Quiz 3 Solutions

Name: ____________________________

Lab: ________________

Directions

Congrats on completing Week 3!

- **This quiz contains 11 questions and will take about 90 minutes.**
- You may use any method to solve a problem unless it clearly asks for a recursive or iterative solution.
- The questions are not necessarily in order of difficulty.
- Please write your full name and lab number on the first page of the quiz.
- When writing code, make sure it is indented clearly and correctly.
- For questions with *circular bubbles*, you should select *exactly one* choice.
  - ○ You must choose either this option
  - ○ Or this one, but not both!

1. For each pair of Big O time complexities below, circle the one that takes fewer operations for large enough n.

a) $O(n^4)$ vs. $\boxed{O(n^2)}$

b) $O(n \log n)$ vs. $\boxed{O(n)}$

c) $\boxed{O(1)}$ vs. $O(\log n)$

d) $O(n^n)$ vs. $\boxed{O(\log n)}$

e) $O(n^2)$ vs. $\boxed{O(n \log n)}$

f) $\boxed{O(n^{1000})}$ vs. $O(2^n)$

2. Each expression below is the number of steps an algorithm uses. Write its tightest Big O bound in terms of n.

a) $2n + 500 + 3n^2$ Answer: $O(n^2)$

b) $3 \log n + 2^n$ Answer: $O(2^n)$

c) $n \log n + 8n$ Answer: $O(n \log n)$

d) $16^4 + 5n^3$ Answer: $O(n^3)$

3. For each function below, circle its tightest Big O running time in terms of n.

a)

```python
def function_1(n):
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                print(i, j, k)
```

○ $O(3^n)$ ○ $O(n^2)$

● $O(n^3)$ ○ $O(1)$

b)

```python
def function_2(n):
    total = 0
    for i in range(100):
        for j in range(50):
            if i % 2 == 0:
                total += j
    return total * n
```

○ $O(n^2)$ ● $O(1)$

○ $O(n)$ ○ $O(n \log n)$

c)

```
def function_3(n):
    if n == 0:
        return
    print(n)
    function_3(n - 1)
    function_3(n - 1)
```

○ $O(\log n)$ ○ $O(n^2)$

○ $O(n)$ ● $O(2^n)$

d)

```
def function_4(n):
    i = 1
    while i < n:
        for j in range(n):
            print(i, j)
        i = i * 2
```

○ $O(n)$ ○ $O(n^2)$

● $O(n \log n)$ ○ $O(2^n)$

e)

```
def function_5(n):
    for i in range(n):
        for j in range(n):
            k = 1
            while k < n:
                print(i, j, k)
                k = k * 2
```

○ $O(n^2)$

● $O(n^2 \log n)$

○ $O(n^3)$

○ $O(2^n)$

4. The binary search function below has an error.

```
def binary_search_buggy(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid
        else:
            high = mid - 1
    return -1
```

Hint: try running `binary_search_buggy([1, 3, 4], 2)` by hand (follow `low`, `high`, and `mid` step by step).

Sample simulation:

1) low $= 0$, high $= 2$, mid $= 1$, `arr[1]` $= 3 > 2 \rightarrow$ high $= 0$

2) low $= 0$, high $= 0$, mid $= 0$, `arr[0]` $= 1 < 2 \rightarrow$ low $= 0$

3) Same as step 2: infinite loop

a) What happens when we run the buggy function?

○ Returns a wrong answer ○ Accesses an invalid index

● Enters an infinite loop

b) Write down the line of code that contains the bug.

Answer: low = mid

c) Write down the line of code that should replace the line above.

Answer: low = mid + 1

5. Below is a working Bubble Sort implementation in Python:

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
    return lst
```

If `lst` is already sorted and has length n, then `bubble_sort(lst)` still uses $O(n^2)$ operations.

a) Fill in the missing lines of `better_bubble_sort(lst)` so it uses only $O(n)$ operations when the input list is already sorted.

Hint: how can you tell that no swaps happened during one full pass through the list?

```
def better_bubble_sort(lst):
    n = len(lst)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]
                swapped = True
        if not swapped:
            break
    return lst
```

6. Selection Sort repeatedly finds the smallest value in the unsorted part of the list and swaps it into the next position.

Starting with `lst = [5, 1, 4, 2, 3]`, write the state of the list after each outer-loop iteration.

a) After iteration 1: `[1, 5, 4, 2, 3]`

b) After iteration 2: `[1, 2, 4, 5, 3]`

c) After iteration 3: `[1, 2, 3, 5, 4]`

d) After iteration 4: `[1, 2, 3, 4, 5]`

7. In this problem, graphs are **directed**. We represent a graph either as an **edge list** (a list of pairs `(u, v)` meaning an edge from `u` to `v`) or as an **adjacency dictionary** (each key is a vertex, and its value is the list of vertices it points to).

Write a function `to_edge_list(adj_dict)` that converts an adjacency dictionary into an edge list.

Example: `to_edge_list({0: [1, 2], 1: [2], 2: []})` returns `[(0, 1), (0, 2), (1, 2)]`.

```
def to_edge_list(adj_dict):
    edge_list = []
    for x, ys in adj_dict.items():
        for y in ys:
            edge_list.append((x, y))
    return edge_list
```

8. The function below is supposed to merge two already-sorted lists into one sorted list. It has a bug.

