Question 1 [10 marks]

This question is about MIPS programming and function calls. Translate the following Python code faithfully into MIPS. Make sure you follow the MIPS function calling and memory usage conventions as discussed in the lectures. Use only instructions in the MIPS reference sheet. Notice that there is no code calling the function, thus the answer should involve only instructions executed by the callee.

Python Code	MIPS Code

`def my_function(n, m):`	`my_function:`
	`    addi $sp, $sp, -8`
	`    sw $ra, 4($sp)`
	`    sw $fp, 0($sp)`
	`    addi $fp, #sp, 0`

`    if n == 0:`	`lw   $t0, 8($fp)        # arg n`
	`bne  $t0, $0, else      # goto else if n != 0`

`        return m`	`lw   $t0, 12($fp)       # arg m`
	`addi $v0, $t0, 0        # return m`
	
	`lw   $fp, 0($sp)`
	`lw   $ra, 4($sp)`
	`addi $sp, $sp, 8`
	`jr   $ra`

`    else:`
`        return m//n`
	`else:`
	
	`  lw   $t1, 12($fp)      # arg m`
	`  lw   $t2, 8($fp)       # arg n`
	`  div $t1, $t2`
	`  mflo $t0.              # t0 = m//n`
	`  addi $v0, $t0, 0       # return`
	
	`  lw   $fp, 0($sp)`
	`  lw   $ra, 4($sp)`
	`  addi $sp, $sp, 8`
	`  jr   $ra`

- 3 marks for correct function entry
- 3 marks for correct function exit/returning
- 2 marks if/else handling
- 2 marks logic of business (div)

Question 2 – Array-based structures [11 marks = 3 + 3 + 3 + 2]

This question is about Array-based structures. The partial implementation below is from a Queue whose underlying array is automatically resizable. The Queue uses the space of the array efficiently, by wrapping around the front and the rear indices (i.e. a circular queue). In addition, it doubles the size of the underlying array when appending to a Queue that is already using all the space available. The partial implementation is as follows:

```
class Queue:

    def __init__(self):
        self.array = build_array(10)
        self.front = 0
        self.rear = 0
        self.count = 0

    def is_full(self):
        return False

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

    def __len__(self):
        return self.count

    def append(self, new_item):
        if self.count == len(self.array):
            self.__resize__()
        self.array[self.rear] = new_item
        self.rear = (self.rear+1) % len(self.array)
        self.count+=1
```

(a) Implement the method `__resize__(self)`, which is used by the `append` function. This method should double the size of the underlying array. It should also, if necessary, re-arrange the values of the instance variables.

```
def __resize__(self):
        temp_array = build_array(len(self.array)*2)
        index = self.front
        index_new_array = 0
        for _ in range(self.count):
            temp_array[index_new_array] = self.array[index]
            index = (index+1) % len(self.array)
            index_new_array+=1
        self.front = 0
        self.rear = len(self.array)
        self.array = temp_array
```

- Correct handling of instance variables - 1 mark
- Correct doubling array and copying items - 1 mark
- Correct handling of circularity - 1 mark

3

(b) Implement the method `serve(self)`, which serves an item out of the Queue. This method never modifies the size of the underlying array, and raises an `Exception` if empty.

```
def serve(self):
    assert not self.is_empty(), "Queue␣is␣empty"
    item = self.array[self.front]
    self.front = (self.front + 1) % len(self.array)
    self.count -= 1
    return item
```

- Correct empty check either (assert or raise) - 1 mark
- Correct handling of circularity - 1 mark
- Correct business of serving - 1 mark

(c) Implement the method `__str__(self)`, which returns a string representing the Queue, including all elements, separated by comma, from the front to the rear. For example, a Queue with two elements 1 and 2, at the front and the rear respectively, is represented by the string `"[1,2]"`. An empty Queue will be `"[]"`.

```
def __str__(self):
    index = self.front
    ans = ""
    for _ in range(self.count):
        ans += (str(self.array[index]) + ',')
        index = (index+1) % len(self.array)
    return ans
```

- Correct print on empty structure - 1 mark
- Correct string handling - 1 mark
- Correct handling of circularity - 1 mark

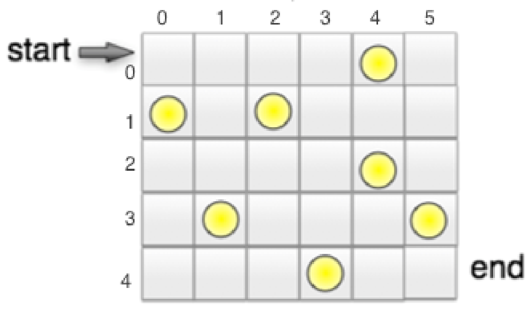
(d) In Big O notation what is the best and worst case for appending to a Queue with n elements and when do the cases occur. Explain by giving an example for each case.

