# PROBLEMS AND SOLUTIONS — PART 3

*Chapter 2: Principles of Programming and Analysis of Algorithms*

*More Important Exam-Oriented Numerical and Analytical Problems with Step-by-Step Solutions*

This is a third set of frequently asked numerical and analytical problems on Algorithms, Design Approaches, Complexity, Big O Notation, and Algorithm Analysis, covering problem types not included in Part 1 or Part 2 — including Omega and Theta proofs, the Master Theorem, Tower of Hanoi, and amortized analysis. Practice all three parts together for complete exam coverage of this chapter.

## Section A: More Loop-Based Time Complexity

---

**Problem 1:**

**Find the time complexity of the following code:**

```
i = n;
while (i >= 1) {
    printf("%d", i);
    i = i / 2;
}
```

**Solution:**

Here i starts at n and is halved (integer division) at every step: n, n/2, n/4, n/8, ... until it becomes less than 1. If the loop runs k times, then n / 2^k ≈ 1, which gives k = $\log_2(n)$.

```
Time Complexity = O(log n)
```

---

**Problem 2:**

**Find the time complexity of the following code:**

```
i = 1;
while (i * i <= n) {
    printf("%d", i);
    i++;
}
```

**Solution:**

The loop continues as long as $i^2 \leq n$, i.e., $i \leq \sqrt{n}$. Since i increases by 1 each time starting from 1, the loop runs approximately $\sqrt{n}$ times before the condition fails.

```
Time Complexity = O(√n)
```

---

**Problem 3:**

**Find the time complexity of the following triple nested loop:**

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            printf("%d", i+j+k);
        }
    }
}
```

**Solution:**

Each of the three loops runs n times and all three are nested inside one another and depend on the same n. The innermost statement therefore executes $n \times n \times n = n^3$ times.

```
Time Complexity = O(n³)
```

---

**Problem 4:**

**Find the time complexity of the following code, where c is a fixed constant (e.g., c = 10) that does not depend on n:**

```
for (i = 0; i < n; i++) {
    for (j = 0; j < c; j++) {
        printf("%d", j);
    }
}
```

**Solution:**

The outer loop runs n times, but the inner loop always runs exactly c times regardless of n, since c is a fixed constant, not a variable that grows with input size. So the total number of iterations is $n \times c$.

Since c is a constant, it is dropped in Big O notation (only growth with respect to n matters).

```
Time Complexity = O(n)
```

This shows that a nested loop is not automatically $O(n^2)$ — it depends on whether the inner loop's bound actually grows with n.

## Section B: Big Omega and Big Theta Proofs

---

**Problem 5: Prove using the definition of Big Omega notation that $f(n) = 3n^2 + 2n$ is $\Omega(n^2)$.**

**Solution:**

By definition, f(n) is $\Omega(g(n))$ if there exist positive constants c and $n_0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Here $f(n) = 3n^2 + 2n$ and we take $g(n) = n^2$. Since $2n \geq 0$ for all $n \geq 0$, we have $f(n) = 3n^2 + 2n \geq 3n^2$ for all $n \geq 0$.

Choosing $c = 3$ and $n_0 = 0$ satisfies the definition, since $f(n) \geq 3 \cdot n^2$ for all $n \geq 0$.

