GMIT H Dip in Data Analytics
Computational Thinking with Algorithms Problem Sheet
Patrick Moore G00364753

Question 1

Consider the following method:

```
def mystery(n):
    print(n)
    if n < 4:
        mystery(n + 1)
    print(n)
```

What will the output of the call mystery(1) be?

Write an explanation of the reasoning behind your answer, using the aid of either a recursion trace diagram or a stack diagram. Include any code which you write for testing or explanation purposes as part of your answer.

Answer for Question 1

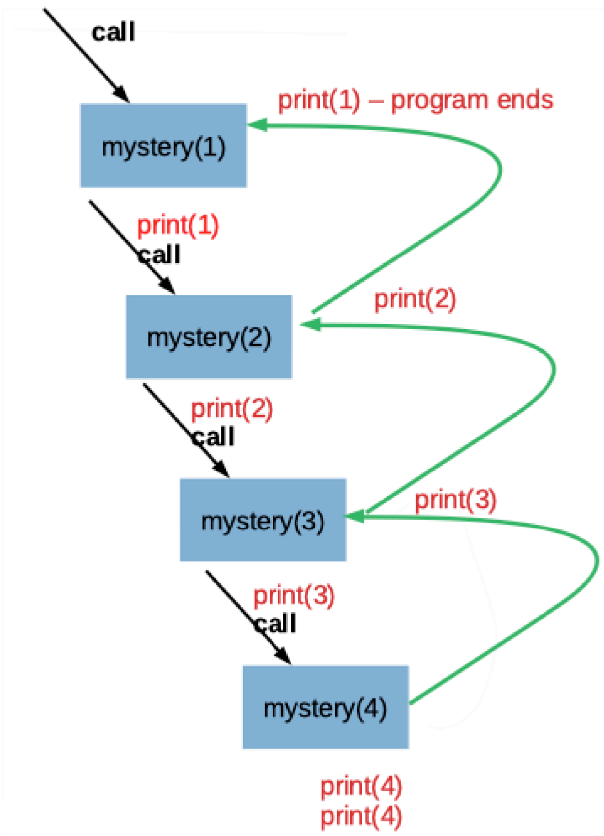
In order to test this, the method was coded in Python in Visual Studio code, and then called by passing the value '1' to it. The source code for this is included with the submission as `Q1.py`

```
# Patrick Moore 2019-02-23
# This is a script to test the algorithm in Question 1 of the
# Computational Thinking with Algorithms problem sheet

def mystery(n):
    # the function program first prints the number passed to it
    print(n)
    # if the number is less than 4 if the function calls itself
    # for n + 1 (this will repeat until n = 4)
    if n < 4:
        mystery(n + 1)
    # once the condtion in the if statement is met the function
    # moves onto the the final line and prints the final value of n
    # the function will trace back and close each of the "open" instances of the function
    print(n)

# caLL the function by passing the value '1' to it
mystery(1)
```

The output of this function call is shown below:

```
patrickmoore@Patricks-MBP:~/Documents/GMIT/GIT/gmit-cta-problems$ python myster.py
1
2
3
4
4
3
2
1
```

The recursion trace for the call “mystery(1)” is shown below with the original function.




```python
def mystery(n):
    print(n)
    if n < 4:
        mystery(n + 1)
    print(n)
```

- When the function is called with the original value for ‘n’ set to 1, the first thing the function does is print out this value (1 in this case)
- It then checks if the value passed for n is less than 4. If this is the case the function will recursively call itself for the case ‘n + 1’ (2 in this case). (Note that the recursion will continue until the ***base case*** of n is greater than or equal to 4 is met. Also note that the recursion is making progress towards the ***base case*** as on each subsequent call, the new value for n is closer to 4)
- The function call with the value of n set to ‘2’, first prints out ‘2’, and again checks to see if n is less than 4. It is less than 4, so it calls the function again with n = 3.
- This third call of the function prints out ‘3’ and checks to see if n is less than 4. It is still less than 4, so the it calls itself with n = 4.

