MIDTERM 2

SAMPLE SOLUTION

Question 1 [3 marks]

(a) **Given that the following assignment statement has been executed:**

```
L1 = [“Yahoo”, “Reddit”, 11, 7]
```

Which line of code does NOT do exactly the same things as the other three? Circle one answer.

```
(A) L2 = L1[-4:4]
(B) L2 = L1[:]
(C) L2 = list(L1)
(D) L2 = L1[0] + L1[1] + L1[2] + L1[3]
```

(b) **Consider the following piece of code:**

```
a_set = {'a','b','c','d'}
b_set = {'c','d','e','f'}
b_set = a_set.union(b_set)
b_set.add('c')
```

Which of the following is a possible representation of the value of `b_set`? Circle one answer.

```
(A) {'a','b','c','d','c','d','e','f'}
(B) {'c','d'}
(C) {'a','b','c','d','e','f'}
(D) {'a','b','e','f'}
(E) None of the above
```

(c) **Given the data structure:**

```
mov_info = (("Disney", 1998), (39.9, "Pixar"),
[133032, "Paramount"])
```

Which of the following statements would result in an error? Circle one answer.

```
(A) mov_info += ("Universal",)
(B) mov_info[2] = ("MGM", 2009)
(C) mov_info *= 3
(D) mov_info[2][0] += 2000
(E) None of the above
```

Question 2 [6 marks]

Indicate what the code will print in its adjacent box.

(a) [3 marks]

```
cellphone = {'iPhoneXs':'$824',
             'Samsung S10':'$738',
             'Google Pixel 3':'$700'}

cellphone['Huawei 9'] = '$673'

print('$673' not in cellphone)
print(cellphone['iPhoneXs'])
```

```
True
$824
```

Marking Scheme:

- True (2 marks: 1 mark understanding that the output is either True or False, 1 mark for the correct answer).
- $824 (1 mark, deducted 0.5 for formatting errors e.g. 824, "824" "824)

(b) [3 marks]

```
number = 10
divisor = 1
list_divisors = []

while divisor < number:
    if number % divisor == 0:
        list_divisors += [divisor]
    divisor += 1

print(list_divisors)
```

```
[1, 2, 5]
```

Marking Scheme:

- 1 mark: the correct values
- 1 mark: the correct order of the outputs (1 mark given for this part if the student made a small mistake in the outputs e.g. [1,2], [2,5])
- 1 mark: correct format of the output in the form of a list (0 for incorrect format of outputs e.g. [[1],[2],[5]])

Question 3. [6 marks]

Write a function called `count_duplicates` that takes a dictionary as an input parameter and **returns a list of the values** that appear two or four times.

```
def count_duplicates(d):
    '''
    (dict)->list

    Returns a list of values from the input dictionary that
    occur two or four times.

    Examples:
    >>> count_duplicates({'R': 1, 'G': 2, 'B': 2, 'Y': 1,\
    'P': 3})
    [1,2]

    >>> count_duplicates({"cheese": "donut", 1: "candy",\
    2: "donut", "carrot": "potato", "candy":"potato", \
    "donut":"potato", "potato":"potato"})
    ['donut', 'potato']

    '''
    occurrences = {}

    for dict_value in d.values():
        if dict_value not in occurences:
            occurences[dict_value] = 0

        occurences[dict_value] += 1

    dup_list = []
    for dict_value, occurs in occurences.items():
        if occurs==2 or occurs==4:
            dup_list.append(dict_value)

    return dup_list
```

Marking Scheme

- 1 mark: docstring (type contract, function description, example(s))
- 1 mark: access values of dictionary
- 1 mark: count value occurrences
- 1 mark: check if values occur two or four times
- 1 mark: add values to the return list
- 1 mark: no other errors (no extra values in the return list, etc.); i.e. correct return value

Question 4 [9 marks]

(a) Examine the following recursive function.

```
def foo(bar):
    '''
    (int) -> int
    '''
    if bar == 0:
        return bar
    else:
        return bar * foo(bar % 2)
```

Indicate the return values of `foo()` for the specified values of `bar` in the table below. **[1.5 marks]**

- 0.5 for each correct answer

bar	foo(bar)	Return Value
0	foo(0)	0
10	foo(10)	0
11	foo(11)	does not return – infinite recursion. Python will end execution due to exceeding max recursion depth.

You may use the space below for rough work, but only the table will be marked.

