PROBLEMS AND SOLUTIONS — PART 4

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 fourth 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 Parts 1 to 3 — including direct Master Theorem applications, Huffman Coding, design-approach identification, and Divide-and-Conquer min-max search. Practice all four 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 / 3;
}
```

Solution:

Here i starts at n and is divided by 3 at every step: n, n/3, n/9, n/27, ... until it becomes ≤ 1. If the loop runs k times, then n / 3^k ≈ 1, giving k = $\log_3(n)$.

Since logarithms with different bases differ only by a constant factor ($\log_3 n = \log_2 n / \log_2 3$), and constant factors are dropped in Big O notation:

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

Problem 2:

Find the time complexity of the following code, where the inner while loop halves j at every step:

```
for (i = 0; i < n; i++) {
    j = n;
    while (j > 1) {
        printf("%d", j);
        j = j / 2;
    }
}
```

Solution:

The outer loop runs n times. For every single iteration of the outer loop, the inner while loop runs independently from j = n down to 1 by halving, which takes O(log n) time (as shown in earlier problems).

Since the inner loop's O(log n) cost is repeated for each of the n outer iterations, the total time complexity is the product:

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

Problem 3:

A function process(arr, n) itself has time complexity O(n). Find the overall time complexity of the following code, which calls process() inside a loop:

```
for (i = 0; i < n; i++) {
    process(arr, n);
}
```

Solution:

The loop runs n times, and each call to process(arr, n) costs O(n) on its own, regardless of the loop. So the total cost is n calls, each costing O(n), giving a total of n × O(n).

```
Time Complexity = O(n²)
```

This illustrates that a function call inside a loop must be analyzed for its own complexity first, and then multiplied by the number of times the loop calls it — a single loop is not automatically O(n) if the work done per iteration is not O(1).

Problem 4:

Find the overall time complexity of the following code, consisting of two sequential (non-nested) blocks:

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

Solution:

The first block is a nested loop costing O(n²). The second block is a single loop costing O(n). Since the two blocks execute one after another (sequentially, not nested inside each other), their costs are added: O(n²) + O(n).

In Big O notation, when adding terms of different growth rates, only the dominant (fastest-growing) term is kept, since it determines the overall growth for large n.