- The 4th call of the function prints out ‘4’ and checks to see if n is less than 4. It isn’t, so the function moves down to the last line in the function and prints out ‘4’ again and finishes.
- Now, the third function call can move onto the final line in the function and print out its value for n (‘3’) and finish.
- Then, the second call of the function can move onto the final line in the function and print out it’s value for n (‘2’) and finish.
- Finally the initial call can move down to the last line in the function, printing it value for n (‘1’).
- The program terminates at this point

Question 2

Consider the following methods:

```
def finder(data):
    return finder_rec(data, len(data)-1)


def finder_rec(data, x):
    if x == 0:
        return data[x]
    v1 = data[x]
    v2 = finder_rec(data, x-1)
    if v1 > v2:
        return v1
    else:
        return v2
```

Q2 (a) What value is returned by a call to `finder` when the following array is used as input? [0, -247, 341, 1001, 741, 22]

Q2 (b) What characteristic of the input data set does the `finder` method determine? How does it determine this result?

Q2 (c) Can you add some inline comments to the code above to explain how it works?

Q2 (d) Write a method which achieves the same result as `finder`, but which uses an iterative approach instead of recursion.

Write an explanation of the reasoning behind your answers to the above questions, using the aid of either a recursion trace diagram or a stack diagram for Q2 (b). Include any code which you write for testing or explanation purposes as part of your answer.

Answer for Question 2

2(a). In order to determine the output of the functions, they were coded in Python in Visual Studio Code as follows:

```python
# Patrick Moore 2019-03-03
# This is a script to test the algorithm in Question 2 of the
# Computational Thinking with Algorithms problem sheet

def finder(data):
    return finder_rec(data, len(data)-1)

def finder_rec(data,x):
    if x == 0:
        return data[x]
    v1 = data[x]
    v2 = finder_rec(data, x-1)
    if v1 > v2:
        return v1
    else:
        return v2

y = [0, -247, 341, 1001, 741, 22]

print(finder(y))
```

The script was ran from the integrated terminal in Visual Studio Code

The output of this code is `1001`

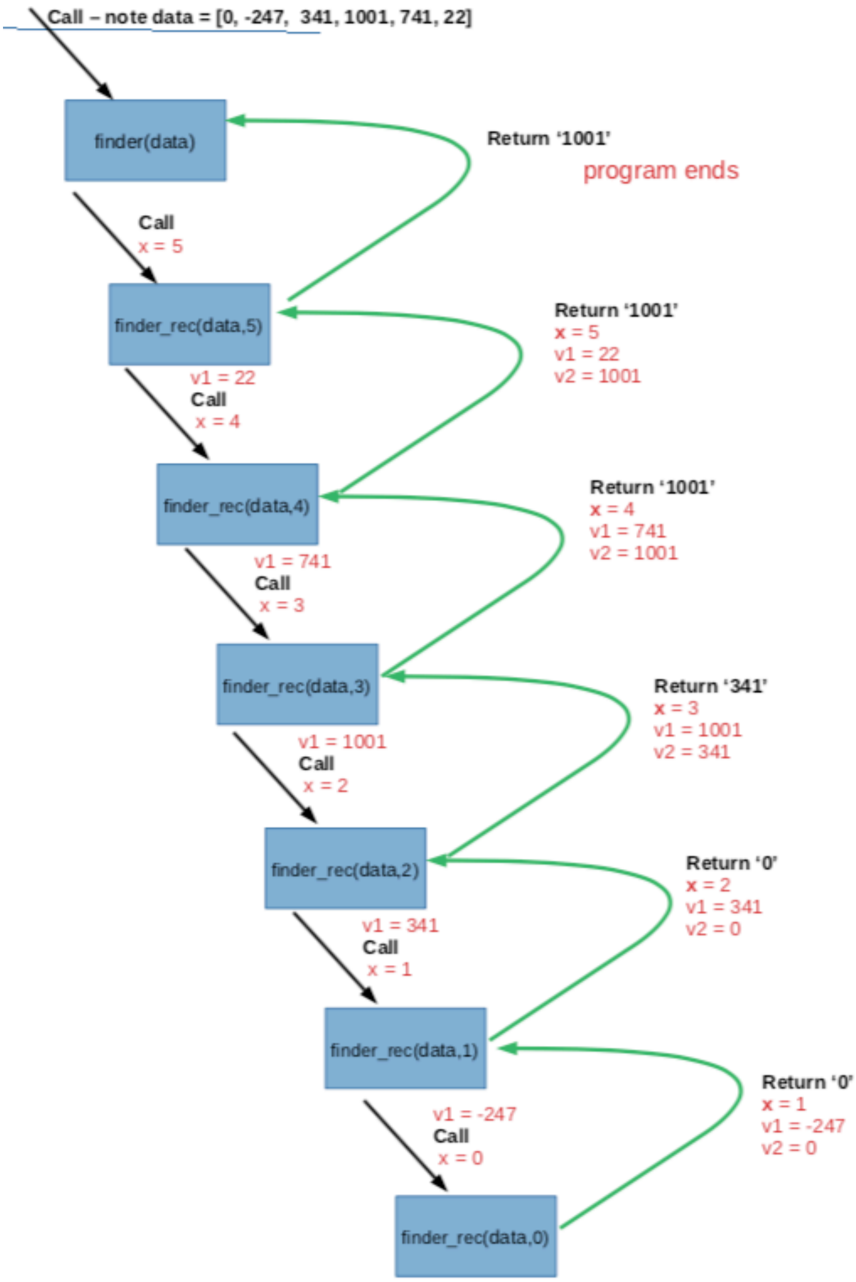
2(b). The `finder` method takes a data array as an input and returns the value of the largest number in the array. In order to determine how the method worked I added some `print()` statements to the code to track the values of the variables at each stage in this recursive process. I have included the source code in the submission as `Q2b.py.`

```python
# Patrick Moore 2019-03-03
# This is a script to test the algorithm in Question 2 of the
# Computational Thinking with Algorithms problem sheet

def finder(data):
    return finder_rec(data, len(data)-1)

def finder_rec(data,x):
    print("____________")
    print("x: " + str(x))
    if x == 0:
        print("____________")
        return data[x]
    v1 = data[x]
    print("v1: " + str(v1))
    v2 = finder_rec(data, x-1)
    print("x: " + str(x))
    print("v1: " + str(v1))
    print("v2: " + str(v2))
    print("____________")
    if v1 > v2:
        return v1
        print(v1)
    else:
        return v2
        print(v2)

y = [0, -247, 341, 1001, 741, 22]

finder(y)
```

