Question 1:

Prove that $f(n) = 9n^3 - 6n^2 + 8 = O(n^4)$

Proof:

According to the Big-O definition, $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 1$ such that:

$$|f(n)| \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

Let $f(n) = 9n^3 - 6n^2 + 8$ and $g(n) = n^4$. To prove $f(n) = O(n^4)$, we need to show that:

$$|9n^3 - 6n^2 + 8| \leq c \cdot n^4 \quad \text{for sufficiently large } n.$$

Divide $f(n)$ by n^4:

$$\frac{|9n^3 - 6n^2 + 8|}{n^4} = \frac{9}{n} - \frac{6}{n^2} + \frac{8}{n^4}.$$

For sufficiently large n, the terms $\frac{9}{n}$, $\frac{6}{n^2}$, and $\frac{8}{n^4}$ decrease as n grows.
Since $\frac{9}{n}$, $\frac{6}{n^2}$, and $\frac{8}{n^4}$ all approach 0 as $n \to \infty$, there exists an n_0 such that:

$$\frac{9}{n} - \frac{6}{n^2} + \frac{8}{n^4} \leq 1 \quad \text{for } n \geq n_0.$$

Thus, we can choose a constant $c > 0$ to satisfy the inequality:

$$|9n^3 - 6n^2 + 8| \leq c \cdot n^4.$$

For simplicity, let $c = 10$ and $n_0 = 2$. Then, for all $n \geq n_0$:

$$|9n^3 - 6n^2 + 8| \leq 10 \cdot n^4.$$

Thus, $f(n) = O(n^4)$ is proven.

Question 2:

Say a, b are two constants such that $a, b > 1$ and $a < b$. Does there exist an a, b pair such that $a^n = \Theta(b^n)$, e.g. $5^n = \Theta(9^n)$?

Solution:

By the definition of Θ-notation, $f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0.$$

Here, let $f(n) = a^n$ and $g(n) = b^n$. To prove whether $a^n = \Theta(b^n)$, we need to determine if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 1$ such that:

$$c_1 \cdot b^n \leq a^n \leq c_2 \cdot b^n \quad \text{for all } n \geq n_0.$$

Simplify $\frac{a^n}{b^n}$ as follows:

$$\frac{a^n}{b^n} = \left(\frac{a}{b}\right)^n.$$

Since $a < b$, we know that $\frac{a}{b} < 1$. Let $r = \frac{a}{b}$, where $0 < r < 1$. Thus:

$$\frac{a^n}{b^n} = r^n.$$

As $n \to \infty$, $r^n \to 0$ because $0 < r < 1$. This means that for sufficiently large n, a^n becomes arbitrarily smaller than b^n.

For $a^n = \Theta(b^n)$, there must exist constants $c_1, c_2 > 0$ such that:

$$c_1 \cdot b^n \leq a^n \quad \text{and} \quad a^n \leq c_2 \cdot b^n \quad \text{for all } n \geq n_0.$$

However, as shown above, $\frac{a^n}{b^n} = r^n \to 0$ as $n \to \infty$. This implies that no constant $c_1 > 0$ can satisfy $c_1 \cdot b^n \leq a^n$ for large n. Therefore, $a^n \neq \Theta(b^n)$.

For any $a, b > 1$ with $a < b$, it is impossible for $a^n = \Theta(b^n)$ because the exponential decay of $\left(\frac{a}{b}\right)^n$ ensures that a^n becomes negligibly small compared to b^n as $n \to \infty$.

Thus, no such pair (a, b) exists such that $a^n = \Theta(b^n)$. $\square$

Question 3:

Order the following functions from least to greatest growth rate such that, for any two functions f and g, if f comes before g, then $f(n) = O(g(n))$:

- $f_1(n) = n^n$
- $f_2(n) = n^{0.00001}$
- $f_3(n) = n^2$
- $f_4(n) = \lg(n)$
- $f_5(n) = 2^n$

We may assume the transitive property: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. This allows us to prove $f(n) = O(g(n))$ only for adjacent functions in the order.

