PROBLEMS AND SOLUTIONS — PART 2

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 second set of frequently asked numerical and analytical problems on Algorithms, Design Approaches, Complexity, Big O Notation, and Algorithm Analysis, covering additional problem types not included in Part 1. Practice both sets together for complete exam coverage of this chapter.

Section A: Time Complexity of Loops and Recursive Code

Problem 1:

Find the time complexity of the following code:

```
for (i = 0; i < n; i = i + 2) {
    printf("%d", i);
}
```

Solution:

Here i increases by 2 in each step instead of 1, so the loop runs approximately n/2 times instead of n times. Since Big O notation drops constant factors, n/2 is still of the order n.

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

Problem 2:

Find the time complexity of the following code:

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

Solution:

The outer loop depends on n and runs n times; the inner loop depends on a different variable m and runs m times for every iteration of the outer loop. The two variables are independent, so the total number of iterations is n × m.

```
Time Complexity = O(n × m)
```

Note: This is different from $O(n^2)$, which would apply only if both loops depended on the same variable n.

Problem 3:

Find the time complexity of the following recursive function:

```
int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Solution:

Each call to fact(n) makes exactly one recursive call, to fact(n-1), and does a constant amount of work (one multiplication) besides that call. The recursion continues until n reaches 0, so the total number of calls is n + 1.

Writing the recurrence: T(n) = T(n-1) + 1, with T(0) = 1. Solving this gives T(n) = O(n).

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

Problem 4:

Find the time complexity of the following recursive function:

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Solution:

Each call to fib(n) makes two further recursive calls: fib(n-1) and fib(n-2). This means the number of calls roughly doubles at each level of recursion, forming a binary recursion tree of depth n.

The recurrence relation is T(n) = T(n-1) + T(n-2) + 1, which is known to grow exponentially.

```
Time Complexity = O(2ⁿ)
```

This is why the naive recursive Fibonacci is inefficient for large n, and Dynamic Programming (as seen in Part 1) is used to reduce it to O(n).

Section B: Comparing and Ordering Algorithms by Complexity

Problem 5: Three algorithms A, B, and C solve the same problem with time complexities $O(n^2)$, $O(n \log n)$, and $O(n)$, respectively. For an input of size n = 10,000, estimate the relative number of operations each algorithm performs and rank them by efficiency.

Solution:

Approximate operation counts for n = 10,000 (using log base 2, $\log_2(10000) \approx 13.3$):

- Algorithm C: $O(n) \approx 10{,}000$ operations.
- Algorithm B: $O(n \log n) \approx 10{,}000 \times 13.3 \approx 133{,}000$ operations.
- Algorithm A: $O(n^2) \approx 10{,}000 \times 10{,}000 = 100{,}000{,}000$ operations.

Ranking from most efficient to least efficient: Algorithm C ($O(n)$) is the most efficient, followed by Algorithm B ($O(n \log n)$), and Algorithm A ($O(n^2)$) is the least efficient for large inputs.

Problem 6: Prove using the definition of Big O notation that f(n) = 4n + 3 is O(n).

Solution:

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

Here f(n) = 4n + 3 and we want to show f(n) = O(n), i.e., g(n) = n.

For $n \geq 1$: $4n + 3 \leq 4n + 3n = 7n$ (since $3 \leq 3n$ when $n \geq 1$). So $f(n) \leq 7n$ for all $n \geq 1$.

Poly Notes Hub

Choosing $c = 7$ and $n_0 = 1$ satisfies the definition, since $f(n) \leq 7 \cdot n$ for all $n \geq 1$.

```
Hence, f(n) = 4n + 3 is O(n).
```

Section C: Sorting Algorithm Complexity Comparison

Problem 7: Compare the worst-case time complexity of Bubble Sort, Selection Sort, Insertion Sort, and Merge Sort for sorting n elements.

Solution:

The worst-case time complexities of these commonly taught sorting algorithms are as follows:

- Bubble Sort — $O(n^2)$, since in the worst case every pair of adjacent elements may need to be compared and swapped across n passes.
- Selection Sort — $O(n^2)$, since for each of the n positions, the algorithm scans the remaining unsorted elements to find the minimum.
- Insertion Sort — $O(n^2)$, since in the worst case (reverse sorted input) each new element may need to be compared with and shifted past all previously sorted elements.
- Merge Sort — O(n log n), since it always divides the array in half and merges in linear time regardless of the input arrangement.

Merge Sort is asymptotically more efficient than the other three for large n, since O(n log n) grows much more slowly than $O(n^2)$.

Problem 8: How many comparisons does Bubble Sort perform in the worst case to sort an array of n = 5 elements? Show the calculation.

Solution:

In the worst case, Bubble Sort compares every pair of elements across all passes. The number of comparisons is given by the sum (n-1) + (n-2) + ... + 1 = n(n-1)/2.

For n = 5: comparisons = 5(5-1)/2 = 5 × 4 / 2 = 20/2 = 10.