```
Hence, f(n) = 3n² + 2n is Ω(n²).
```

---

**Problem 6: Prove that $f(n) = 5n^2 + 2n + 1$ is $\Theta(n^2)$.**

**Solution:**

To prove $f(n) = \Theta(n^2)$, we must show both $f(n) = O(n^2)$ (upper bound) and $f(n) = \Omega(n^2)$ (lower bound), using the same $g(n) = n^2$.

Upper bound: For $n \geq 1$, $5n^2 + 2n + 1 \leq 5n^2 + 2n^2 + n^2 = 8n^2$ (since $2n \leq 2n^2$ and $1 \leq n^2$ for $n \geq 1$). So $f(n) \leq 8n^2$, giving $f(n) = O(n^2)$ with $c_1 = 8$, $n_0 = 1$.

Lower bound: For $n \geq 0$, $5n^2 + 2n + 1 \geq 5n^2$. So $f(n) \geq 5n^2$, giving $f(n) = \Omega(n^2)$ with $c_2 = 5$, $n_0 = 0$.

Since $c_2 \cdot n^2 \leq f(n) \leq c_1 \cdot n^2$ holds for $n \geq 1$ (with $c_1 = 8$, $c_2 = 5$),

$$f(n) = \Theta(n^2)$$

---

**Problem 7: Show that $n^2 = O(n^3)$, but $n^3$ is NOT $O(n^2)$. What does this tell us about the Big O relation?**

**Solution:**

$n^2 = O(n^3)$: For all $n \geq 1$, $n^2 \leq n^3$ (since $n \geq 1$ implies multiplying both sides of $n^2$ by n only increases it or keeps it equal). Choosing $c = 1$ and $n_0 = 1$ satisfies the definition, so $n^2 = O(n^3)$ is true.

$n^3$ is NOT $O(n^2)$: Suppose $n^3 \leq c \cdot n^2$ for some constant c and all $n \geq n_0$. Dividing both sides by $n^2$ gives $n \leq c$. But n grows without bound while c is fixed, so this inequality must eventually fail for large n. Hence no such c and $n_0$ can exist, and $n^3$ is NOT $O(n^2)$.

This shows that the Big O relation is not symmetric: a function with a smaller growth rate is always O of a function with a larger or equal growth rate, but not the other way around. This is analogous to how $5 \leq 10$ does not imply $10 \leq 5$.

## Section C: Recurrence Relations and the Master Theorem

---

**Problem 8: The Tower of Hanoi problem for n disks has the recurrence $T(n) = 2T(n-1) + 1$, with $T(1) = 1$. Solve this recurrence and find the time complexity.**

**Solution:**

Expanding the recurrence step by step: $T(n) = 2T(n-1) + 1 = 2[2T(n-2)+1] + 1 = 4T(n-2) + 2 + 1 = 4[2T(n-3)+1] + 3 = 8T(n-3) + 4 + 3$.

In general, after k expansions: $T(n) = 2^k \cdot T(n-k) + (2^k - 1)$. When $k = n-1$, $T(n-k) = T(1) = 1$, giving:

$$T(n) = 2^{n-1} \cdot 1 + (2^{n-1} - 1) = 2^n - 1$$

So the minimum number of moves required to solve Tower of Hanoi for n disks is $2^n - 1$, and the time complexity is:

$$T(n) = O(2^n)$$

---

**Problem 9: Solve the recurrence $T(n) = T(n/2) + 1$, with $T(1) = 1$, using the substitution method (this is the recurrence for Binary Search).**

**Solution:**

Let $n = 2^k$, so that $k = \log_2(n)$. Substituting repeatedly: $T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 1 + 1 = \ldots = T(2^0) + k = T(1) + k = 1 + k$.

Substituting back $k = \log_2(n)$:

$$T(n) = 1 + \log_2(n) = O(\log n)$$

---

**Problem 10: Apply the Master Theorem to solve the recurrence $T(n) = 2T(n/2) + n^2$.**

**Solution:**

The Master Theorem applies to recurrences of the form $T(n) = aT(n/b) + f(n)$. Here $a = 2$, $b = 2$, and $f(n) = n^2$.

Compute $n^{(\log_b a)} = n^{(\log_2 2)} = n^1 = n$. Compare this with $f(n) = n^2$.

Since $f(n) = n^2$ grows strictly faster than $n^{(\log_b a)} = n^1$ (this is Case 3 of the Master Theorem, where $f(n)$ dominates), and the regularity condition holds, the solution is dominated by $f(n)$ itself.

$$T(n) = O(n^2)$$

## Section D: Design Approach Applications

---

**Problem 11: Using Dynamic Programming, find the length of the Longest Common Subsequence (LCS) between the strings X = "ABCBDAB" and Y = "BDCABA".**

**Solution:**

The LCS is found by building a table dp[i][j] representing the LCS length of the first i characters of X and the first j characters of Y, using the rule: if X[i] = Y[j], dp[i][j] = dp[i-1][j-1] + 1; otherwise, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).

Applying this rule across the full table for X = "ABCBDAB" (length 7) and Y = "BDCABA" (length 6), the resulting longest common subsequence is "BCBA" (or an equally long alternative such as "BDAB"), which has length 4.

