# PROBLEMS AND SOLUTIONS

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

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

This document contains frequently asked numerical and analytical problems, along with fully worked solutions, based on the topics of Algorithms, Design Approaches, Complexity, Big O Notation, and Algorithm Analysis. These problem types are commonly asked in semester examinations and should be practiced thoroughly.

## Section A: Time Complexity of Code Snippets

---

**Problem 1:**

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

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

**Solution:**

The loop runs from i = 0 to i = n - 1, so the statement inside the loop executes exactly n times. Since there is a single loop that depends linearly on n, the time complexity is:

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

---

**Problem 2:**

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

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

**Solution:**

The outer loop runs n times. For each iteration of the outer loop, the inner loop also runs n times. So the total number of times the inner statement executes is $n \times n = n^2$.

```
Time Complexity = O(n²)
```

---

**Problem 3:**

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

```
for (i = 1; i < n; i = i * 2) {
    printf("%d", i);
```

```
}
```

**Solution:**

Here, i starts at 1 and doubles each time (1, 2, 4, 8, ... 2^k). The loop continues until $2^k \geq n$, i.e., $k = \log_2(n)$. So the loop runs approximately $\log_2(n)$ times.

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

---

**Problem 4:**

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

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

**Solution:**

For each value of i, the inner loop runs i times. So the total number of iterations is the sum 0 + 1 + 2 + ... + (n-1) = n(n-1)/2, which simplifies to $(n^2 - n)/2$. Dropping the lower-order term and the constant factor as per Big O rules:

```
Time Complexity = O(n²)
```

---

**Problem 5:**

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

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

**Solution:**

These are two separate (sequential, not nested) loops, each running n times. Since they are not nested, their complexities are added, not multiplied: n + n = 2n. Dropping the constant factor 2:

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

## Section B: Big O Comparison and Growth Rate

---

**Problem 6: Arrange the following time complexities in increasing order of growth rate: $O(n^2)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(1)$, $O(2^n)$.**

**Solution:**

Growth rates are compared by how fast each function increases as n becomes very large. Arranging from slowest-growing (most efficient) to fastest-growing (least efficient):

```
O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(2ⁿ)
```

---

**Problem 7: An algorithm performs exactly $f(n) = 5n^3 + 20n^2 + 100n + 50$ basic operations. Express f(n) in Big O notation.**

**Solution:**
Step 1: Identify the term that grows fastest as $n \to \infty$. Among $n^3$, $n^2$, n, and the constant, the term $5n^3$ dominates for large n.

Step 2: Drop all lower-order terms ($20n^2$, 100n, 50) since they become insignificant compared to $n^3$ for large n.

Step 3: Drop the constant multiplier 5, since Big O notation is concerned only with the rate of growth, not exact coefficients.

```
f(n) = O(n³)
```

---

**Problem 8: If Algorithm A has time complexity O(n log n) and Algorithm B has time complexity $O(n^2)$, which algorithm is more efficient for large values of n, and why?**

**Solution:**
For small values of n, both algorithms may take comparable time, but as n grows large, $O(n^2)$ grows much faster than O(n log n). For example, at n = 1000: n log n ≈ 1000 × 10 = 10,000, whereas $n^2$ = 1,000,000.

Therefore, Algorithm A, with time complexity O(n log n), is more efficient than Algorithm B for large input sizes, since its running time grows at a much slower rate.

## Section C: Best, Worst, and Average Case Analysis

---

**Problem 9: Analyze the best case, worst case, and average case time complexity of Linear Search on an array of n elements.**

**Solution:**
Linear Search checks each element of the array one by one, starting from the first, until the target element is found or the array ends.

- Best Case: The target element is found at the very first position. Only 1 comparison is needed → Time Complexity = O(1).
- Worst Case: The target element is at the last position, or not present in the array at all. In this case, all n elements must be compared → Time Complexity = O(n).
- Average Case: On average, the target is expected to be found after checking about n/2 elements. Dropping the constant factor → Time Complexity = O(n).

