Homework: Computational Complexity

Problem 1: Complexity Analysis

For each of the following code snippets, determine the time complexity and space complexity in Big-O notation. Provide a brief justification for each.

(a)

```
def func_a(data):
    n = len(data)
    total = 0
    for i in range(0, n, 2):
        for j in range(5):
            total += data[i]
    return total
```

- Time complexity: $O(n)$
- Space complexity: $O(1)$
- Justification: The outer loop runs $n/2$ times and the inner loop 5 times. Dropping constants simplifies $O(5n/2)$ to $O(n)$ time. Space is $O(1)$ because we only store a single variable (total).

(b)

```
def func_b(matrix):
    n = len(matrix)
    result = []
    for i in range(n):
        for j in range(i):
            result.append(matrix[i][j])
    return result
```

- Time Complexity: $O(n^2)$
- Space Complexity: $O(n^2)$
- Justification: The nested loops run $0 + 1 + \cdots + (n-1)$ times, totaling $\frac{n(n-1)}{2}$ iterations, or $O(n^2)$ time. Space is $O(n^2)$ because the result list stores all of those accessed elements.

(c)

```
def func_c(n):
    if n <= 1:
        return 1
    return func_c(n // 2) + func_c(n // 2)
```

- Time Complexity: $O(n)$
- Space Complexity: $O(logn)$
- Justification: The recursion creates a tree with $2n - 1$ total calls, resulting in $O(n)$ time. Space is $O(\log n)$ because that is the maximum depth of the call stack.

(d)

```
def func_d(items):
    seen = set()
    duplicates = []
    for item in items:
        if item in seen:
            duplicates.append(item)
        seen.add(item)
    return duplicates
```

- Time Complexity: $O(n)$
- Space Complexity: $O(n)$
- Justification: Iterating through n items takes $O(n)$ time because hash set lookups and insertions are $O(1)$ on average. Space is $O(n)$ because, in the worst case (no duplicates), the seen set stores every single item.

Problem 2: Choosing the Right Data Structure

Finding elements common to two collections is a frequent operation in data processing (for example, finding shared gene IDs across two experiments).

(a) The following function finds common elements using a naive approach. What is its time complexity if `list1` has n elements and `list2` has m elements? Explain why.

```
def find_common_naive(list1, list2):
    common = []
    for x in list1:
        for y in list2:
            if x == y and x not in common:
                common.append(x)
    return common
```

- Time Complexity: $O(n \cdot m)$
- Justification: The outer loop runs n times (for list1). Inside it, the inner loop runs m times (for list2). This nesting forces the core comparison to execute exactly $n \cdot m$ times. The not in common lookup only triggers on a match and doesn't exceed the dominant $O(n \cdot m)$ bound established by the loops.

(b) Write an efficient version called `find_common_fast` that uses a set to achieve better time complexity. What is the time complexity of your version? For example, `find_common_fast([1, 2, 3, 4], [3, 4, 5, 6])` should return `[3, 4]` (in any order).

```
def find_common_fast(list1, list2):
    return list(set(list1) & set(list2))
```

- Time Complexity: $O(n + m)$
- Justification: Converting both lists to sets takes $O(n)$ and $O(m)$ time respectively. The intersection operation runs in $O(min(n, m))$ time, which is dominated by the initial conversions. Thus, the overall time complexity is $O(n + m)$. The space complexity is also $O(n + m)$ due to the additional sets created.

(c) Verify the speedup empirically. Write a script that times both functions on lists of size n = 1000, 5000, 10000, and 20000 (where elements are random integers from 0 to n). Print the time for each function at each size.

```
import time
import random

def find_common_naive(list1, list2):
    common = []
    for x in list1:
        for y in list2:
            if x == y and x not in common:
                common.append(x)
    return common

def find_common_fast(list1, list2):
    return list(set(list1) & set(list2))

# Test sizes
sizes = [1000, 5000, 10000, 20000]

print(f"{'n':<8} | {'Naive Time (s)':<15} | {'Fast Time (s)':<15}")
print("-" * 44)

for n in sizes:
    # Generate random lists of size n with integers from 0 to n
    list1 = [random.randint(0, n) for _ in range(n)]
    list2 = [random.randint(0, n) for _ in range(n)]

    # Time the naive function
    start_time = time.time()
    find_common_naive(list1, list2)
    naive_time = time.time() - start_time

    # Time the fast function
    start_time = time.time()
    find_common_fast(list1, list2)
    fast_time = time.time() - start_time
```

```
    # Print results
    print(f"{n:<8} | {naive_time:<15.4f} | {fast_time:<15.6f}")
```

```
n        | Naive Time (s)  | Fast Time (s)
-----------------------------------------
1000     | 0.0282          | 0.000158
5000     | 0.5254          | 0.000917
10000    | 2.0812          | 0.001857
20000    | 9.5555          | 0.003279
```

Problem 3: Empirical Complexity Measurement

The following function performs a computation on a list:

```
def mystery_function(data):
    n = len(data)
    data = sorted(data)
    total = 0
    for i in range(n):
        left, right = 0, n - 1
        while left < right:
            if data[left] + data[right] == data[i]:
                total += 1
            if data[left] + data[right] < data[i]:
                left += 1
            else:
                right -= 1
    return total
```

Your task is to empirically determine the time complexity of `mystery_function` using the log-log method from the lecture.

