Python Programming Examination

Advanced Programming Division

Duration: 2 Hours Maximum Marks: 50

Section A: Algorithm Development (25 Marks)

Write detailed algorithms for the following problems. Include time and space complexity analysis.

1. (5 marks) Design an algorithm to find all possible anagrams of a given string that are also valid English words. Your algorithm should minimize memory usage while maintaining reasonable time complexity. Explain how you would handle:
   - Dictionary lookup optimization
   - Memory management for large strings
   - Time-space tradeoffs in your solution
2. (5 marks) Create an algorithm for a circular buffer implementation using Python lists that automatically overwrites the oldest elements when the buffer is full. The buffer should support:
   - Efficient insertion and deletion
   - Circular traversal
   - Memory-efficient storage
3. (5 marks) Design an algorithm to find the longest palindromic subsequence in a string using dynamic programming. Your algorithm should:
   - Use optimal memoization
   - Handle both even and odd-length palindromes
   - Include space optimization techniques
4. (5 marks) Develop an algorithm for implementing a nested dictionary flattener that converts a deeply nested dictionary into a flat dictionary with compound keys. Consider:
   - Handling of different data types
   - Recursive vs iterative approaches
   - Memory efficiency for deep structures
5. (5 marks) Create an algorithm for a custom sorting method that sorts strings based on both their length and lexicographical order, with length having higher priority. Include:
   - Comparison logic
   - Implementation considerations
   - Optimization strategies

Section B: Python Internals and Concepts (20 Marks)

Answer the following questions with detailed explanations.

6. (2 marks) Explain the memory implications of using a list comprehension versus a generator expression. How does Python handle memory allocation in each case?
7. (2 marks) Describe the memory footprint of a single item in a Python list. Include overhead costs and explain how Python's memory management affects this.
8. (2 marks) Compare and contrast the memory usage of strings in Python 2 vs Python 3. How has string internment evolved?
9. (2 marks) Explain how Python's garbage collector handles circular references in dictionaries. What mechanisms are used to detect and clean up such references?
10. (2 marks) Describe the internal implementation of Python's dictionary and how it handles hash collisions. How does this affect performance?
11. (2 marks) How does Python implement immutability for tuples? What are the memory implications compared to lists?
12. (2 marks) Explain the concept of string interning in Python. When does Python automatically intern strings?
13. (2 marks) Describe how Python's interpreter handles variable scope in nested functions. What is the memory impact of closures?
14. (2 marks) Compare the memory efficiency of using __slots__ versus regular instance dictionaries in Python classes.
15. (2 marks) Explain how Python's integer caching works. What range of integers are typically pre-cached and why?

Section C: String Formatting MCQs (5 Marks)

Choose the correct output for each code snippet.

16. (1 mark) What is the output of:

```
name = "Alice"
age = 25
print(f"{name:>10} is {age:03d} years old")
```

    a) ”Alice is 025 years old”
    b) ” Alice is 025 years old”
    c) ”Alice is 25 years old”
    d) ” Alice is 25 years old”

17. (1 mark) What is the output of:

```
num = 123.456789
print("{:.2%}".format(num))
```

    a) ”123.46%”
    b) ”12345.68%”

c) ”12346%”

d) ”123.46”

18. (1 mark) What is the output of:

```
data = {"name": "Bob", "score": 95}
print("{name}'s score is {score:b}".format(**data))
```

a) ”Bob’s score is 1011111”

b) ”Bob’s score is 95”

c) Error

d) ”Bob’s score is 0b1011111”

19. (1 mark) What is the output of:

```
x = 42
print(f"{x:>5o}")
```

a) ” 52”

b) ”00052”

c) ” 042”

d) ”42”

20. (1 mark) What is the output of:

```
value = 0.1234
print(f"|{value:+10.2f}|")
```

a) ”— +0.12—”

b) ”— +0.123—”

c) ”— 0.12—”

d) ”—+ 0.12—”

1 Python Programming Examination - Answer Key

Advanced Programming Division Detailed Solutions and Explanations

Section A: Algorithm Development (25 Marks)

Question 1: Anagram Generator Algorithm (5 marks)
Solution:

a) Algorithm:

```python
def find_valid_anagrams(input_word, word_dict):
    # Step 1: Preprocess word dictionary
    word_dict = set(word.lower() for word in word_dict)

    # Step 2: Create character frequency map
    char_freq = {}
    for char in input_word.lower():
        char_freq[char] = char_freq.get(char, 0) + 1

```

```python
    # Step 3: Generate anagrams using backtracking
    def backtrack(curr_word, char_count):
        if len(curr_word) == len(input_word):
            if curr_word in word_dict:
                valid_anagrams.add(curr_word)
            return

        for char in char_count:
            if char_count[char] > 0:
                char_count[char] -= 1
                backtrack(curr_word + char, char_count)
                char_count[char] += 1

    valid_anagrams = set()
    backtrack("", char_freq.copy())
    return valid_anagrams
```

b) Time Complexity: O(n!), where n is the length of input word

c) Space Complexity: O(n) for recursion stack

d) Optimizations:

- Use trie data structure for dictionary lookup (O(m) where m is word length)
- Implement word length filtering before anagram generation
- Use sorted character string as key for quick lookup

Question 2: Circular Buffer Algorithm (5 marks)
Solution:

a) Algorithm Implementation:

```python
class CircularBuffer:
    def __init__(self, size):
        self.size = size
        self.buffer = [None] * size
        self.head = 0  # Points to next write position
        self.tail = 0  # Points to oldest element
        self.count = 0  # Current number of elements

    def push(self, item):
        self.buffer[self.head] = item
        self.head = (self.head + 1) % self.size

        if self.count < self.size:
            self.count += 1
        else:
            self.tail = (self.tail + 1) % self.size

    def pop(self):
        if self.count == 0:
            raise IndexError("Buffer is empty")

        item = self.buffer[self.tail]
        self.tail = (self.tail + 1) % self.size
        self.count -= 1
        return item

    def is_full(self):
```

```python
        return self.count == self.size

    def is_empty(self):
        return self.count == 0
```

b) Time Complexity:

- Push: O(1)
- Pop: O(1)
- Space Complexity: O(n) where n is buffer size

Question 3: Longest Palindromic Subsequence (5 marks)
Solution:

a) Algorithm with Dynamic Programming:

```python
def longest_palindromic_subsequence(s):
    n = len(s)
    # Initialize dp table
    dp = [[0] * n for _ in range(n)]

    # Base case: single characters are palindromes
    for i in range(n):
        dp[i][i] = 1

    # Fill dp table
    for length in range(2, n + 1):
        for start in range(n - length + 1):
            end = start + length - 1

            if s[start] == s[end]:
                dp[start][end] = dp[start + 1][end - 1] + 2
            else:
                dp[start][end] = max(
                    dp[start + 1][end],
                    dp[start][end - 1]
                )

    # Reconstruct palindrome
    def reconstruct_palindrome():
        result = []
        i, j = 0, n - 1
        while i <= j:
            if s[i] == s[j]:
                if i != j:
                    result.append(s[i])
                else:
                    if i == j:
                        result.append(s[i])
                i += 1
                j -= 1
            elif dp[i + 1][j] > dp[i][j - 1]:
                i += 1
            else:
                j -= 1
        return ''.join(result + result[::-1])

    return dp[0][n-1], reconstruct_palindrome()
```

b) Time Complexity: $O(n^2)$

c) Space Complexity: $O(n^2)$

d) Space Optimization:

- Can be optimized to O(n) space by using two arrays
- Rolling array technique possible for space optimization

Question 4: Nested Dictionary Flattener (5 marks)
Solution:

a) Algorithm Implementation:

```python
def flatten_dict(nested_dict, parent_key='', separator='.'):
    items = []
    for key, value in nested_dict.items():
        new_key = f"{parent_key}{separator}{key}" if parent_key else key

        if isinstance(value, dict):
            items.extend(
                flatten_dict(value, new_key, separator).items()
            )
        elif isinstance(value, list):
            for i, item in enumerate(value):
                if isinstance(item, dict):
                    items.extend(
                        flatten_dict(
                            item,
                            f"{new_key}{separator}{i}",
                            separator
                        ).items()
                    )
                else:
                    items.append((f"{new_key}{separator}{i}", item))
        else:
            items.append((new_key, value))

    return dict(items)

# Example usage:
nested = {
    'a': 1,
    'b': {
        'c': 2,
        'd': {
            'e': 3
        }
    },
    'f': [{'g': 4}, 5, {'h': 6}]
}

# Result:
# {
#     'a': 1,
#     'b.c': 2,
#     'b.d.e': 3,
#     'f.0.g': 4,
#     'f.1': 5,
#     'f.2.h': 6
```

```
# }
```

b) Complexity Analysis:

- Time: O(n) where n is total number of elements
- Space: O(d) where d is maximum depth of nesting

Question 5: Custom String Sorting (5 marks)
Solution:

a) Algorithm Implementation:

```
def custom_string_sort(strings):
    # Create tuple pairs of (length, string) for sorting
    def get_sort_key(s):
        return (len(s), s.lower())

    # Using TimSort (Python's built-in sort)
    return sorted(strings, key=get_sort_key)

# Alternative implementation using quick sort
def quick_sort_strings(strings):
    if len(strings) <= 1:
        return strings

    pivot = strings[len(strings)//2]
    pivot_len = len(pivot)

    left = [s for s in strings if (len(s), s.lower()) <
            (pivot_len, pivot.lower())]
    middle = [s for s in strings if (len(s), s.lower()) ==
              (pivot_len, pivot.lower())]
    right = [s for s in strings if (len(s), s.lower()) >
             (pivot_len, pivot.lower())]

    return quick_sort_strings(left) + middle + \
           quick_sort_strings(right)
```

b) Complexity Analysis:

- Time: O(n log n) average case
- Space: O(n) for storing sorted array

c) Optimization Strategies:

- Cache string lengths
- Use insertion sort for small subarrays
- Implement parallel sorting for large datasets

Section B: Python Internals and Concepts (20 Marks)

6. List Comprehension vs Generator Expression (2 marks)

- List Comprehension:
  - Creates entire list in memory at once

  - Memory footprint: O(n) where n is number of elements
  - Example:

```
nums = [x*x for x in range(1000000)]
# Creates list of 1M elements immediately
```

- Generator Expression:
  - Creates iterator, generates one item at a time
  - Memory footprint: O(1)
  - Example:

```
nums = (x*x for x in range(1000000))
# Creates generator object, negligible memory
```

7. Python List Item Memory Footprint (2 marks)

- List overhead: 40 bytes (Python 3.8+)
- Per-item overhead: 8 bytes (64-bit system)
- Reference count: 8 bytes
- Object header: 16 bytes
- Example:

```
# For a list of integers:
my_list = [1, 2, 3]
# Total memory = 40 (list overhead)
#              + 3 * 8 (per-item overhead)
#              + 3 * 28 (int object size)
#              = 148 bytes
```

8. String Memory in Python 2 vs 3 (2 marks)

- Python 2:
  - ASCII strings: 1 byte per character + header
  - Unicode strings: 2 or 4 bytes per character
- Python 3:
  - All strings are Unicode
  - Latin-1: 1 byte per character
  - UCS-2: 2 bytes per character
  - UCS-4: 4 bytes per character
- String interning:
  - Python 3 interns strings at compile time
  - More aggressive interning than Python 2

9. Garbage Collection of Circular References (2 marks)

- Reference counting:
  - Primary mechanism

- Fails with circular references

- Generational garbage collector:
  - Three generations (0, 1, 2)
  - Cycle detector algorithm
  - Example:

```python
a = {}
b = {}
a['b'] = b
b['a'] = a
# Creates circular reference
# Detected by GC, not ref counting
```

10. Python Dictionary Implementation (2 marks)

- Hash table implementation
- Open addressing with random probing
- Collision resolution:
  - Perturb function
  - Quadratic probing
- Load factor maintained below 0.66
- Example:

```python
d = {}
# Initial size: 8 slots
# Resizes at 6 items (0.66 load factor)
for i in range(10):
    d[i] = i
# Triggers resize to 16 slots
```

11. Tuple Immutability Implementation (2 marks)

- Memory layout:
  - Fixed-size array of references
  - No over-allocation
  - Read-only memory pages
- Compared to lists:
  - No resize overhead
  - Smaller memory footprint
  - Better cache performance

12. String Interning in Python (2 marks)

- Automatic interning:
  - All length-0 and length-1 strings
  - String literals
  - Identifier-like strings

- Manual interning:

```
import sys
a = sys.intern("hello")
b = sys.intern("hello")
# a and b reference same object
```

13. Variable Scope in Nested Functions (2 marks)

- LEGB Rule:
  - Local
  - Enclosing
  - Global
  - Built-in
- Closure implementation:
  - Cell objects
  - Free variables table
- Example:

```
def outer(x):
    def inner():
        return x  # Creates closure
    return inner
# Memory overhead: cell object + function object
```

14. __slots__ vs Instance Dictionaries (2 marks)

- Regular Classes:
  - Use __dict__ for attribute storage
  - Dynamic attribute addition
  - Higher memory overhead
- __slots__ Classes:
  - Fixed attribute array
  - No dynamic attributes
  - Reduced memory footprint
- Example:

```
# With __dict__
class Regular:
    def __init__(self, x, y):
        self.x = x
        self.y = y
# ~144 bytes per instance

# With __slots__
class Optimized:
    __slots__ = ['x', 'y']
    def __init__(self, x, y):
        self.x = x
        self.y = y
# ~80 bytes per instance
```

15. Integer Caching in Python (2 marks)

- Default range: -5 to 256
- Implementation:
  - Small integer cache pool
  - Singleton objects
  - Immutable integers
- Example:

```python
a = 256
b = 256
print(a is b)  # True (cached)

x = 257
y = 257
print(x is y)  # False (not cached)
```

Section C: String Formatting MCQs (5 Marks)

16. Answer: b)

- ” Alice is 025 years old”
- Explanation:
  - :¿10 right-aligns with width 10
  - :03d zero-pads number to 3 digits

17. Answer: b)

- ”12345.68%”
- Explanation:
  - .2% formats as percentage
  - Multiplies by 100 and adds % symbol
  - Rounds to 2 decimal places

18. Answer: a)

- ”Bob’s score is 1011111”
- Explanation:
  - :b formats as binary
  - Doesn’t include ’0b’ prefix

19. Answer: a)

- ” 52”
- Explanation:
  - :¿5 right-aligns with width 5
  - o formats as octal without ’0o’ prefix

- 42 in octal is 52

20. Answer: a)

- ”— +0.12—”
- Explanation:
  - :+10.2f forces plus sign
  - Width of 10 characters
  - 2 decimal places
  - Right-aligned within the width