is the sum of the two elements directly above it.
- **Alternative approach**: Each element in row `r` at position `k` equals the binomial coefficient
  $\mathbf{C(r,\ k) = r!\ /\ (k!\ \times\ (r-k)!)}$.

We'll use an **iterative combinatorial formula** to compute each row efficiently without factorials.

Sample Input and Output

Input (`numRows`)	Output
5	`[[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1]]`
1	`[[1]]`
3	`[[1], [1,1], [1,2,1]]`

Code with Detailed Comments

```python
class Solution:
    def generate(self, numRows: int) -> list[list[int]]:
        """
        Generates Pascal's Triangle up to numRows.
        Uses combinatorial method for each row.
        """
        # Build triangle by generating each row from 0 to numRows-1
        return [self.pascals_row(row) for row in range(numRows)]

    def pascals_row(self, row_index: int) -> list[int]:
        """
        Computes the row_index-th row (0-based) of Pascal's Triangle
        using iterative binomial coefficient calculation.
        """
        current = 1          # C(row, 0) = 1
        result = [1]         # First element is always 1

        # Compute C(row, k) for k = 1 to row_index
        for k in range(row_index):
            # Use recurrence: C(n, k+1) = C(n, k) * (n - k) // (k + 1)
            numerator = current * (row_index - k)
            denominator = k + 1
            current = numerator // denominator
            result.append(current)

        return result
```

Key Insight: Instead of computing factorials (which cause overflow and are slow), we use the **multiplicative recurrence** of binomial coefficients.

Code Example Walkthrough

Let's trace `row_index = 3` (which should produce `[1, 3, 3, 1]`):

- Start: `current = 1`, `result = [1]`
- Loop for `k = 0` to `2` (since `row_index = 3`):

- **k = 0**:
  `numerator = 1 * (3 - 0) = 3`
  `denominator = 1`
  `current = 3 // 1 = 3` → `result = [1, 3]`
- **k = 1**:
  `numerator = 3 * (3 - 1) = 6`
  `denominator = 2`
  `current = 6 // 2 = 3` → `result = [1, 3, 3]`
- **k = 2**:
  `numerator = 3 * (3 - 2) = 3`
  `denominator = 3`
  `current = 3 // 3 = 1` → `result = [1, 3, 3, 1]`

Final row: `[1, 3, 3, 1]`

Time and Space Complexity

- **Time Complexity**: $\mathbf{O(n^2)}$
  - There are `n` rows.
  - Row `i` has `i + 1` elements → total elements `n(n+1)/2` → $\mathbf{O(n^2)}$.
- **Space Complexity**: $\mathbf{O(n^2)}$
  - We store all `n(n+1)/2` elements in the output triangle.
  - No extra auxiliary space beyond output (computation uses O(1) extra per row).

  Technique: **Combinatorics with iterative binomial coefficient update** – avoids factorial computation and uses integer arithmetic safely.

Test Cases

```
sol = Solution()

# Test Case 1
assert sol.generate(1) == [[1]]

# Test Case 2
```

```python
assert sol.generate(3) == [[1], [1, 1], [1, 2, 1]]

# Test Case 3 (from prompt)
expected = [
    [1],
    [1, 1],
    [1, 2, 1],
    [1, 3, 3, 1],
    [1, 4, 6, 4, 1]
]
assert sol.generate(5) == expected

print("All test cases passed!")
```

Caesar Cipher

Problem Statement

Implement the **Caesar cipher**, a classic substitution cipher that shifts each alphabetic character in a message by a fixed number of positions in the alphabet.

- Only letters are shifted; case is preserved.
- Non-alphabetic characters (spaces, punctuation, digits) remain unchanged.
- The shift wraps around the alphabet (e.g., shifting `'z'` by 1 gives `'a'`).
- Support both **encryption** (positive shift) and **decryption** (negative shift).

The core technique relies on **modular arithmetic** to handle wrap-around cleanly.

Sample Input and Output

Input Text	Shift	Output (Encrypted)
`"Hello, World!"`	`3`	`"Khoor, Zruog!"`
`"abc"`	`1`	`"bcd"`
`"XYZ"`	`4`	`"BCD"`

Decryption reverses the process: - Input: `"Khoor, Zruog!"`, Shift: `3` → Output: `"Hello, World!"`

Code with Detailed Comments

```python
def caesar_cipher_encrypt(text: str, shift: int) -> str:
    """
    Encrypts text using Caesar cipher with given shift.
    Uses modular arithmetic to wrap letters within A-Z or a-z.
    """
    encrypted = ""
    for char in text:
        if char.isalpha():
            # Determine base ASCII value ('A' for upper, 'a' for lower)
            base = ord('A') if char.isupper() else ord('a')

            # Compute 0-based position in alphabet
            pos = ord(char) - base

            # Apply shift with modulo 26 to wrap around
            new_pos = (pos + shift) % 26

            # Handle negative modulo results (Python handles it,
            # but modulo ensures 0-25 range)
            encrypted_char = chr(new_pos + base)
            encrypted += encrypted_char
        else:
            # Keep non-alphabetic characters unchanged
            encrypted += char
    return encrypted


def caesar_cipher_decrypt(text: str, shift: int) -> str:
    """
    Decrypts Caesar-encrypted text by shifting backward.
    Reuses encrypt function with negative shift.
    """
    return caesar_cipher_encrypt(text, -shift)
```

Math Insight:
`(x + shift) % 26` ensures the result stays in `[0, 25]`,
which maps cleanly back to letters. Python's `%` always returns non-negative results.

Code Example Walkthrough

Trace encryption of `'Z'` with `shift = 4`:

1. `char = 'Z'` → alphabetic and uppercase.
2. `base = ord('A') = 65`
3. `pos = ord('Z') - 65 = 90 - 65 = 25`
4. `new_pos = (25 + 4) % 26 = 29 % 26 = 3`
5. `encrypted_char = chr(3 + 65) = chr(68) = 'D'`

`'Z'` → `'D'` (wraps from end to start of alphabet).

Now trace `'o'` in `"Hello"` with `shift = 3`: - `base = ord('a') = 97` - `pos = 111 - 97 = 14` - `new_pos = (14 + 3) % 26 = 17` - `chr(17 + 97) = chr(114) = 'r'`

So `'o'` → `'r'`.

Time and Space Complexity

- **Time Complexity**: **O(n)**
  Each character is processed once, where `n = len(text)`.
- **Space Complexity**: **O(n)**
  A new string of length `n` is built (strings are immutable in Python).

  Technique: **Modular arithmetic** enables cyclic shifting without conditional wrap logic.

Test Cases

```
# Test Case 1: Basic encryption and decryption
msg1 = "Hello, World!"
shift1 = 3
enc1 = caesar_cipher_encrypt(msg1, shift1)
dec1 = caesar_cipher_decrypt(enc1, shift1)
assert enc1 == "Khoor, Zruog!"
assert dec1 == msg1

# Test Case 2: Wrap-around and case preservation
msg2 = "XYZ abc"
```

```
shift2 = 4
enc2 = caesar_cipher_encrypt(msg2, shift2)
dec2 = caesar_cipher_decrypt(enc2, shift2)
assert enc2 == "BCD efg"
assert dec2 == msg2

print("All tests passed!")
```