Solution:

The growth rates of the functions from least to greatest are as follows:

$$f_4(n) = \lg(n), \quad f_2(n) = n^{0.00001}, \quad f_3(n) = n^2, \quad f_5(n) = 2^n, \quad f_1(n) = n^n.$$

Compare $f_4(n) = \lg(n)$ and $f_2(n) = n^{0.00001}$

We know that logarithmic functions grow slower than any positive power of n. Specifically:

$$\lg(n) = O(n^{0.00001}) \quad \text{and} \quad n^{0.00001} \neq O(\lg(n)).$$

Thus, $f_4(n)$ comes before $f_2(n)$.

Compare $f_2(n) = n^{0.00001}$ and $f_3(n) = n^2$

Since $n^{0.00001}$ is a smaller power of n than n^2, we have:

$$n^{0.00001} = O(n^2) \quad \text{and} \quad n^2 \neq O(n^{0.00001}).$$

Thus, $f_2(n)$ comes before $f_3(n)$.

Compare $f_3(n) = n^2$ and $f_5(n) = 2^n$

Exponential functions grow faster than any polynomial function. Specifically:

$$n^2 = O(2^n) \quad \text{and} \quad 2^n \neq O(n^2).$$

Thus, $f_3(n)$ comes before $f_5(n)$.

Compare $f_5(n) = 2^n$ and $f_1(n) = n^n$

The function n^n grows faster than 2^n because:

$$n^n = n \cdot n \cdot n \cdots (\text{n times}) \quad \text{and for large } n, \quad n^n \gg 2^n.$$

Thus:

$$2^n = O(n^n) \quad \text{and} \quad n^n \neq O(2^n).$$

Thus, $f_5(n)$ comes before $f_1(n)$.

The final order from least to greatest growth rate is:

$$f_4(n) = \lg(n), \quad f_2(n) = n^{0.00001}, \quad f_3(n) = n^2, \quad f_5(n) = 2^n, \quad f_1(n) = n^n.$$

□

Question 4:

Solve the following problems for the given code snippets

- **(a) foo1: Use summations**
- **(b) foo2: Use a recurrence**

Part (a): foo1 - Using Summations

Step 1: Write a summation for the runtime.

The outer `while` loop runs n times (from $a = 1$ to $a = n$). The inner `while` loop starts at $b = a$ and runs $n - a + 1$ times for each iteration of the outer loop. The total runtime is represented by the summation:

$$T(n) = \sum_{a=1}^{n} \sum_{b=a}^{n} C,$$

where C is the constant runtime of operations inside the loops.

Step 2: Simplify the summation.

$$T(n) = \sum_{a=1}^{n} \sum_{b=a}^{n} C = C \sum_{a=1}^{n} (n - a + 1).$$

Simplify $\sum_{a=1}^{n}(n - a + 1)$:

$$\sum_{a=1}^{n}(n - a + 1) = \sum_{a=1}^{n} n - \sum_{a=1}^{n} a + \sum_{a=1}^{n} 1.$$

Using known summation formulas:

$$\sum_{a=1}^{n} n = n^2, \quad \sum_{a=1}^{n} a = \frac{n(n+1)}{2}, \quad \sum_{a=1}^{n} 1 = n.$$

Substitute:

$$\sum_{a=1}^{n}(n - a + 1) = n^2 - \frac{n(n+1)}{2} + n.$$

Simplify:

$$\sum_{a=1}^{n}(n - a + 1) = \frac{n^2 + n}{2}.$$

Thus:

$$T(n) = C \cdot \frac{n^2 + n}{2}.$$

Step 3: Tight bound.

The runtime $T(n)$ simplifies to:

$$T(n) = \Theta(n^2).$$

Part (b): foo2 - Using a Recurrence

Step 1: Write the recurrence relation.