---

**Problem 10: Analyze the worst-case time complexity of Binary Search on a sorted array of n elements.**

**Solution:**
Binary Search repeatedly divides the search interval in half. In the worst case, the target is found only after the array has been divided down to a single element (or is not found at all).

If T(n) represents the number of comparisons needed for n elements, then T(n) = T(n/2) + 1, since each step reduces the problem size by half with one comparison. Solving this recurrence gives $T(n) = \log_2(n) + 1$.

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

## Section D: Recurrence Relations

---

**Problem 11: Solve the recurrence relation T(n) = 2T(n/2) + n, with T(1) = 1, and find its time complexity using the Divide and Conquer approach (this is the recurrence for Merge Sort).**

**Solution:**

This recurrence splits the problem of size n into 2 sub-problems of size n/2 each (the 2T(n/2) term), and does n additional work to combine the results (the +n term).

Expanding the recurrence level by level: at level 0, cost = n; at level 1, there are 2 sub-problems each of size n/2, so cost = 2 × (n/2) = n; at level 2, cost = 4 × (n/4) = n; and so on. Each level contributes a total cost of n.

Since the problem size is halved at every level, the number of levels required to reduce n down to 1 is $\log_2(n)$. Multiplying the per-level cost (n) by the number of levels ($\log_2 n$):

```
T(n) = O(n log n)
```

---

**Problem 12: Solve the recurrence relation T(n) = T(n-1) + 1, with T(1) = 1 (this is the recurrence for a simple linear recursive function).**

**Solution:**

This recurrence reduces the problem size by only 1 at each step, and does 1 unit of constant work at each step. Expanding: T(n) = T(n-1) + 1 = T(n-2) + 2 = T(n-3) + 3 = ... = T(1) + (n-1) = 1 + (n-1) = n.

```
T(n) = O(n)
```

## Section E: Algorithm Design Approach Based Problems

---

**Problem 13: A shopkeeper needs to give change of Rs. 87 using the minimum number of currency notes/coins available in denominations of 50, 20, 10, 5, 2, and 1. Which algorithm design approach is suitable here, and how would it work?**

**Solution:**

This problem is best solved using the Greedy Approach, since at every step we pick the largest available denomination that does not exceed the remaining amount, without reconsidering earlier choices.

- Step 1: Remaining = 87. Pick 50 → Remaining = 37.
- Step 2: Remaining = 37. Pick 20 → Remaining = 17.
- Step 3: Remaining = 17. Pick 10 → Remaining = 7.
- Step 4: Remaining = 7. Pick 5 → Remaining = 2.
- Step 5: Remaining = 2. Pick 2 → Remaining = 0.

Total notes/coins used = 5 (denominations: 50, 20, 10, 5, 2), which is the minimum possible for this set of denominations.

---

**Problem 14: Compute the 6th Fibonacci number (F(6)) using Dynamic Programming, and explain why Dynamic Programming is more efficient than plain recursion here.**

**Solution:**

In plain recursion, F(n) = F(n-1) + F(n-2) recomputes the same sub-problems (like F(2), F(3)) many times, leading to exponential time complexity $O(2^n)$.

In Dynamic Programming, we store each computed Fibonacci value in a table and reuse it instead of recomputing it. Building the table from the bottom up:

```
F(0)=0, F(1)=1, F(2)=1, F(3)=2, F(4)=3, F(5)=5, F(6)=8
```

Since each value is computed only once and simply looked up thereafter, the Dynamic Programming solution runs in O(n) time, which is far more efficient than the exponential time taken by plain recursion.

---

**Problem 15: Calculate the space complexity of an algorithm that takes an array of n integers as input and creates one additional array of size n to store the result (e.g., a simple copy operation).**

**Solution:**

Space complexity includes space for the input and any extra (auxiliary) space used by the algorithm.

- Input space: O(n), for storing the given array of n integers.
- Auxiliary space: O(n), for the additional result array of size n created by the algorithm.

Adding these, the total space complexity is O(n) + O(n) = O(n), since constant multiples are dropped in Big O notation.

```
Space Complexity = O(n)
```