Best case is $O(1)$ and occurs when there is no need to resize, Worst case $O(n)$ when array is full.

- Correct best case and when - 1 mark
- Correct worst case and when - 1 mark

Question 3 – Dynamic Programming [9 marks = 3 + 6]

This question is about Dynamic programming. Seven coins are placed on a 5 × 6 board (see figure below). A robot located in the square `[0, 0]`, marked as start, needs to collect as many coins as possible and bring them to the end square. At any time, the robot can only move one square to the right or one square down of its current location, always picking up any coins found in a square it visits. Diagonal moves, as well as moving up or moving left are not permitted.



An instance of the problem is given by an object of the class below.

```
import numpy as np
class RobotBoard:

    def __init__(self, number_of_rows, number_of_columns):
        # creates an empty board
        self.board = np.zeros((number_of_rows, number_of_columns))

    def number_of_rows(self):
        return self.board.shape[0]

    def number_of_columns(self):
        return self.board.shape[1]

    def place_coin(self, row, column):
        assert 0 < row <  self.number_of_rows(), "Non-existing␣row"
        assert 0 < column <  self.number_of_columns(), "Non-existing␣column"
        self.board[row, column] = 1

    def get_value(self, row, column):
        assert 0 < row <  self.number_of_rows(), "Non-existing␣row"
        assert 0 < column <  self.number_of_columns(), "Non-existing␣column"
        return self.board[row, column]
```

Notice that `get_value` will return a 0's or 1. Ones represent coins and zeroes represent empty spaces.

(a) Complete the sequence below, which gives an optimal path to be followed by the robot for the instance given in the figure.

```
[0,0] -- [1,0] -- [1, 1] -- [1, 2] -- [1, 3] -- [1, 4] -- [2, 4] -- [3, 4]
-- [3, 5 ] -- [4,5]
```

- Sequence respects rules - 1.5 marks; Sequence picks up 4 coins - 1.5 marks

3

(b) Considering the rules of this robot we need to determine the maximum number of coins that can be collected for any possible board of arbitrary size $n \times m$. The input is given by an instance of the class `RobotBoard`. Write a python function `maximum_value(self)` as part of the `RobotBoard` class. This methoduses the dynamic programming approach to determine the maximum number of coins that can be collected from the board.
For example, the board given in the first part of the question is instantiated:

```
my_board = RobotBoard(5, 6)
my_board.place_coin(0, 4)
my_board.place_coin(1, 0)
my_board.place_coin(1, 2)
my_board.place_coin(2, 4)
my_board.place_coin(3, 1)
my_board.place_coin(3, 5)
my_board.place_coin(4, 3)
sol = my_board.maximum_value()
print(sol)
```

will print 4.

```
def maximum_value(self):
        (n, m) = (self.number_of_rows(), self.number_of_columns())
        table = np.zeros((n, m))
        for i in range(1, n):
            table[i, 0] = table[i-1, 0] + self.get_value(i, 0)
        for i in range(1, m):
            table[0, i] = table[0, i-1] + self.get_value(0, i)
        for i in range(1, n):
            for j in range(1, m):
                table[i,j] = max(table[i-1,j], table[i,j-1]) + self.get_value(
        return table[n-1, m-1]
```

- Evidence of understanding the idea of DP - 1 mark
- Correct initialisation of first row and column - 2 marks
- Correct update of cells (recurrence) - 3 marks

Question 4 – Sorting Algorithms [9 marks = 3 + 3 + 3]

Consider the BubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort and HeapSort algorithms we have seen in the lectures, for sorting an array of length N. For those we have seen several versions (e.g. Bubble Sort) use the most efficient version we have seen.

(a) Name those (if any) that are not stable, and briefly explain why they are not stable.