(b) Examine the following recursive function.

```
def foo_too(bar):
    '''
    (int) -> int
    '''
    if bar == 0:
        return bar + 1
    else:
        return bar * foo_too(bar // 2)
```

Note: in Python 3, the integer divide operator (denoted with //) takes the floating-point division result and rounds it down to the nearest integer value.

Indicate the return values of `foo_too()` for the specified values of `bar` in the table below. **[2.5 marks]**

- 0.5 for each correct answer

bar	foo_too(bar)	Return Value
0	foo_too(0)	1
1	foo_too(1)	1
2	foo_too(2) 2 * foo_too(1)	2
10	foo_too(10) 10*foo_too(5) 10*5*foo_too(2)	100
11	foo_too(11) 11*foo_too(5) 11*5*foo_too(2)	110

You may use the space below for rough work, but only the table will be marked.

(c) Without using recursion, write a function `foo_too_2()` which produces the same return value as `foo_too()` for **any** input `bar` of type `int`. **[2 marks]**

```
def foo_too_2(bar):
    '''
    (int) -> int
    '''

    if bar == 0:
        return 1

    curr_multiplier = bar
    result = bar
    while curr_multiplier > 1:
        curr_multiplier = curr_multiplier // 2
        result *= curr_multiplier
    return result
```

- 1 mark: base case (bar = 0)
- 1 mark: correct intermediate steps
- 1 mark: for returning correct value
- Note: marks were granted based on whether the function **holistically** attempted to replicate the functionality of the original function. I.e. simply indicating the base case and nothing else is **not** enough to get 1/3

(d) Remark on the comparative runtime complexities of the recursive and non-recursive functions `foo_too()` and `foo_too_2()`, respectively, in 1-2 sentences. **[1 mark]**

Both algorithms will have the same runtime complexity for a given value of bar, as a single execution path is run for both functions.

- 0.5 mark: identifying the runtimes are the same
- 0.5 mark: explanation

(e) Derive a relationship between the value of `bar` and the runtime complexity of the non-recursive solution, `foo_too_2`. **[1 mark]**

The runtime complexity will scale according to $\log_2(bar)$.

- 1 mark: correct answer

Question 5 [6 marks]

This question examines the runtime complexity of the function `nanana()`.

```
def is_seven(seven):
    '''
    (List[int]) -> Bool
    '''
    is_seven = True
    for s in seven:
        if s != 7:
            is_seven = False
    return is_seven


def nanana(seven):
    '''
    (List[int]) -> None
    '''
    if not is_seven(seven):
        limit = 1
        while limit <= len(seven) and not is_seven(seven):
            for i in range(limit):
                seven[i]= 7
            limit+=1
```

(a) **Write a new function**, `ananas()` which has the same functionality as `nanana()`, but with improved runtime performance. **Underline your statement which modifies `seven`.** You may use the `is_seven()` function, but are not required to. **[2 marks]**

-0.5/2 for any issues with syntax

```
def ananas(seven):
    '''
    (List[int]) -> None
    '''

    for i in range(len(seven)):
        seven[i] = 7
```

2/2 for any variation of this that produces the right output in less than n^2

```
def ananas(seven):
    '''
    (List[int]) -> None
    '''
    for s in seven:
        s = 7
```

-1/2 for this solution

(b) Fill out the table below for the function `nanana()`, providing examples for the input parameter, `seven`, when it has a length of 7. **[2 marks]**

- 0.5 for each correct answer

Function	**Case**	**Example input**	**Number of times `(*)` runs**
`nanana()`	Best-case	`[7, 7, 7, 7, 7, 7, 7]`	`0`
	Worst-case	`[1, 1, 1, 1, 1, 1, 1]` `Any list where the last element is not 7.`	`1 + 2 + 3 + 4 + 5 + 6 + 7 = 28`

(c) Fill in the following table. You do not have to simplify any expressions, but write them completely and concisely. **[2 marks]**

- 0.5 for each correct answer

Function	**Case**	**Number of times `(*)` runs, as a function of n**
`nanana(sept)`	Best-case	`0`
	Worst-case	`1 + 2 + ... + (n-1) + n =` $\sum_{i=1}^{n} i$ `n^2 or n(n+1)/2 or (1 + 2 + ... + n) all acceptable`
`ananas(sept)`	Best-case	`n (depending on implementation)`
	Worst-case	`n (depending on implementation)`