The base case is:

$$T(0) = C, \quad \text{where } C \text{ represents the constant runtime of the base case.}$$

The recursive case is:

$$T(n) = T(n-1) + \sum_{i=1}^{n} C,$$

where C is the constant runtime of the operations inside the loop.

Simplify the summation:

$$\sum_{i=1}^{n} C = C \cdot \frac{n(n+1)}{2}.$$

Thus, the recurrence becomes:

$$T(n) = T(n-1) + C \cdot \frac{n(n+1)}{2}.$$

Step 2: Solve the recurrence using the substitution method.

Expand the recurrence:

$$T(n) = T(n-1) + C \cdot \frac{n(n+1)}{2}.$$

$$T(n) = T(n-2) + C \cdot \frac{(n-1)n}{2} + C \cdot \frac{n(n+1)}{2}.$$

Continue expanding until the base case:

$$T(n) = T(0) + \sum_{k=1}^{n} C \cdot \frac{k(k+1)}{2}.$$

Substitute $T(0) = C$:

$$T(n) = C + C \cdot \sum_{k=1}^{n} \frac{k(k+1)}{2}.$$

Simplify the summation:

$$\sum_{k=1}^{n} \frac{k(k+1)}{2} = \frac{1}{2} \sum_{k=1}^{n} (k^2 + k).$$

Use summation formulas:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Substitute:

$$\sum_{k=1}^{n} \frac{k(k+1)}{2} = \frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right).$$

Simplify:

$$\sum_{k=1}^{n} \frac{k(k+1)}{2} = \frac{n(n+1)}{12}(2n+1+6) = \frac{n(n+1)(2n+7)}{12}.$$

Thus:

$$T(n) = C + C \cdot \frac{n(n+1)(2n+7)}{12}.$$

Step 3: Tight bound.
The leading term of $T(n)$ is $\frac{C \cdot 2n^3}{12}$, so:

$$T(n) = \Theta(n^3).$$

Question 5:

Part (a): Explanation of `mystery`

The algorithm `mystery` implements a sorting algorithm that works as follows: 1. It repeatedly identifies the largest element in the unsorted portion of the list. 2. Once the largest element is identified, it is swapped with the last element of the unsorted portion. 3. The unsorted portion of the list shrinks by one element after each iteration, and this process repeats until the entire list is sorted.

At a high level, this algorithm is similar to **selection sort**. It works by iteratively selecting the largest remaining element and placing it in its correct position, starting from the end of the list and working backward.

How it works: - The outer `while` loop (lines 7–13) iterates from the end of the list ($i = n - 1$) to the start ($i = 1$). - The inner `for` loop (lines 9–11) scans the unsorted portion of the list (indices 0 to i) to find the index of the largest element (x). - Once the largest element is identified, the `helper` function swaps it with the element at index i. - This ensures that the largest unsorted element is moved to its correct position, reducing the unsorted portion by one element in each iteration.

Part (b): Performance Analysis

Step 1: Runtime Analysis of `mystery`

The runtime of the algorithm depends on the number of iterations in the outer and inner loops: - The outer `while` loop runs $n - 1$ times (from $i = n - 1$ to $i = 1$). - The inner `for` loop runs $i + 1$ times for each value of i (from $j = 0$ to $j = i$).

The total number of comparisons made by the inner loop across all iterations is:

$$\sum_{i=1}^{n-1}(i+1) = \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \frac{(n-1)n}{2} + (n-1) = \frac{n^2-n}{2} + (n-1).$$

Simplifying:

$$\text{Total comparisons} = \frac{n^2+n-2}{2}.$$

The runtime of the `helper` function is constant ($O(1)$), so the runtime of `mystery` is determined by the number of comparisons. Thus:

$$T(n) = \Theta(n^2).$$

Step 2: Best and Worst Cases

- **Best Case:** Even if the input list is already sorted, the algorithm will still perform all comparisons in the inner loop and all swaps in the outer loop. Therefore, the best-case runtime is:

$$T_{\text{best}}(n) = \Theta(n^2).$$

- **Worst Case:** The runtime for the worst case (e.g., the list is sorted in reverse order) is also:

$$T_{\text{worst}}(n) = \Theta(n^2).$$

Thus, the algorithm has the same runtime for both the best and worst cases.