```
Time Complexity = O(n²)
```

General rule: for sequential (non-nested) code blocks, the overall complexity is the maximum of the individual complexities, not their sum written out in full.

Section B: Direct Applications of the Master Theorem

The Master Theorem solves recurrences of the form T(n) = aT(n/b) + f(n) by comparing f(n) with n^(log^b a):

- Case 1: If f(n) = O(n^(log^b a − ε)) for some ε > 0, then T(n) = Θ(n^(log^b a)).
- Case 2: If f(n) = Θ(n^(log^b a)), then T(n) = Θ(n^(log^b a) · log n).
- Case 3: If f(n) = Ω(n^(log^b a + ε)) for some ε > 0 (and a regularity condition holds), then T(n) = Θ(f(n)).

Problem 5: Solve T(n) = 4T(n/2) + n using the Master Theorem.

Solution:

Here a = 4, b = 2, f(n) = n. Compute n^(log^b a) = n^(log$_2$ 4) = n^2.

Compare f(n) = n with n^2: since n = O(n^(2−ε)) for ε = 1 (n grows strictly slower than n^2), this is Case 1 of the Master Theorem.

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

Problem 6: Solve T(n) = 2T(n/2) + n using the Master Theorem (this is the Merge Sort recurrence, now solved directly by the theorem instead of by expanding the recursion tree).

Solution:

Here a = 2, b = 2, f(n) = n. Compute n^(log^b a) = n^(log$_2$ 2) = n^1 = n.

Since f(n) = n = Θ(n^(log^b a)) = Θ(n), this matches Case 2 of the Master Theorem exactly.

$$T(n) = \Theta(n \log n)$$

Problem 7: Solve T(n) = 7T(n/2) + O(n^2) using the Master Theorem (this is the recurrence for Strassen's Matrix Multiplication algorithm).

Solution:

Here a = 7, b = 2, f(n) = n^2. Compute n^(log^b a) = n^(log$_2$ 7) ≈ n^2.81.

Compare f(n) = n^2 with n^2.81: since n^2 grows strictly slower than n^2.81, this is Case 1 of the Master Theorem.

$$T(n) = \Theta(n^{(\log_2 7)}) \approx \Theta(n^{2.81})$$

This is why Strassen's algorithm (Θ(n^2.81)) is asymptotically faster than the conventional matrix multiplication algorithm, which takes Θ(n^3).

Section C: Greedy Approach — Huffman Coding

Problem 8: Six characters a, b, c, d, e, f occur with frequencies 45, 13, 12, 16, 9, 5 (out of 100 total characters) respectively. Using the Greedy Huffman Coding algorithm, construct the Huffman codes and find the total number of bits required to encode the message. Compare this with a fixed-length encoding scheme.

Solution:

Huffman Coding is a Greedy Approach that repeatedly merges the two nodes with the smallest frequency into a new node, until only one node (the root) remains.

- Step 1: Sort by frequency: f(5), e(9), c(12), b(13), d(16), a(45). Merge f and e → new node 'fe' with frequency 14.
- Step 2: Remaining: c(12), b(13), fe(14), d(16), a(45). Merge c and b → new node 'cb' with frequency 25.
- Step 3: Remaining: fe(14), d(16), cb(25), a(45). Merge fe and d → new node 'fed' with frequency 30.
- Step 4: Remaining: cb(25), fed(30), a(45). Merge cb and fed → new node 'cbfed' with frequency 55.
- Step 5: Remaining: a(45), cbfed(55). Merge both → root with frequency 100.

Tracing the tree paths (0 for left, 1 for right) gives the following Huffman codes: a = 0 (1 bit), c = 100 (3 bits), b = 101 (3 bits), f = 1100 (4 bits), e = 1101 (4 bits), d = 111 (3 bits).

Total bits using Huffman coding = (45×1) + (13×3) + (12×3) + (16×3) + (9×4) + (5×4) = 45 + 39 + 36 + 48 + 36 + 20 = 224 bits.

With a fixed-length encoding, since there are 6 symbols, each needs $\lceil \log_2 6 \rceil = 3$ bits, giving a total of 100 × 3 = 300 bits.

```
Huffman Coding saves 300 – 224 = 76 bits compared to fixed-length
encoding.
```

This demonstrates how the Greedy Approach, by always merging the two least frequent symbols first, produces an optimal (minimum weighted path length) prefix code.

Section D: Identifying the Correct Design Approach

Problem 9: For each of the following problems, identify the most suitable algorithm design approach (Brute Force, Divide and Conquer, Greedy, Dynamic Programming, Backtracking, or Branch and Bound), with a one-line justification.

Solution:

Problem	Suitable Approach	Justification
Finding the shortest path in a weighted graph with non-negative edges (Dijkstra's Algorithm)	Greedy	Always extends the path through the nearest unvisited vertex first.
Multiplying two large numbers by splitting them into halves (Karatsuba's Algorithm)	Divide and Conquer	Breaks the problem into smaller sub-problems of the same type and combines results.
Finding the optimal way to multiply a chain of matrices (Matrix Chain Multiplication)	Dynamic Programming	Has overlapping sub-problems and optimal substructure; avoids recomputation.
Solving a Sudoku puzzle	Backtracking	Builds a partial solution and undoes choices that lead to a conflict.
Finding the optimal solution to the Travelling Salesman Problem exactly	Branch and Bound	Prunes branches of the search tree using cost bounds to avoid exploring all n! routes.

Section E: Analysis of Specific Algorithms

Problem 10: Derive the formula for the number of comparisons made by Selection Sort in the worst case, and compute the value for n = 6.

Solution:

Selection Sort works by finding the minimum element in the unsorted part of the array and placing it at the correct position, for each of the n positions.

For the 1st position, it scans (n-1) remaining elements; for the 2nd position, (n-2) elements; and so on, until the last comparison involves just 1 element. Total comparisons = (n-1) + (n-2) + ... + 1 = n(n-1)/2.

For n = 6: comparisons = 6(6-1)/2 = 6 × 5 / 2 = 30/2 = 15.

