# quiz3_sols

August 12, 2025

## 1 Addis Coder Quiz 3 (2025 SOLUTIONS)

Welcome to Quiz 3! You will have 2 hours to complete this written quiz.

This quiz will cover material from weeks 1 through 3, including: * Time Complexity * Search Algorithms * Sorting Algorithms * Graph Traversal

### 1.1 Problem 1: Algorithm Comparison

**1.1** Which of the following time complexities grows the slowest as n increases?

a) $O(n)$ *

b) $O(n \log n)$

c) $O(n^2)$

d) $O(2^n)$

**1.2** You are comparing two sorting algorithms: one with time complexity $O(n \log n)$ and another with $O(n^2)$. For large inputs, which is more efficient?

a) $O(n \log n)$ *

b) $O(n^2)$

c) They are the same

d) Depends on the values in the list

### 1.2 Problem 2: Time Complexity

What is the time compexity of the following functions? **Circle the correct choice** for each.

**2.1**

```
def count_pairs(n):
    for i in range(n):
        for j in range(n):
            print(i, j)
```

a) $O(n)$

b) $O(n \log n)$

c) $O(n^2)$ *

d) $O(2^n)$

**2.2**

```
def half_count(n):
    i = n
    while i > 1:
        i = i // 2
        print(i)
```

a) $O(\log n)$ *

b) $O(n)$

c) $O(n \log n)$

d) $O(n^2)$

**2.3**

```
def mystery(n):
    for i in range(n):
        for j in range(10):
            print(i, j)
```

a) $O(n)$ *

b) $O(n^2)$

c) $O(1)$

d) $O(n \log n)$

**2.4**

```
def run_something(n):
    for i in range(n):
        if i % 3 == 0:
            for j in range(n):
                print(i + j)
```

a) $O(n)$

b) $O(n \log n)$

c) $O(n^2)$ *

d) $O(2^n)$

## 1.3 Problem 3: More Big-O

**3.1** Below are the number of operations performed by 4 different algorithms. Circle the option with the most operations for large n.

a) $50n + n \log n$

b) $n^2 + 100$

c) $n^3$ *

d) $\log(n) + 2n$

**3.2** Write out the **Big-O time complexity** for all the algorithms in **Problem 3.1**.

a) _____ $O(n \log n)$

b) _____ $O(n^2)$

c) _____ $O(n^3)$

d) _____ $O(n)$

## 1.4 Problem 4: True or False

**State whether the following statements are True or False.**

**I)** Binary search has time complexity O(n) in the worst case.

`Answer: _________`

False

**II)** Selection sort always makes the same number of comparisons regardless of input.

`Answer: _________`

True

**III)** A directed graph can have more edges than vertices.

`Answer: _________`

True

## 1.5 Problem 5: Adding an Element to a Sorted List

Suppose you want to write a function `add_and_sort(x, L)` that adds a new element `x` to a list `L` and returns the sorted version of the list. You are given the following four versions:

```
def add1(x, L):
    L.append(x)
    return sorted(L)

def add2(x, L):
    L = sorted(L)
    L.append(x)
    return sorted(L)

def add3(x, L):
    L.append(x)
    return insertion_sort(L)

def add4(x, L):
    return insertion_sort(L + [x])
```

Assume `insertion_sort()` is a correct implementation of insertion sort.

**5.1** Circle all implementations that will give the correct answer.

a) add1 *

b) add2 *

c) add3 *

d) add4 *

**5.2** Suppose L is already sorted and has length n. Write the Big-O time complexity for each version.

a) add1 → _____ $O(n \log n)$

b) add2 → _____ $O(n \log n)$

c) add3 → _____ $O(n^2)$

d) add4 → _____ $O(n^2)$

**5.3** Suppose now L is unsorted and has length n. Write the Big-O time complexity for each version.

a) add1 → _____ $O(n \log n)$

b) add2 → _____ $O(n \log n)$

c) add3 → _____ $O(n^2)$

d) add4 → _____ $O(n^2)$

### 1.6 Problem 6: Find 99

Write a function `find_99(L)` that takes as input a sorted list of integers `L` and uses **binary search** to find and return the position (index) of the number `99`. If `99` is not in the list, `find_99(L)` should return -1.

For example, if `L = [88, 53, 90, 99]`, then `find_99(L)` would return 3 and if `L = [-30, 8, 23, 43, 89]`, then `find_99(L)` would return -1.

What is the time complexity of your `find_99` algorithm, if $n$ is the length of `L`?

`Answer: _____`

$O(\log n)$

### 1.7 Problem 7: Implement Your Own Sort

Without using `sorted()` or `list.sort()`, Write a function sort_first_letter(L) that takes a list of strings and returns them sorted in **ascending order** of their first letter.

For example, `sort_first_letter(["banana", "apple", "fig", "pear"])` → `["apple", "banana", "fig", "pear"]`

What is the time complexity of your `find_99` algorithm, if $n$ is the length of `L`?

`Answer: _____`

$O(n^2)$

## 1.8 Problem 8: Mountain Climbing

You are a new mountain climber and want to count the number of peaks along the trail. You are given a sequence of heights along the trail. The trail has a "peak" at index `i` if `heights[i-1] < heights[i] > heights[i+1]`. (There are no peaks at the beginning or end of the trail.)

For example, `heights = [3, 5, 2, 9, 4]` has one peak at index `1` and one peak at index `3`.

Write a function `count_peaks(heights)` that takes in a list of heights `heights` and returns the number of peaks.

## 1.9 Problem 9: Graphs

**9.1** Draw the following directed graph below: `{0: [3], 1: [1, 3], 2: [0, 3], 3: []}`

**9.2** What is the **list of edges** of this graph?

`Answer: _________`

[(0, 3), (1, 1), (1, 3), (2, 0), (2, 3)]

**9.2** Is the graph above the same as the graph below? (Hint: two graphs are the same if their **list of edges** is the same.)

`Answer: _________`

False

**9.3** For each graph below, write `True` if each node has the same in-degree and out-degree as in the graph above (in problem 9.2), and otherwise `False`.

a) _____ True

b) _____ True

c) _____ False

## 1.10 Problem 10: Graph Traversal

**10.1** Write a function `flip_edges(G)` that takes a directed graph `G` as an adjacency list and returns the graph with all the edges flipped.

An example of a graph and its flipped version is shown below.

**10.2** Imagine you are an explorer looking for treasure. You are given a directed graph G. Each node $u$ represents a position on a map. An edge from $u$ to $v$ means that you can travel from position $u$ to position $v$. You are also given a set of "treasure nodes" T.

Write a function `can_reach_treasure(G, T)` that returns the list of nodes that can reach a treasure node.

For example, in the graph above (problem 10.1), if node `1` is the only treasure node, then `can_reach_treasure(G, T)` should return `[0, 1, 2, 4]`, since only node `3` cannot reach node `1`.

Hint: you may use the `flip_edges` function you defined above.

[ ]:
```python
def can_reach_treasure(G, T):
    G_flip = flip_edges(G)
    visited = set()
    stack = T
    while stack:
        current = stack.pop()
        for neighbor in G_flip[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append(neighbor)
    return visited
```