(a) Write a timing script that measures the runtime of `mystery_function` for input sizes n = 500, 1000, 2000, 4000, and 8000. Use random integer data for each size. Run each size at least 3 times and take the average.

```
import time
import random

def mystery_function(data):
    n = len(data)
    data = sorted(data)
    total = 0
    for i in range(n):
        left, right = 0, n - 1
        while left < right:
            if data[left] + data[right] == data[i]:
                total += 1
            if data[left] + data[right] < data[i]:
                left += 1
            else:
                right -= 1
    return total

# Test configurations
sizes = [500, 1000, 2000, 4000, 8000]
trials = 3

print(f"{'n':<8} | {'Avg Time (s)':<15}")
print("-" * 26)

for n in sizes:
    total_time = 0
    for _ in range(trials):
        # Generate random integer data
        data = [random.randint(0, n * 10) for _ in range(n)]

        start_time = time.time()
        mystery_function(data)
        total_time += time.time() - start_time

    avg_time = total_time / trials
    print(f"{n:<8} | {avg_time:<15.6f}")
```

```
n        | Avg Time (s)
--------------------------
500      | 0.037330
1000     | 0.155143
2000     | 0.616604
4000     | 2.988187
8000     | 11.034911
```

(b) Compute the log-log slope using `scipy.stats.linregress` on the log of the sizes and the log of the times. Report the slope value.

```
import time
import random
import numpy as np
from scipy.stats import linregress

def mystery_function(data):
    # (Implementation from Part A)
    n = len(data)
    data = sorted(data)
    total = 0
    for i in range(n):
        left, right = 0, n - 1
        while left < right:
            if data[left] + data[right] == data[i]:
                total += 1
            if data[left] + data[right] < data[i]:
                left += 1
            else:
                right -= 1
    return total

# Run trials
sizes = [500, 1000, 2000, 4000, 8000]
trials = 3
avg_times = []

for n in sizes:
    total_time = 0
    for _ in range(trials):
        data = [random.randint(0, n * 10) for _ in range(n)]
        start_time = time.time()
        mystery_function(data)
        total_time += time.time() - start_time
    avg_times.append(total_time / trials)

# Compute log-log slope
log_sizes = np.log(sizes)
log_times = np.log(avg_times)

slope, intercept, r_value, p_value, std_err = linregress(log_sizes, log_times)
print(f"Calculated Slope: {slope:.3f}")
```

```
Calculated Slope: 2.107
```

(c) Based on the slope, what is the time complexity of `mystery_function`? Explain why this matches (or doesn't match) what you would expect from reading the code.

- Empirical Time Complexity: $O(n^2)$
- Justification: A log-log slope of ≈ 2 indicates quadratic growth. This perfectly matches the theoretical behavior of the code:

1. Sorting: sorted() takes $O(n \log n)$ time.
2. Outer Loop: The for i in range(n) loop runs n times.
3. Inner Loop: The while loop uses a two-pointer approach (left and right) that traverse the remaining array in $O(n)$ time.
4. Total: The nested loops give $O(n) \cdot O(n) = O(n^2)$, which dominates the initial $O(n \log n)$ sort, resulting in an overall time complexity of $O(n^2)$.

Problem 4: Improving a Statistical Computation

In Bayesian statistics and spatial statistics, you often need to solve many linear systems with the same coefficient matrix but different right-hand side vectors. For example, drawing samples from a multivariate normal or computing conditional distributions.

The following function solves m linear systems using a loop:

```
import numpy as np


def solve_systems_naive(A, B):
    """Solve A @ X = B for X, where B has m columns.

    Parameters
    ----------
```

```
    A : np.ndarray
        Symmetric positive definite matrix of shape (n, n).
    B : np.ndarray
        Matrix of shape (n, m), each column is a right-hand side vector.

    Returns
    -------
    np.ndarray
        Solution matrix X of shape (n, m).
    """
    n, m = B.shape
    X = np.zeros_like(B)
    for i in range(m):
        X[:, i] = np.linalg.solve(A, B[:, i])
    return X
```

(a) What is the time complexity of `solve_systems_naive` in terms of n and m? Explain what happens inside `np.linalg.solve` on each iteration.

- Time Complexity: $O(m \cdot n^3)$
- Justification: The function `np.linalg.solve` uses an algorithm (like LU decomposition) that has a time complexity of $O(n^3)$ for solving a single system of equations. Since this is done m times (once for each column of B), the overall time complexity is $O(m \cdot n^3)$. Each call to `np.linalg.solve` independently factors the matrix A and solves the system, leading to redundant computations when A is the same across all iterations.