```
Total comparisons for n = 6 = 15, and in general, Time Complexity = O(n²)
```

Problem 11: Compare the auxiliary space complexity of computing the nth Fibonacci number using (a) plain recursion, and (b) Dynamic Programming with an array to store all computed values.

Solution:

(a) Plain recursion: Although this method does not use any array, each call to fib(n) leads to further recursive calls, and the maximum depth of the recursion tree (the longest chain of nested calls) is n. So the call stack requires O(n) auxiliary space at any point, even though the total number of calls made is $O(2^n)$.

(b) Dynamic Programming (array-based): This method stores all n+1 computed Fibonacci values in an array of size n+1, giving an auxiliary space complexity of O(n) as well.

Although both approaches use O(n) auxiliary space, Dynamic Programming is far more time-efficient (O(n) time vs $O(2^n)$ time for plain recursion). A further optimization of the Dynamic Programming approach — storing only the last two computed values instead of the full array — reduces the space complexity to O(1), since Fibonacci only ever needs the two previous values to compute the next one.

Problem 12:

Find the overall time complexity of the following code, which contains two sequential blocks of different complexity:

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

Solution:

The first (triple-nested) block costs $O(n^3)$. The second (double-nested) block costs $O(n^2)$. Since the blocks run sequentially, the total cost is $O(n^3) + O(n^2)$.

Keeping only the dominant term, since n^3 grows faster than n^2 for large n:

```
Time Complexity = O(n³)
```

Problem 13: Graph algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) are commonly stated to have time complexity O(V + E), where V is the number of vertices and E is the number of edges. Explain what this means, and compute the value for a graph with V = 50 vertices and E = 120 edges.

Solution:

The notation O(V + E) means that the running time of the algorithm grows linearly with the total number of vertices plus the total number of edges — every vertex is visited once (contributing the V term), and every edge is examined once while exploring neighbours (contributing the E term).

This is different from O(n) alone because a graph has two independent size parameters (vertices and edges), and neither one alone fully determines the size of the input.

For V = 50 and E = 120: total basic operations ≈ V + E = 50 + 120 = 170, so the algorithm performs work proportional to 170 for this specific graph, and in general the complexity is stated as:

```
Time Complexity = O(V + E)
```

Problem 14: Solve the recurrence $T(n) = 8T(n/2) + n^3$ using the Master Theorem.

Solution:

Here a = 8, b = 2, $f(n) = n^3$. Compute $n^{(\log_b a)} = n^{(\log_2 8)} = n^3$.

Since $f(n) = n^3 = \Theta(n^{(\log_b a)}) = \Theta(n^3)$, the two match exactly, so this falls under Case 2 of the Master Theorem.

```
T(n) = Θ(n³ log n)
```

Problem 15: Using the Divide and Conquer approach, find the minimum number of comparisons needed to find both the maximum and minimum elements of an array of n = 8 elements, and compare it with the naive approach of finding max and min separately.

Solution:

Naive approach: Finding the maximum alone requires (n-1) comparisons, and finding the minimum alone requires another (n-1) comparisons, giving a total of 2(n-1) comparisons. For n = 8: 2(8-1) = 14 comparisons.

Divide and Conquer approach: The array is split into pairs of elements. Each pair requires 1 comparison to determine its own local max and min. Then, the local maximums are compared among themselves, and the local minimums are compared among themselves, to find the overall max and min.

The total number of comparisons using this method works out to approximately 3n/2 − 2. For n = 8: 3(8)/2 − 2 = 12 − 2 = 10 comparisons.

```
Naive approach = 14 comparisons; Divide and Conquer approach = 10
comparisons
```

This shows that the Divide and Conquer strategy reduces the number of comparisons from 2n to about 1.5n, giving Time Complexity = O(n) in both cases, but with a smaller constant factor for the Divide and Conquer method.