Step 3: Can the Algorithm Be Tweaked for Faster Best-Case Runtime?

To achieve asymptotically faster best-case performance, the algorithm must detect when the input list is already sorted. One possible tweak is: - Add a flag to check if any swaps occurred during the current iteration of the `while` loop. If no swaps occurred, the algorithm terminates early, as the list is already sorted.

With this modification: - The best-case runtime becomes $O(n)$, as the algorithm will detect a sorted list after one pass through the inner loop. - The worst-case runtime remains $\Theta(n^2)$, as all comparisons and swaps are still required in the general case.

Conclusion: - The original algorithm has the same asymptotic runtime ($\Theta(n^2)$) for both the best and worst cases. - A modification with an early exit condition can reduce the best-case runtime to $O(n)$, while maintaining $\Theta(n^2)$ runtime in the worst case.

Question 6:

Trimerge Sort: Analysis

Part (a): Number of Comparisons in the Merge Step

In merge sort, merging two subarrays of size $n/2$ takes $n-1$ comparisons because the merge step compares elements pairwise until all elements from both subarrays are placed in the sorted array.

For trimerge sort, the merge subroutine merges three sorted subarrays of size $n/3$. The merging process can be described as follows: - At each step, the smallest of the three current elements (one from each subarray) is selected and placed into the sorted array. - The process repeats until all n elements are merged.

Since n elements are placed into the sorted array, there are $n-1$ comparisons made across all elements during the merging process. This result arises because the merging process stops after the last element is added, requiring $n-1$ comparisons to resolve all relative orderings.

Conclusion: The trimerge step makes $n-1$ comparisons to merge three subarrays of size $n/3$.

Part (b): Tight Bound on Runtime Using the Recursion Tree Method

Trimerge Sort Recurrence:

The recurrence for trimerge sort is:

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n),$$

where: - $3T\left(\frac{n}{3}\right)$: The cost of recursively sorting three subarrays of size $n/3$. - $\Theta(n)$: The cost of merging three subarrays of size $n/3$.

Step 1: Value inside interior nodes.

At each level of the recursion tree, the cost of merging subarrays is $\Theta(n)$. Since there are three recursive calls per node, the total work at a given level of the tree is:

$$\text{Work per level} = \Theta(n).$$

Step 2: Value inside leaf nodes.

The recursion stops when the size of the subarray is less than 3. At this point, the cost is constant ($\Theta(1)$).

Step 3: Number of child nodes per node.

Each node has 3 child nodes because the array is split into three parts at each step.

Step 4: Height of the recursion tree.

At each recursive step, the input size is divided by 3. The height of the recursion tree is:

$$h = \log_3(n).$$

Step 5: Total work.

The total work is the sum of the work across all levels of the tree. Since there are $\log_3(n)$ levels and the work at each level is $\Theta(n)$, the total work is:

$$T(n) = \Theta(n) \cdot \log_3(n).$$

Conclusion: The runtime of trimerge sort is $T(n) = \Theta(n \log_3(n))$.

Part (c): Is Trimerge Sort Asymptotically Faster Than Merge Sort?

The runtime of merge sort is $\Theta(n \log_2(n))$, while the runtime of trimerge sort is $\Theta(n \log_3(n))$.

The logarithmic base change formula states that:

$$\log_3(n) = \frac{\log_2(n)}{\log_2(3)}.$$

Thus:

$$n \log_3(n) = n \cdot \frac{\log_2(n)}{\log_2(3)}.$$

Since $\log_2(3) > 1$, the runtime of trimerge sort is larger than that of merge sort for sufficiently large n. Therefore, trimerge sort is **not** asymptotically faster than merge sort.

Conclusion: Trimerge sort is not asymptotically faster than merge sort.