```
LCS Length = 4
```

Since the table has $7 \times 6$ entries, each computed in $O(1)$ time, the overall time complexity of the Dynamic Programming solution is $O(m \times n)$, where m and n are the lengths of the two strings.

---

**Problem 12: Explain why Quick Sort has a worst-case time complexity of $O(n^2)$ but an average-case time complexity of $O(n \log n)$. Under what condition does the worst case occur?**

**Solution:**

Quick Sort (a Divide and Conquer algorithm) works by selecting a pivot element and partitioning the array into elements smaller and larger than the pivot, then recursively sorting each part.

If the pivot chosen at each step happens to split the array into two roughly equal halves, the recurrence becomes $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$ — this is the average case (and best case).

However, if the pivot chosen is always the smallest or largest element (which happens, for example, when the array is already sorted and the first or last element is always picked as the pivot), one partition ends up empty and the other has n-1 elements. The recurrence becomes $T(n) = T(n-1) + n$, which solves to $O(n^2)$ — this is the worst case.

This is why randomized pivot selection or median-of-three pivot selection is often used in practice, to make the worst case highly unlikely.

---

**Problem 13: Explain, in brief, how the Branch and Bound technique reduces the search space compared to a pure Brute Force approach when solving the 0/1 Knapsack Problem.**

**Solution:**

A pure Brute Force approach to the 0/1 Knapsack Problem examines every possible subset of items ($2^n$ subsets in total) and checks which ones fit within the capacity, keeping the one with maximum value. This is highly inefficient for large n.

Branch and Bound instead explores the same solution space in the form of a decision tree (branching on whether each item is included or excluded), but at every node it calculates an upper bound (often using the fractional knapsack value) on the best possible value obtainable from that node onward.

If this upper bound is not better than the best solution found so far, that entire branch is discarded (pruned) without being explored further. This pruning significantly reduces the number of nodes actually visited compared to brute force, while still guaranteeing the optimal solution is found.

## Section E: Space Complexity and Amortized Analysis

---

**Problem 14:**

**Compare the space complexity of the following iterative and recursive functions, both of which compute the sum of the first n natural numbers.**

```
// Iterative version
int sumIter(int n) {
    int sum = 0;
    for (i = 1; i <= n; i++) sum += i;
    return sum;
}

// Recursive version
int sumRec(int n) {
    if (n == 0) return 0;
    return n + sumRec(n - 1);
}
```

**Solution:**

The iterative version uses only a fixed number of variables (sum, i) regardless of n, so its auxiliary space complexity is O(1).

The recursive version, however, makes n nested recursive calls before reaching the base case, and each call adds a new frame to the call stack. So its auxiliary space complexity is O(n).

Although both versions have the same time complexity, O(n), the iterative version is more space-efficient. This illustrates that two algorithms with identical time complexity can still differ significantly in space complexity.

---

**Problem 15: A dynamic array starts with capacity 1 and doubles its capacity (creating a new array and copying all elements) whenever it becomes full, each time an element is inserted. Show that the amortized time complexity per insertion, over n insertions, is O(1), even though individual insertions can take O(n) time.**

**Solution:**

Doubling happens at insertions 1, 2, 4, 8, 16, ..., i.e., whenever the array size reaches a power of 2. At the doubling step where the array grows from size k to 2k, the copying operation costs O(k).

Over n insertions, the total cost of all copying operations is the sum $1 + 2 + 4 + 8 + ... + n$ (a geometric series), which sums to approximately 2n. Adding the cost of the n individual O(1) insertions themselves gives a total cost of about $n + 2n = 3n$ for all n insertions.

Dividing the total cost (3n) by the number of insertions (n) gives an amortized cost of $3n / n = 3$ per insertion, which is a constant.

```
Amortized Time Complexity per insertion = O(1)
```

This shows that although a single insertion that triggers doubling costs O(n) in the worst case, the average (amortized) cost per insertion across the whole sequence of operations remains O(1).