DS5012: Foundations of Computer Science
Module 1 Homework: Analysis of Algorithms

Q1 [10 pts] For each function below, state its Big-O classification:

(a) $f(n) = 2n^2 + 3n + 1$

(b) $g(n) = n \log n + 5n$

(c) $h(n) = 3 \cdot 2^n + n^3$

Q2 [10 pts] True or False? "Big-O notation describes an *exact* growth rate." Justify your answer.

Q3 [10 pts] Analyze the following code and give its time complexity in Big-O form. Show your work.

```python
for i in range(n):
    for j in range(i):
        print(i, j)
```

Q4 [10 pts] Solve the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

using the Master Theorem. State which case applies and the resulting complexity.

Q5 [10 pts] Algorithm A runs in $100n \log n$ time, Algorithm B in n^2 time. Find the smallest integer n_0 such that for all $n \geq n_0$, Algorithm A is faster than Algorithm B, i.e.,

$$100n \log n < n^2.$$

Q6 [10 pts] Analyze the time complexity of this linear-regression function. Assume ‘X‘ is an $n \times d$ matrix (list of lists) and ‘y‘ is a vector (list) of length n.

Listing 1: Linear Regression

```python
def linear_regression_loops(X, y):
    n = len(X)       # Number of samples
    d = len(X[0])    # Number of features

    # 1. Transpose X: XT is d x n
    XT = [[0 for _ in range(n)] for _ in range(d)]
    for i in range(n):
        for j in range(d):
            XT[j][i] = X[i][j]

    # 2. Matrix Multiplication: XT * X -> XT_X (XT_X is d x d)
    XT_X = [[0 for _ in range(d)] for _ in range(d)]
    for i in range(d):
        for j in range(d):
            sum_val = 0
            for k in range(n):
                sum_val += XT[i][k] * X[k][j]
            XT_X[i][j] = sum_val

    # 3. Matrix Inversion: inv(XT_X) -> inv_XT_X (d x d)
    inv_XT_X = np.linalg.inv(XT_X)

    # 4. Compute intermediate term: XT * y -> XT_y (d x 1)
    XT_y = [0 for _ in range(d)]
    for i in range(d):       # Row of result (d)
        sum_val = 0
        for j in range(n):   # Common dimension (n)
             sum_val += XT[i][j] * y[j]
        XT_y[i] = sum_val

    # 5. Final Multiplication: inv_XT_X * XT_y -> w (d x 1)
    w = [0 for _ in range(d)]
    for i in range(d):
         sum_val = 0
         for j in range(d):
             sum_val += inv_XT_X[i][j] * XT_y[j] #
         w[i] = sum_val


     return w
```

Give the overall time complexity in terms of n (samples) and d (features), based on the explicit loops and the stated complexity for matrix inversion. Identify the dominant term(s).

Q7 [10 pts] Consider the following recursive function. Analyze its time complexity. Let n be the size of the input list ‘data‘.

Listing 2: Recursive Mystery Function

```
def recursive_mystery(data):
    n = len(data)
    # Base case
    if n <= 1:
        return 1

    # Recursive step: Divide into 3 parts (approx n/3)
    size_third = n // 3
    part1 = data[0 : size_third]
    part2 = data[size_third : 2 * size_third]
    part3 = data[2 * size_third : n]

    # Make 3 recursive calls
    res1 = recursive_mystery(part1)
    res2 = recursive_mystery(part2)
    res3 = recursive_mystery(part3)

    # Combine step:
    count = 0
    for i in range(n):
        # Performing some constant time work per element
        count += (i + res1 + res2 + res3) % 100

    return count
```

Set up a recurrence relation for the time complexity $T(n)$ of this function, considering the cost of slicing. Solve the recurrence relation (e.g., using the Master Theorem) and state the Big-O time complexity.

Q8 [10 pts] Amortized analysis: dynamic-array append. Consider the ‘append‘ method shown below.

Listing 3: Dynamic Array Append

```python
class DynamicArray:
    def __init__(self):
        self._capacity = 1
        self._n = 0 # number of actual elements
        self._A = [None] * self._capacity

    def _resize(self, new_capacity):
        # Create new array B
        B = [None] * new_capacity

        # Copy elements from A to B
        for i in range(self._n):
            B[i] = self._A[i]

        # Update array reference and capacity
        self._A = B
        self._capacity = new_capacity

    def append(self, element):
        if self._n == self._capacity:
            # Resize to double capacity when full
            new_capacity = 2 * self._capacity
            self._resize(new_capacity)

        # Append element (after potential resize)
        self._A[self._n] = element
        self._n += 1
```

Explain the difference between the *worst-case cost* of a single ‘append‘ operation and the *amortized cost* over a sequence of operations. Explain why the amortized cost is $O(1)$.