The output of this code is now:

```
(base) patrickmoore@Patricks-MBP:~/Documents/GMIT/GIT/gmit-cta-problems$ python max.py
____________
x: 5
v1: 22
____________
x: 4
v1: 741
____________
x: 3
v1: 1001
____________
x: 2
v1: 341
____________
x: 1
v1: -247
____________
x: 0
____________
x: 1
v1: -247
v2: 0
____________
x: 2
v1: 341
v2: 0
____________
x: 3
v1: 1001
v2: 341
____________
x: 4
v1: 741
v2: 1001
____________
x: 5
v1: 22
v2: 1001
____________
```

This simplifies the process of creating a recursion trace for the call described in the problem:



The recursive trace for this program works as follows:

- When the `finder([0, -247, 341, 1001, 741, 22])` function call is made, the `finder()` function calls the `finder_rec()` function, passing the original list `(data)` and the index of the last position in that list (5 in this case) as arguments.
- The `finder_rec(data, 5)` first checks to see if the passed index `(x)` is equal to zero, if this is the case it will return the first item in the list. Note that this represents the base case for this recursive process.
- For the initial call, x is equal to 5, so it sets a variable `v1` equal to the number at index 5 (22 in this case. It sets another variable, `v2`, by calling `finder_rec(data, 4)`