Selection sort: when swapping element X for the minimum Y, another element with value X might be positioned in between. This happens when sorting `[2a,2b,1]` returns `[1,2b,2a]`

Quicksort: during partition the initial and final swap with the pivot can leave elements out of original relative order. This happens also when sorting `[2a,2b,1]` with mid element as pivot also returns `[1,2b,2a]`.

HeapSort: each `get_max` will swap the bottom-right element to the root, and then sink it. That swap can leave elements out of original relative order.

- 1 mark each correct, with explanation.

(b) Name those (if any) that run in worst-case time O(N log N), and briefly explain how they manage to take O(N log N).

MergeSort: the splitting by half creates a binary tree of depth $logN$, where the amount of work done whole splitting is constant. It then goes back merging N elements (almost) at each level of this tree, which means at each level traverses all N elements, for each doing a constant amount of work. Thus, it is $N * logN$

HeapSort: it first adds each element. This takes either $N * logN$ time (each element might need to sink to the bottom of the heap). Then we need to do N `get_max` operations, each taking $logN$ time. so we get $N * logN$ in total.

- 1.5 each with correct explanation.

(c) Names those (if any) that run in best-case time O(N), and briefly explain how they manage to take O(N).

BubleSort: if the list is already sorted, the first traversal of the N elements (which only makes constant operations) notices there are no swaps and stops the outer loop.

InsertionSort: if the list is already sorted, when inserting each of the $N-1$ elements (which only makes constant operations) it notices it is sorted and does not even start the inner loop.

- 1.5 each with correct explanation.

9

Question 5 – Hashing [8 marks]

You have started coding a HashTable as follows:

```
class MyHashTable:

    def __init__(self, size):
        self.table_size = size
        self.array = build_array(self.table_size)
        self.count = 0
```

Assume you need to choose a hash function for your hash table. Each key to be hashed is a sequence of N integers, such as [10,3,5,3,20]. For example, a `(key, value)` pair to be stored could be `([10,3,5,3,20], "Introduction to CS")`. You are given the following three possibilities to choose from.

(a) Returns the value of `random.randint(0, self.table_size-1)` (i.e., a random integer between 0 and the size of the table).

(b) Returns the minimum value in the sequence to be hashed (3 in our example above) mod size of the table.

(c) Returns the multiplication of all values in the sequence to be hashed (10*2*5*3*20 in our example above) mod size of the table.

For each function explain briefly what are the disadvantages. Rank the three possibilities from best to worst.

The worst is `random.randint(0, self.table_size)` as it cannot be used: it returns different values for the same key – i.e., it is not a mathematical function.

The minimum value can be used as hash function but is not the best, as it does not spread the hash values very well. This is because it disregards all other integers in the key. Thus, any two keys with a single number in common (the minimum one) will have the same hash value regardless of the value of the remaining $N-1$ elements (say `[1,2,3]` and `[17,21,1]`)

The best is the last one, as while `[1,2,3]` and `[3,2,1]` will have the same hash value (multiplication is commutative), all numbers contribute to the final value).

- 5 marks correct ranking
- 1 mark for each correct explanation (advantages, disadvantages) – total 3 marks

Question 6 – Hash Tables [7 marks = 5 + 2]

Consider a hash table implemented with an array of size 7.

(a) Show in the figure below (right) the final state of the array after inserting the numbers, 7, 3, 10, 5, 4, and 11. Collisions are resolved using linear probing with the hash function $h(k) = k\%7$, provided in the table below.

key	h(key)
7	0
3	3
10	3
5	5
4	4
11	4

0	7
1	11
2	
3	3
4	10
5	5
6	4

- 5 marks correct solution
- substract 1.5 for each mistake – consequentially

(b) What is the load of your hash table once all the numbers have been inserted?

