1. Multiple Choices Questions (45 Points)

After answering all knowledge questions, transfer your solution letter to the table below. Only one solution is correct for each answer option. Please mark your solution clearly:

Questions 1.1 to 1.10

	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
Answer	B	B	A	B	D	B	A	B	D	C
Points										

Questions 1.11 to 1.15

	1.11	1.12	1.13	1.14	1.15
Answer	C	C	B	B	C
Points					

Question 1.1 | Turing Machine (3 Points)

The Turing machine is a conceptual model of computation. All modern computers are based on the Von Neumann architecture, virtually a Turing machine. Which part of a modern computer is equivalent to the state table in a Turing machine?

- ☐ A: Memory
- ☐ B: Program
- ☐ C: Pseudocode
- ☐ D: CPU

Question 1.2 | Computer Memory (3 Points)

Which of the following is the first place a CPU will check when it needs data?

- ☐ A: Hard drive
- ☐ B: Cache
- ☐ C: Main memory
- ☐ D: Secondary memory

Question 1.3 | Machine Language (3 Points)

Which of the following statements is FALSE about machine language and high-level programming languages?

- ☐ A: Machine language code requires fewer resources to write and is easier to debug.
- ☐ B: High-level programming languages can not be understood by machines directly and require compiling.
- ☐ C: High-level programming language hides unnecessary details of machine languages for programmers.
- ☐ D: Machine language code is not portable across different computers (e.g., PC, Mac).

Question 1.4 | Algorithm (3 Points)

Which description below is NOT a characteristic of an algorithm in computer science?

- ☐ A: Steps to solve a task are put into a structure
- ☐ B: Some steps contain ambiguity
- ☐ C: All steps are executable
- ☐ D: The execution comes to an end eventually

Question 1.5 | Big-O Notation (3 Points)

Which statement below is FALSE about the big-O notation?

☐ A: It assumes the worst case and infinite size of the input

☐ B: It is also called asymptotic notation

☐ C: It can be used to represent both time and space complexity

☐ D: It denotes the lower bound of the complexity function

Question 1.6 | Recursion (3 Points)

What will be the return of *fMystery(x = 4)*?

```
def fMystery(x):
    if x <= 1:
        return 1
    elif x == 2:
        return 2
    else:
        return fMystery(x - 1) + fMystery(x - 2) + fMystery(x - 3)
```

☐ A: 3

☐ B: 7

☐ C: 9

☐ D: 10

Question 1.7 | Sorting (3 Points)

Which of the following statements is TRUE about sorting algorithms?

☐ A: Bubble sort is preferred over merge sort when the memory space is very limited.

☐ B: Early stopping mechanism can be applied to all sorting algorithms to improve efficiency at best-case scenario.

☐ C: Merge sort always executes faster than bubble sort no matter what input list is given.

☐ D: Complexity of quick sort does not vary as long as we always choose the first element of a (sub-)list as the pivot.

Question 1.8 | Search (3 Points)

Which of the following statements is FALSE about searching algorithms?

- ☐ A: Linear search is an instance of brute-force algorithm.
- ☐ B: Binary search always runs faster than linear search even when the input list is unsorted and small.
- ☐ C: Binary search follows the divide-and-conquer paradigm.
- ☐ D: Hash table trades space for time and can find value more efficiently than either linear or binary search.

Question 1.9 | Tree (3 Points)

Which statement below is TRUE about tree, the data structure?

- ☐ A: All real-world problems can be solved by answering a series of yes or no questions and represented by a tree
- ☐ B: There is an inherent tree structure in bubble sort
- ☐ C: There is no inherent tree structure in backtracking
- ☐ D: Traversing on tree nodes is the process of finding solution

Question 1.10 | P and NP Problems (3 Points)

Which of the following statements about polynomial problems is TRUE?

- ☐ A: For any real-world problems, algorithmic solutions always exist, but are not necessarily findable.
- ☐ B: The classic traveling salesman problem can be eventually reduced to a polynomial problem.
- ☐ C: Nondeterministic algorithms can be used to solve some hard problems in polynomial time.
- ☐ D: Every problem whose solution can be quickly verified can also be quickly solved.

Question 1.11 | Dynamic Programming (3 Points)

What is memoization in the context of dynamic programming?

