Midterm Exam

Part 1: Algorithm Analysis and Conceptual Thinking (40 points)

1. **(10 points)** Consider the following code snippet. Analyze its **worst-case time complexity** using Big-O notation and justify your reasoning:

```python
def mystery(arr):
    n = len(arr)
    for i in range(n):
        for j in range(i, n):
            k = 1
            while k < n:
                k *= 2
                print(arr[i], arr[j])
```

   What is the time complexity, and what factors contribute to it? Explain each component of the code and its effect on overall performance.

2. **(10 points)** Consider the following code snippet. Analyze its worst-case time complexity and justify why:

```python
def mystery(arr):
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            print(arr[i], arr[j])
```

   What is the best and worst-case scenario for this algorithm?

3. **(10 points)** Propose an algorithm to find the **k-th smallest element** in an unsorted array. Analyze its time complexity. What would change if the array were already sorted?

4. **(10 points)** Explain the difference between an algorithm's **time complexity** and **space complexity** using your own words. Provide an example where an algorithm has low time complexity but high space complexity, and another where the reverse is true.

Part 2: Data Structures and Algorithm Design (60 points)

5. **(20 points)** You are given a **binary search tree** and asked to design and implement an algorithm that outputs the **second smallest** element in the tree. Write the code, and explain how

your implementation ensures the correct output. Justify its efficiency in terms of time complexity.

6. **(20 points)** Implement an **in-place** algorithm to **reverse a singly linked list**. Explain how the algorithm works, write the code, and analyze its time and space complexity.
7. **(20 points)** You need to develop an algorithm to find all **anagrams** of a given word within a large dictionary of 500,000 words. For example, given the word "silent", the algorithm should identify "listen", "enlist", and "tinsel". Discuss the optimal data structure and approach for this problem, and provide the pseudocode (or, optionally, write the python code).

Extra Credit: Classroom Scheduling Algorithm (Up to 10 Points)

In the provided script, you will find a function named `student_schedule_algorithm` with a placeholder implementation. Your task is to complete this function to create a scheduling algorithm that assigns classes to classrooms while avoiding time conflicts.

Problem Description:

You have a list of **classrooms**, each with a unique ID and availability time slots. Additionally, you have a list of **classes**, each with a specified start time, end time, and duration. Your goal is to develop a scheduling algorithm that:

1. **Assigns classes to classrooms** such that no two classes overlap within the same classroom.
2. **Maximizes classroom utilization** by scheduling as many classes as possible without conflicts.
3. Handles variations in classroom availability and class durations effectively.

Function Requirements:

1. Implement the `student_schedule_algorithm(classrooms, classes, timeslots)` function.
2. The function should take in three parameters:
    - `classrooms`: A list of classroom dictionaries, each containing a unique `Classroom_ID`, `Classroom_Name`, and `Capacity`.
    - `classes`: A list of class dictionaries, each containing a unique `Class_ID`, `Class_Name`, `Duration`, `Start_Time`, and `End_Time`.
    - `timeslots`: A list of time slot dictionaries for each classroom, containing `Classroom_ID`, `Available_Start_Time`, and `Available_End_Time`.
3. The function should return a **schedule** in the following format:

```
schedule = {
    1: [101, 102],  # Classroom 1 has classes 101 and 102
    2: [103],       # Classroom 2 has class 103
    3: [104, 105]   # Classroom 3 has classes 104 and 105
}
```

Each key in the dictionary represents a classroom ID, and the value is a list of class IDs scheduled in that classroom.

Additional Instructions:

- **Do not modify** the rest of the provided script, including the data generation and validation functions.
- Test your function using the `test_schedule_algorithm()` function provided in the script. This will generate random datasets and validate your solution.

- Ensure that no two classes in the same classroom overlap in time. Use the `is_schedule_valid` function to check the validity of your schedule.
- Include comments in your function to explain your logic and approach.

Tips:

- Start by sorting the classes and time slots by start time to simplify your logic.
- Consider using a **greedy algorithm** or an **iterative approach** to assign classes to the first available room.
- You don’t need to find the optimal solution but should aim to minimize conflicts and maximize room utilization.

How to Use the Script:

- Open the provided Python file.
- Implement your logic inside the `student_schedule_algorithm` function.
- Run the script to test your solution against randomly generated datasets.
- Review the results printed by the `is_schedule_valid` function to ensure your implementation is correct.