The load is 6/7

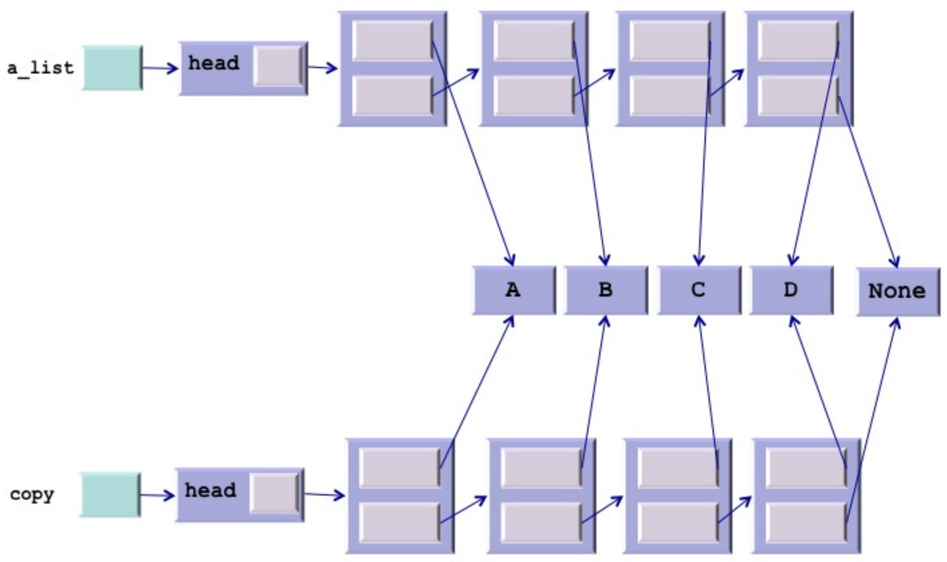
- 2 marks correct answer

7

Question 7 – Linked Lists [9 marks = 6 + 3]

Consider the following linked `List` class, which you are in the process of defining:

```
class Node:

    def __init__(self, item=None, link=None):
        self.item = item
        self.link = link

class List:

    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None
```



(a) Define a method `copy(self)` **within the** `List` **class** that returns a new linked list containing a copy of the nodes in `self`, in the given order and without modifying `self`. For example, given `a_list` with elements A,B,C, and D in the Figure above, the call to `a_list.copy()` will return the new list `copy` also shown in the figure, leaving `a_list1` unchanged. If `a_list` is empty, `a_list.copy()` will return a new empty list. Do not assume the existence of other methods beyond those defined above.

```
def copy(self):
    copy = List()
    current = self.head
    previous = None
    while current is not None:
       if previous is None:
            copy.head = Node(current.item)
       else:
            previous.link = Node(current.item)
       previous = current
       current = current.link
    return copy
```

- 1 marks for creating a new instance to return
- 2 marks for creating a new node for each element copied
- 3 marks for correct handling of links

6

(b) Below is a partial implementation of an Iterator for the `List` class defined in part *a*.

```
class MyLinkedListIterator:

    def __init__(self, head):
        self.current = head
```

Complete this iterator implementation by writing down the methods `__next__(self)` and `__iter__(self)`.

```
    def __iter__(self):
        return self

    def __next__(self):
        if self.current == None:
            raise StopIteration
        else:
            item_required = self.current.item
            self.current = self.current.next
            return item_required
```

- 1 mark for `__iter__` returning self
- 1 mark StopIteration
- 1 mark correct current update

Question 8 – Data Structures and Complexity [8 marks]

Consider an **unsorted list** of N integers. Use Python to write a function that takes the list as a parameter and prints the elements that appear more than once. For example, for a list [6, 10, 6, 11, 6, 4, 5, 4], the algorithm will print 6 and 4. Your implementation should have worst case $O(n)$ time complexity, where n is the size of the list. You can assume and use any sorting algorithm we have seen (`bubble_sort`, `insertion_sort`, `selection_sort`, `merge_sort`, `quick_sort` and `heap_sort`), and any data structure we have seen (`List`, `Stack`, `Queue`, `HashTable`, `Heap`) – without writing them, including any methods we have defined in class as part of each structure. **Note:** Do not worry about exact method names or parameters as long as they are indicative, we are interested in your algorithmic reasoning, not syntax details.

```
def duplicates(a_list):
    hash_table = dict()
    for x in a_list:
        try:
            hash_table[x]+=1
        except KeyError:
            hash_table[x] = 1
    for x in hash_table.keys():
        if hash_table[x] >=2:
            print(x)
```

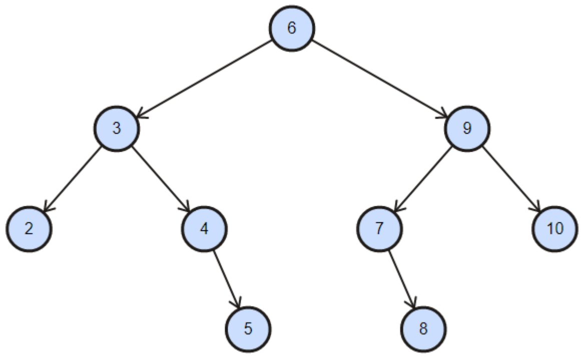
- 8 marks for a solution that works in worst case O(n), using a Dictionary
- 4 marks for a solution that works performing in more than linear time
- Do not focus on syntax, but on use of Data Structures.