```
Total comparisons in the worst case = 10
```

Section D: Algorithm Design Approach Problems

Problem 9: A set of activities with start and finish times must be scheduled on a single resource so that the maximum number of non-overlapping activities are selected. Which algorithm design approach is suitable, and what is the general strategy?

Solution:

This is the classic Activity Selection Problem, which is solved efficiently using the Greedy Approach.

Strategy: First, sort all activities by their finish time in increasing order. Then, select the first activity. For each subsequent activity, select it only if its start time is greater than or equal to the finish time of the previously selected activity.

This greedy choice — always picking the activity that finishes earliest among the remaining valid options — guarantees the maximum number of non-overlapping activities can be scheduled, and the algorithm runs in O(n log n) time (dominated by the sorting step).

Poly Notes Hub

Problem 10: Explain, with an example, why the Greedy Approach does not always give the optimal solution for the 0/1 Knapsack Problem, whereas Dynamic Programming does.

Solution:

Consider a knapsack of capacity 10 kg with three items: Item 1 (weight 6 kg, value 30), Item 2 (weight 5 kg, value 25), Item 3 (weight 5 kg, value 25).

A greedy approach based on value-per-weight ratio would pick Item 1 first (ratio = 5), filling 6 kg, leaving only 4 kg capacity — not enough for Item 2 or Item 3 (5 kg each). Total value obtained = 30.

However, picking Item 2 and Item 3 together uses exactly 10 kg and gives a total value of 25 + 25 = 50, which is better than the greedy result of 30.

This happens because, unlike the Fractional Knapsack, items in the 0/1 Knapsack cannot be broken into parts, so a locally optimal greedy choice can block a better combination later. Dynamic Programming avoids this by systematically considering all combinations of items and capacities using a table, guaranteeing the optimal solution.

Problem 11: A robot needs to find a path from the top-left corner to the bottom-right corner of a grid, avoiding blocked cells, and must be able to undo a move if it leads to a dead end. Which algorithm design approach is most suitable, and why?

Solution:

This problem is best solved using the Backtracking approach.

In backtracking, the robot builds a path step by step, moving into a cell only if it is valid (not blocked and not already visited). If it reaches a dead end (no valid moves possible from the current cell), it backtracks — undoes the last move — and tries a different direction from the previous cell.

This systematic try-and-undo strategy ensures that all possible paths are explored without repeating work unnecessarily, and it will find a valid path (if one exists) or correctly determine that no path exists.

Section E: Space Complexity and Miscellaneous

Problem 12:

Find the space complexity of the following recursive factorial function:

```
int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Solution:

Although this function does not use any extra arrays, each recursive call adds a new frame to the call stack (to store the parameter n and the return address) until the base case n == 0 is reached.

Since the maximum depth of recursion is n, the maximum number of stack frames active at once is n + 1.

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

This is called the auxiliary space due to recursion, and it is an important consideration separate from time complexity.

Problem 13: Explain the term 'space-time trade-off' in algorithm design with a suitable example.

Solution:
A space-time trade-off is a situation in algorithm design where an algorithm can be made to run faster (reduce time complexity) by using more memory (increasing space complexity), or made to use less memory by allowing it to run slower.

Example: To check whether a number has already been seen in a stream of numbers, one approach is to search a list each time (O(n) time per check, O(1) extra space), while another approach is to store seen numbers in a hash set (O(1) average time per check, but O(n) extra space for the hash set).

The hash set approach trades additional memory for significantly faster lookup time, which is a classic example of a space-time trade-off. Choosing between them depends on whether time or memory is the more limited resource in a given situation.

Problem 14: Two algorithms solve the same sorting problem: Algorithm X takes $2n^2 + 3n$ microseconds, and Algorithm Y takes $50n \log n$ microseconds. Find the approximate value of n beyond which Algorithm Y becomes faster than Algorithm X.

Solution:
We need to find n such that $50n \log n < 2n^2 + 3n$. Dividing both sides by n ($n > 0$): $50 \log n < 2n + 3$.

Testing n = 20: LHS = $50 \times \log_2(20) \approx 50 \times 4.32 = 216$; RHS = $2(20) + 3 = 43$. LHS > RHS, so X is still faster at n = 20.

Testing n = 100: LHS = $50 \times \log_2(100) \approx 50 \times 6.64 = 332$; RHS = $2(100) + 3 = 203$. LHS is still greater, so X is faster.

Testing n = 200: LHS = $50 \times \log_2(200) \approx 50 \times 7.64 = 382$; RHS = $2(200) + 3 = 403$. Now RHS > LHS, meaning Algorithm Y has become faster.

So the crossover point lies between n = 100 and n = 200 (closer to $n \approx 190$–200). This illustrates that an algorithm with better asymptotic complexity ($O(n \log n)$) eventually outperforms one with worse complexity ($O(n^2)$), even if it has a larger constant factor, once n becomes sufficiently large.

Problem 15: State whether the following statement is True or False, with justification: "An algorithm with time complexity O(n) is always faster than an algorithm with time complexity $O(n^2)$, for every value of n."

Solution:
The statement is False.

Big O notation describes the asymptotic (long-term) growth rate of an algorithm's running time as n becomes very large, and it ignores constant factors. It does NOT guarantee that an O(n) algorithm is faster for every value of n, especially for small n.

For example, an O(n) algorithm with a large constant factor, say 1000n, will take longer than an $O(n^2)$ algorithm with a small constant factor, say n^2, for small values of n. At n = 10: 1000n = 10,000, while $n^2 = 100$ — here the $O(n^2)$ algorithm is actually faster.

The O(n) algorithm is guaranteed to become faster only after n crosses a certain threshold value (n_0), beyond which its growth advantage takes over. This is precisely why Big O comparisons are meaningful only for sufficiently large n, not for every value of n.

Poly Notes Hub