Exercise 1: Object-Oriented Programming with Properties and Magic Methods (30-40 minutes)

Create a `BankAccount` class that demonstrates proper encapsulation and operator overloading:

Requirements:

- Private balance attribute that cannot be directly accessed
- Property decorators for safe balance access with validation
- Implement `__str__`, `__repr__`, and `__eq__` magic methods
- Overload `+` and `-` operators for deposits and withdrawals
- Context manager protocol (`__enter__` and `__exit__`) for transaction logging
- Custom exception handling for insufficient funds

Example usage your class should support:

python
```
with BankAccount("Alice", 1000) as account:
    account += 500  # deposit
    account -= 200  # withdrawal

    print(account.balance)  # Should print 1300
```

Exercise 2: Advanced Iterators and Generators (30-40 minutes)

Implement a custom iterator class `PrimeGenerator` and corresponding functions:

Part A: Create an iterator that generates prime numbers up to a given limit

- Implement `__iter__` and `__next__` methods
- Include proper StopIteration handling
- Add a reset method to restart iteration

Part B: Write a generator function `fibonacci_primes(n)` that yields the first n numbers that are both Fibonacci numbers and prime numbers

Part C: Create a decorator `@memoize` that caches function results and apply it to optimize a recursive function

Bonus: Make your iterator work with `itertools` functions like `islice` and `takewhile`

Exercise 3: Graph Algorithm with Advanced Python Features (50-60 minutes)

Implement Dijkstra's shortest path algorithm using Python's advanced features:

Requirements:

- Use `collections.defaultdict` and `heapq` module appropriately
- Implement the algorithm as a class method within a `Graph` class
- Use type hints throughout your implementation
- Include comprehensive docstrings following Google/NumPy style
- Implement path reconstruction to return actual shortest paths, not just distances
- Add data validation using `@property` setters
- Handle edge cases (disconnected graphs, negative weights, self-loops)

Additional features to implement:

- A `@classmethod` constructor that builds a graph from an adjacency list
- Method chaining for graph operations
- Custom string representation showing graph statistics
- Unit tests using assertions to verify your algorithm works correctly

The class should support usage like:

python

```
g = Graph.from_adjacency_list({
    'A': [('B', 4), ('C', 2)],
    'B': [('C', 1), ('D', 5)],
    'C': [('D', 8), ('E', 10)],
    'D': [('E', 2)]
})

distance, path = g.dijkstra('A', 'E')
print(f"Shortest distance: {distance}, Path: {' -> '.join(path)}")
```

Grading Distribution:

- Exercise 1: 25 points (Focus on OOP concepts and Python magic methods)
- Exercise 2: 25 points (Iterator protocol and generator functions)
- Exercise 3: 50 points (Algorithm implementation + advanced Python features)

Tips for students:

- Read all exercises first and allocate time accordingly
- Exercise 3 is the most complex - start with a basic implementation then add features
- Code style, documentation, and error handling matter for full credit
- Test your code with the provided examples

This exam tests core Python concepts (OOP, iterators, decorators), advanced language features (magic methods, context managers, type hints), and algorithmic thinking while remaining challenging but achievable in 2 hours.