8

Question 9 – Binary Trees [11 marks = 5 + 4 + 2]

Consider the partial implementation of `BinarySearchTree` class given below, which uses the `TreeNode` class:

```
class TreeNode:

    def __init__(self, key, value, left=None, right=None):
        self.item = (key, value)
        self.left = left
        self.right = right

class BinarySearchTree:

    def __init__(self):
        self.root = None

    def is_empty(self):
        return self.root is None

    def LCA(self, key1, key2):
        return self.LCA_aux(self.root, key1, key2)
```



(Only keys depicted)

If the method `LCA` is applied to the binary search tree above, `LCA(self, 2, 5)` will return 3, `LCA(self, 7, 4)` will return 6, and `LCA(self, 7, 8)` will return 7.

(a) The Lowest Common Ancestor (LCA) of two nodes x and y in a binary tree is the node with the lowest key that has both x and y as descendants. Assuming `key1` and `key2` are both integers, implement the method `LCA_aux(self, current, key1, key2)` called from the method `LCA(self, key1, key2)` given above. The method returns the key of the lowest common ancestor of Nodes containing `key1` and `key2`. You can safely assume that `key1` and `key2` do exist, in other words, the precondition of the method is that `key1` and `key2` are part of the Binary Search Tree. See the examples above and notice that a node is its own ancestor as shown by the last example.

```
def LCA_aux(self, current, key1, key2):
    if current is not None:
        if current.key > key1 and current.key > key2:
            return LCA_aux(current.left, key1, key2)
        elif current.key < key1 and current.key < key2:
            return LCA_aux(current.right, key1, key2)
        else:
            return current.item
```

- 1.5 left exploration (correct recursion and condition)
- 1.5 right exploration (correct recursion and condition)
- 1 marks for correct base case
- 1 marks for checking current is not `None`

5

(b) Implement the method `traverse_preorder(self)` within the `BinarySearchTree` class above. The method should return a Python list containing all the keys in the tree, as given by a pre-order traversal. For example, for the BST in the figure above the method returns `[6, 3, 2, 4, 5, 9, 7, 8, 10]`. You can use the usual `append` method to append an element to the end of the Python list. For an empty heap, it naturally returns the empty list.

```
def traverse_preorder(self):
    a_list = []
    a_list = self._traverse_preorder(self.root, a_list)
    return a_list

def _traverse_preorder(self, current, a_list):
    if current is not None:
        a_list.append(current.item)
        a_list = self._traverse_preorder(current.left, a_list)
        a_list = self._traverse_preorder(current.right, a_list)
    return a_list
```

- 1 mark for correct setup of the first recursion
- 1 mark for preorder feature (root, left, right)
- 0.5 mark base case
- 1.5 mark for correctly returning and assigning the list in each recursive call

(c) What is the best-case and worst-case time complexity of the most efficient implementation of `traverse_preorder(self)`? Explain. No explanation, no marks.

Best and worst case are both O(N), because you have to visit every node exactly once.