- ☐ `A: A method of analyzing the time complexity of algorithms.`
- ☐ `B: A technique to write memory-efficient programs.`
- ☐ `C: A way to avoid solving subproblems by storing their solutions and reusing them.`
- ☐ `D: A process of converting recursive algorithms into iterative ones.`

Question 1.12 | OOP I (3 Points)

Which of the following best describes the advantage of Object-Oriented Programming?

- ☐ `A: OOP eliminates the need for documentation since the code is self-explanatory.`
- ☐ `B: OOP is a programming paradigm that reduces the time complexity of algorithms by space-time tradeoff.`
- ☐ `C: OOP promotes code reusability and modularity through the use of classes.`
- ☐ `D: OOP requires less memory compared to procedural programming.`

Question 1.13 | OOP II (3 Points)

What is method overriding in Object-Oriented Programming?

- ☐ `A: A subclass overloading the built-in operator.`
- ☐ `B: A subclass providing its own version of a method that is already defined in the superclass.`
- ☐ `C: A subclass calling the superclass method with new arguments.`
- ☐ `D: A subclass using the method of the parent class without any changes.`

Question 1.14 | Polymorphism (3 Points)

Which of the following best describes polymorphism in Object-Oriented Programming?

- ☐ A: It refers to the ability to define multiple methods with the same name but different numbers of parameters within a single class.
- ☐ B: It allows a subclass to re-implement a superclass’s functions, providing different behavior.
- ☐ C: It enables a class to inherit attributes and methods from multiple parent classes.
- ☐ D: It restricts access to certain attributes and methods to improve security.

Question 1.15 | Decorators in OOP (3 Points)

Consider the following Python code:

```
class Employee:
    def __init__(self, name):
        self._name = name

    # Getter
    @property
    def name(self):
        return self._name

    # Setter
    @name.setter
    def name(self, new_name):
        self._name = new_name
```

What is the primary purpose of the `@property` and `@name.setter` decorators in the class above?

- ☐ A: They improve performance by optimizing attribute retrieval in Python’s memory management.

☐ B: They enable method overloading so multiple functions with the same name can exist.

☐ C: They define getter and setter methods for attributes, allowing controlled access and modification.

☐ D: They prevent the modification of attributes, making them read-only.

2. Short Answer Questions (20 Points)

Question 2.1 | Computer Memory (10 Points, 3/3/4 points each)

1. Give a function name to the following command: ______________________________

What are the expected outputs for register 2 and 3?

1 DEB 3 2 3

2 INC 2 1

3 END

ADD [3][2]: destructive add Reg 3 to Reg 2 | Reg 2: 5 | Reg 3: 0

2. Give a function name to the following command: ______________________________

What are the expected outputs for register 4 and 5?

1 DEB 5 1 2

2 DEB 4 3 4

3 INC 5 2

4 END

MOVE [4][5]: move (and clear) Reg 4 to Reg 5 | Reg 4: 0 | Reg 5: 4

3. Give a function name to the following command: ______________________________

What are the expected outputs for register 1, 3, 4?

```
1 DEB 3 1 2
2 DEB 4 2 3
3 DEB 1 4 6
4 INC 3 5
5 INC 4 3
6 DEB 4 7 8
7 INC 1 6
8 END
```

COPY [1][3]: copy data in Reg 1 to Reg 3 | Reg 1: 1 | Reg 3: 1 | Reg 4: 0

Question 2.2 | Complexity Analysis (10 Points, 2 points each)

1. Analyze the time complexity of the following function in Big-O notation:

```
def int_to_binary(x):
    """Assumes x is a nonnegative int
    Returns a binary string representation of x."""

    digits = '01'
    if x == 0:
        return '0'
    result = ''
    while x > 0:
        result = digits[x%2] + result
        x = x//2
    return result

Output = int_to_binary(n)
```

O(log n)

2. Analyze the time complexity of the selection sort in Big-O notation:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
```

O(n^2)

3. Analyze the time complexity of the following function in Big-O notation:

```
def mystery_function(n):
    sum = []
    i, j = 1, 2
    while i <= N:
        while j <= N+1:
            sum.append(i+j)
            i *= 3
            j *= 2
```

O(log n)

4. Analyze the time complexity of the following function in Big-O notation:

```
def method1(n):
    total, i = 0, 1
    while i*i < n:
        total += 1
```

```
        i += 1
    return total
```

O(sq rt n)

5. Analyze the SPACE complexity of the following function with respect to power n:

```
def power(base, n):
    if n == 0:
        return 1
    else:
        return base * power(base, n - 1)
```

O(n)

3. Programming Questions (35 Points)

Please write your programming solution as clearly as possible.
Try to add comments to explain each code segment.

Question 3.1 | Recursion (5 Points)

The harmonic series is the reciprocals of the positive integers, which is as the following,

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, 1/n, \ldots$$

Write a function that calculates the sum of the harmonic series up to *n* items using **recursion**. (Partial credits will be given for other implementations)

```
def harmonic_sum(n):
    s = 0
    if n == 1:
        return 1
    else:
        s = 1/n + harmonic_sum(n-1)
    return s
```

Question 3.2 | Tree Traversal (10 Points)

1. Traversal Order (6 Points, 2 points each)

What will be the expected sequence of different traversal approaches on this tree?

`Preorder traversal: 1 - 2 - 4 - 5 - 7 - 8 - 3 - 6`

`Inorder traversal: 4 - 2 - 7 - 5 - 8 - 1 - 3 - 6`

`Postorder traversal: 4 - 7 - 8 - 5 - 2 - 6 - 3 - 1`

2. Traversal Methods (4 Points)

Fill in the methods below: preorder is given as an example.

```
def preorder(r):
    if r == None:
        return
    else:
        return str(r.value) + preorder(r.left) + preorder(r.right)

def inorder(r):
    if r == None:
        return
    else:
        return inorder(r.left) + str(r.value) + inorder(r.right)

def postorder(r):
    if r == None:
        return
    else:
        return postorder(r.left) + postorder(r.right) + str(r.value)
```

Question 3.3 | Merge Sort using OOP (20 Points)

```
class Node():

    def __init__(self, num_lst):
        self.numbers = num_lst

    def get_numbers(self):
        return self.numbers

    def set_numbers(self, new_lst):
        self.numbers = new_lst

    def __getitem__(self, idx):
        try:
            return self.numbers[idx]
        except IndexError:
            print("Index Error")
            return None

    def length(self):
        # print("length is:", self.numbers)
        return len(self.numbers)

    def __str__(self):
        return "{}".format(self.numbers)

def divide(self):      # 3 Points
     # write your code here
     Mid = len(self.numbers) // 2
     return Node(self.numbers[:mid]), Node(self.numbers[mid:])

def merge(self, other):     # 5 Points
     # write your code here
     Lst = []
     I = 0
     J = 0
     While (i<len(self.numbers)) and (j<len(other.numbers)):
```

```
        If self.numbers[i] < other.numbers[j]:
             lst.append(self.numbers[i])
             I += 1
        Else:
             lst.append(other.numbers[j])
             J += 1
   If i == len(self.numbers):
        While (j<len(other.numbers):
             lst.append(other.numbers[j])
             J += 1
   Else:
        While (i<len(self.numbers):
             lst.append(self.numbers[i])
             i += 1
   Return Node(lst)
```

Ver. B Question 3.3 | Quick Sort using OOP (20 Points)

```
class Node():

    def __init__(self, num_lst):
        self.numbers = num_lst

    def get_numbers(self):
        return self.numbers

    def set_numbers(self, new_lst):
        self.numbers = new_lst

    def __getitem__(self, idx):
        try:
            return self.numbers[idx]
        except IndexError:
            print("Index Error")

    def length(self):
        # print("length is:", self.numbers)
        return len(self.numbers)
```

```
def __str__(self):
    return "{}".format(self.numbers)

def add_elements(self, numbers):
    try:
        self.numbers.extend(numbers)
    except TypeError:
        self.numbers.append(numbers)

def partition(self, idx=0):
    pivot = self.numbers[idx]
    seq = self.numbers[:idx] + self.numbers[idx+1:]
    low =  Node([x for x in seq if x <= pivot])
    high = Node([x for x in seq if x > pivot])

    return low, Node([pivot]), high
```