```python
def merge(left, right):
    i = 0
    j = 0
    result = []

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    return result
```

a) What does the buggy function return on `merge([1, 4, 7], [2, 3])`?

Answer: `[1, 2, 3]`

b) What should the correct output be?

Answer: `[1, 2, 3, 4, 7]`

c) Write the code that should be added before `return result` to fix the function. You do not need to rewrite the whole function.

```python
while i < len(left):
    result.append(left[i])
    i += 1

while j < len(right):
    result.append(right[j])
    j += 1
```

d) Using the corrected `merge` function, write the full `merge_sort(lst)` function.

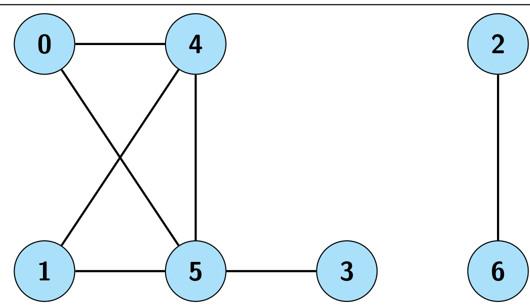
```
def merge_sort(lst):
    if len(lst) <= 1:
        return lst

    mid = len(lst) // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left, right)
```

9. Consider the **undirected** graph G with $N = 7$ vertices (indexed 0 through 6) and edge set

$$E = \{(0,4),\ (0,5),\ (1,4),\ (1,5),\ (2,6),\ (3,5),\ (4,5)\}.$$

a) Draw the graph.



(Note: vertices 2 and 6 form a separate component.)

b) True or false?

i) Vertex 0 is reachable from vertex 1.

● True ○ False

ii) Vertex 2 is reachable from vertex 0.

○ True ● False

iii) The graph G is connected.

○ True ● False

c) Write down all paths, with no repeated vertices, from vertex 0 to vertex 3 (you may not need all the lines).

$0 \to 5 \to 3$

$0 \to 4 \to 5 \to 3$

$0 \to 4 \to 1 \to 5 \to 3$

d) Of all the paths from 0 to 3, how many edges does the shortest one have? (This is called the *distance* between 0 and 3.)

Answer: **2**

10. Consider the **directed** graph below, given by its adjacency list:

```
G = { 0: [1, 3],  1: [2],  2: [0],  3: [5, 1],  4: [0],  5: [6, 4],  6: []}
```

Perform a Breadth-First Search (BFS) starting at vertex 0. Whenever a vertex has more than one unvisited neighbor, visit them in the order they appear in the adjacency list above. Write down the order in which the vertices are visited.

Space for the drawing if you need it (optional):

Answer: 0 1 3 2 5 6 4

11. The TAs are staying across N hotel rooms. Tonight, they plan to throw a secret party. To avoid being seen, everyone must gather in a single room (any of them) through underground tunnels connecting pairs of rooms.

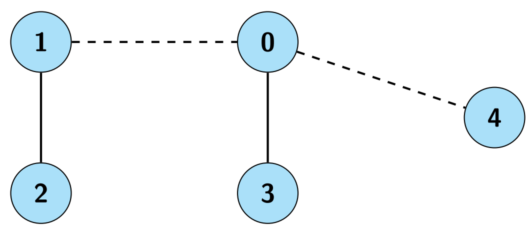
They have already discovered several secret tunnels connecting some of the rooms. What is the smallest number of new tunnels they must dig so everyone can reach the party?

Hint: if you want to minimize the tunnels, you never want to connect two rooms that are already connected.

a) Solve the following example where $N = 5$ (rooms are indexed $0, 1, 2, 3, 4$) and the existing tunnels are $(0, 3)$ and $(1, 2)$.

Draw the graph showing the existing tunnels and the new tunnels you add. **Use dashed lines for the new tunnels.** Example: - - - - - -

Draw the graph.



Min. number of new tunnels: 2

Valid list of new tunnels: `[(1, 0), (0, 4)]`

b) Given an adjacency dictionary G representing the existing tunnels between the N rooms, implement a traversal function `mark_reachable(G, start, visited)`. You may use DFS or BFS. Assume every room $0, \ldots, N-1$ appears as a key in `G`, with `[]` if it has no tunnels. Since tunnels are undirected, each tunnel appears in both directions. The function should mark `start` itself and every room reachable from `start` as visited.

Hint: *`visited` can be a list, set, or dictionary. Depending on your choice, marking a room as visited might look like `visited.append(room)`, `visited.add(room)`, or `visited[room] = True`.*

```
def mark_reachable(G, start, visited):
    visited.add(start)
    for v in G[start]:
        if v not in visited:
            mark_reachable(G, v, visited)
```

c) With the help of the traversal function from part (b), implement `get_extra_tunnels(G)`.

Even if you could not finish part (b), you may assume here that `mark_reachable(G, start, visited)` correctly marks every room reachable from `start` as visited. The adjacency dictionary has the same format as in part (b). The function should return the minimum number of additional tunnels needed.

```python
def get_extra_tunnels(G):
    visited = set()
    components = 0

    for room in G:
        if room not in visited:
            components += 1
            mark_reachable(G, room, visited)

    return components - 1
```

Scratch space (not graded)