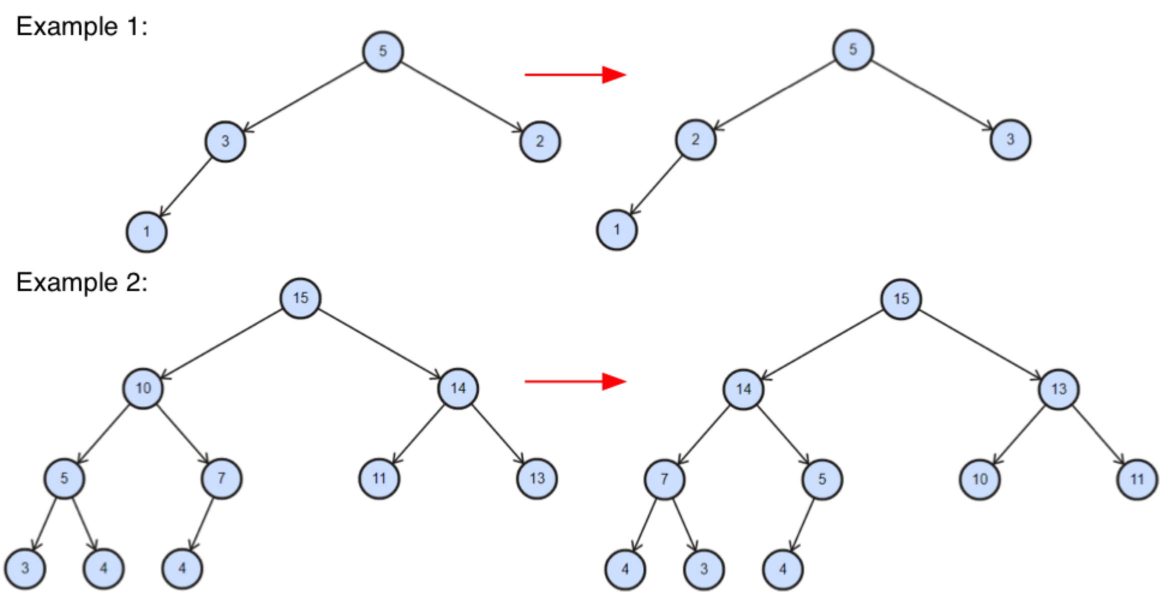
- 1 mark correct best case, with explanation
- 1 mark correct worst case, with explanation

Question 10 – Heaps [10 marks = 6 + 4]

Consider the partial implementation of `MaxHeap` class given below:

```
class MaxHeap:

    def __init__(self):
        self.array = build_array(50)
        self.count = 0

    def swap(self, i, j):
        self.array[i], self.array[j] = self.array[j], self.array[i]

    def largest_child(self, k):
        if 2 * k == self.count or self.array[2 * k][0] > self.array[2 * k + 1][0]:
            return 2 * k
        else:
            return 2 * k + 1

    def sink(self, k):
        while 2 * k <= self.count:
            child = self.largest_child(k)
            if self.array[k][0] >= self.array[child][0]:
                break
            self.swap(child, k)
            k = child
```

The underlying array stores tuples `(key, value)`. The examples below depict keys only.



0

(a) Implement the method `swap_children(self)`, which modifies the heap structure to swap the immediate children of every node (if any), sinking as necessary to keep the structure a valid max heap. For example, in the figures above, if the method is applied to the max heap on the left, it will modify it to be the one on the right.

```
def swap_children(self):
    node = 1
    while node*2 + 1 <= self.count:
      self.swap(node*2, node*2+1)
      self.sink(node*2)
      self.sink(node*2+1)
      node+=1
```

- 1 mark for correct initialization of root node
- 2 mark for correct loop condition
- 3 mark for correct swap, sink, and node increment

(b) Draw the array behind a heap (using the convention seen in the lectures) after inserting the keys 15, 32, 17, 51, 29, 10, 23 into the max heap and then deleting 51 and 32 (no need to depict values, only keys).

```
[None, 29, 17, 23, 15, 10]
```

- 4 marks for correct content
- -1 for each mistake

Question 11 – Classes, Objects, and Namespaces [8 marks]

Examine the following Python code:

```
class Car:

    numberOfTyre = 4
    steeringLocation = "Left"
    engineLocation = "Front"
    regionCode = 1770174

    def __init__(self, brand, enginePower):
        self.brand = brand
        self.enginePower = enginePower

    def setSeatNumber(self, numberOfSeat):
        self.seatNumber = numberOfSeat

    def setColour(self, colour):
        Car.colour = colour

car2 = Car("Nissan", 2400)
car2.setColour("Green")
car2.engineLocation = "Back"
car2.numberOfTyre = 6
car1 = Car("Toyota", 1000)
car1.setSeatNumber(3)
car1.steeringLocation = "Right"
car1.colour = "Red"
car1.regionCode = 1000000
print(car2.colour) #1
print(car1.colour) #2
print(car2.steeringLocation) #3
print(car1.engineLocation) #4
Car.numberOfTyre = 3
car1.enginePower = 500
print(car2.numberOfTyre) #5
print(car1.numberOfTyre) #6
print(car2.enginePower) #7
print(car1.seatNumber) #8
```

(a) Provide the result of each print statement (marked with comments from #1 to #8) – **next to the comment above**. If the results is an error, explain why assuming the execution will continue after executing the Python code above.

Green
Red
Left
Front
6
3
2400
3

- 1 mark per correct print

8