(b) Rewrite the function as `solve_systems_cholesky` using `scipy.linalg.cholesky` and `scipy.linalg.cho_solve` to factor A once and then solve each system cheaply. What is the new time complexity? For example, `solve_systems_cholesky(np.array([[2, 1], [1, 2]]), np.array([[1, 0], [0, 1]]))` should return `np.array([[2/3, -1/3], [-1/3, 2/3]])` (the inverse of A, since B is the identity).

```
import scipy.linalg
import numpy as np

def solve_systems_cholesky(A, B):
    c = scipy.linalg.cholesky(A)
    return scipy.linalg.cho_solve((c, False), B)
```

- Justification: The new function first computes the Cholesky decomposition of A, which takes $O(n^3)$ time. Then, for each of the m right-hand side vectors, it solves the system using `cho_solve`, which takes $O(n^2)$ time per system. Therefore, the overall time complexity is $O(n^3 + m \cdot n^2)$. This is a significant improvement over the naive approach, especially when m is large, since the expensive factorization step is done only once.

(c) Time both approaches with n = 300 and m = 100, and report the speedup. Use a random symmetric positive definite matrix (for example, `A = Z @ Z.T + n * np.eye(n)` where `Z` is random).

```
import numpy as np
import scipy.linalg
import time

def solve_systems_naive(A, B):
    n, m = B.shape
    X = np.zeros_like(B)
    for i in range(m):
        X[:, i] = np.linalg.solve(A, B[:, i])
    return X

def solve_systems_cholesky(A, B):
    c = scipy.linalg.cholesky(A)
    return scipy.linalg.cho_solve((c, False), B)

# Setup dimensions
n = 300
m = 100

# Generate random symmetric positive definite matrix A and random B
np.random.seed(42)
Z = np.random.randn(n, n)
A = Z @ Z.T + n * np.eye(n)
B = np.random.randn(n, m)

# Time the naive approach
start_time = time.time()
solve_systems_naive(A, B)
naive_time = time.time() - start_time
```

```
# Time the Cholesky approach
start_time = time.time()
solve_systems_cholesky(A, B)
chol_time = time.time() - start_time

# Calculate speedup
speedup = naive_time / chol_time

print(f"Naive Time: {naive_time:.4f} s")
print(f"Cholesky Time: {chol_time:.4f} s")
print(f"Speedup: {speedup:.1f}x")
```

```
Naive Time: 0.1767 s
Cholesky Time: 0.0134 s
Speedup: 13.2x
```

Problem 5: Optimizing Pairwise Computation

The following function computes the sum of all pairwise absolute differences in an array:

$$S = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} |a_i - a_j|$$

```
def pairwise_abs_diff_slow(arr):
    """Compute sum of all pairwise absolute differences.

    Parameters
    ----------
    arr : list
        A list of numbers.

    Returns
    -------
    float
        Sum of |a_i - a_j| for all pairs i < j.

    Examples
    --------
    >>> pairwise_abs_diff_slow([1, 2, 4])
    6
    >>> pairwise_abs_diff_slow([2, 8, 4, 6])
    20
    """
    n = len(arr)
    total = 0
    for i in range(n):
        for j in range(i + 1, n):
            total += abs(arr[i] - arr[j])
    return total
```

This runs in O(n^2) time. Your task is to write a function `pairwise_abs_diff_fast` that computes the same result in O(n log n) time.

```
def pairwise_abs_diff_fast(arr):
    sorted_arr = sorted(arr)
    n = len(sorted_arr)
    total = 0
    for k in range(n):
        total += sorted_arr[k] * (2 * k - n + 1)
    return total
```

Example

```
import time
import random

def pairwise_abs_diff_slow(arr):
    n = len(arr)
    total = 0
    for i in range(n):
        for j in range(i + 1, n):
            total += abs(arr[i] - arr[j])
    return total

n = 5000
test_arr = [random.randint(1, 1000) for _ in range(n)]
```

```
# Time the slow version
start_time = time.time()
ans_slow = pairwise_abs_diff_slow(test_arr)
slow_time = time.time() - start_time

# Time the fast version
start_time = time.time()
ans_fast = pairwise_abs_diff_fast(test_arr)
fast_time = time.time() - start_time

# Print the results
print(f"Do the answers match? {ans_slow == ans_fast}")
print("-" * 30)
print(f"Slow O(n^2) time:      {slow_time:.4f} seconds")
print(f"Fast O(n log n) time:  {fast_time:.6f} seconds")
print(f"Speedup multiplier:    {slow_time / fast_time:.1f}x faster")
```

```
Do the answers match? True
------------------------------
Slow O(n^2) time:      1.5883 seconds
Fast O(n log n) time:  0.001606 seconds
Speedup multiplier:    989.2x faster
```

Hint: consider what happens when you sort the array first. After sorting, every element `a[k]` is greater than or equal to all elements before it. Think about how many times `a[k]` is added versus subtracted across all pairs that include index k.