- When `finder_rec(data, 4)` is called we have not yet reached the base case so it sets it's variable `v1` equal to 741, and it's `v2` variable equal to the recursive call `finder_rec(data, 3)`
- When `finder_rec(data, 3)` is called we have not yet reached the base case so it sets it's variable `v1` equal to 1001, and it's `v2` variable equal to the recursive call `finder_rec(data, 2)`
- When `finder_rec(data, 2)` is called we have not yet reached the base case so it sets it's variable `v1` equal to 341, and it's `v2` variable equal to the recursive call `finder_rec(data, 1)`
- When `finder_rec(data, 1)` is called we have not yet reached the base case so it sets it's variable `v1` equal to -247, and it's `v2` variable equal to the recursive call `finder_rec(data, 0)`
- When `finder_rec(data, 0)` is called we have now reached the base case (x == 0) so we return the first item in the list to the `finder_rec(data,1)` function call. This just so happens to be 0 in this case.
- So in the `finder_rec(data, 1)` call, `v1` is still equal to -247, and `v2` is now 0, so the function checks to see the bigger number and returns it (closing out this instance of the function). 0 is the larger number so it gets returned to the `finder_rec(data,2)` function call.
- In the `finder_rec(data, 2)` call, `v1` is still equal to 341, and `v2` is now 0, so the function checks to see the bigger number and returns it (closing out this instance of the function). 341 is the larger number so it gets returned to the `finder_rec(data,3)` function call.
- In the `finder_rec(data, 3)` call, `v1` is still equal to 1001, and `v2` is now 341, so the function checks to see the bigger number and returns it (closing out this instance of the function). 1001 is the larger number so it gets returned to the `finder_rec(data,4)` function call.
- In the `finder_rec(data, 4)` call, `v1` is still equal to 741, and `v2` is now 1001, so the function checks to see the bigger number and returns it (closing out this instance of the function). 1001 is the larger number so it get returned to the `finder_rec(data,5)` function call.
- In the `finder_rec(data, 5)` call, `v1` is still equal to 22, and `v2` is now 1001, so the function checks to see the bigger number and returns it (closing out this instance of the function). 1001 is the larger number so it get returned to the `finder(data)`function call.
- `finder(data)` can now return the value 1001 as the largest item in list and the program can now close out.

Q2 (c) I have added some inline comments to the code to explain how it works. I have included the source code in the submission as `Q2c.py` Please see below

```python
# Patrick Moore 2019-03-03
# This is a script to test the algorithm in Question 2 of the
# Computational Thinking with Algorithms problem sheet

# define a function that takes a list as an argument
def finder(data):
    # call the finder_rec() function by passing the orignal list,
    # and the index of the last item in the list as arguments
    return finder_rec(data, len(data)-1)

# define a function that takes a list and an integer as arguments
def finder_rec(data,x):
    # if x is zero - return the first item in the list (base case for recursion)
    if x == 0:
        return data[x]
    # otherwise set v1 to be the value of the item in that position in the list
    v1 = data[x]
    # v2 is set by calling the finder_rec() function for the previous item in the list
    v2 = finder_rec(data, x-1)
    # once we have reached the base case it will the compare each item in the list,
    # returning the greatest number each time
    # ultimately it will return the greatest number in the list
    if v1 > v2:
        return v1
    else:
        return v2

# create a variable to store the data list
y = [0, -247, 341, 1001, 741, 22]

# call the finder function
finder(y)
```

Q2 (d) I have written a function in Python to determine the largest item in a list of numbers using an iterative approach instead of a recursive one. I have included the source code in the submission as `Q2d.py` This is shown below:

```python
# Patrick Moore 2019-03-05
# This is a script to create a function that finds the max value in a list
# using an iterative approach, as directed by Question 2(d) of the
# Computational Thinking with Algorithms problem sheet

# define a function that takes a list as an argument
def max_iter(data):
    # set the maximum value to be the first item in the list
    maximum = data[0]
    # create a counter variable for the while loop
    counter = 0
    # use a while loop to iterate over the length of the list
    while counter < len(data):
        # check if any item in the list is greater than the current maximum
        if data[counter] > maximum:
            # if so, set maximum to that value
            maximum = data[counter]
        # increment the counter
        counter = counter + 1
    # once the while loop terminates, return the max value
    return maximum

# create a variable to store the data list
y = [0, -247, 341, 1001, 741, 22]

# call the max_iter() function
print(max_iter(y))
```

Question 3

Consider the following method which checks if an array of integers contains duplicate elements:

```
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:  # avoid self comparison
                continue
            if elements[i] == elements[j]:
                return True  # duplicate found
    return False
```

Q3 (a) What is the best-case time complexity for this method, and why?

Q3 (b) What is the worst-case time complexity for this method, and why?

Q3 (c) Modify the code above, so that instead of returning a boolean indicating whether or not a duplicate was found, it instead returns the number of comparisons the method makes between different elements until a duplicate is found.

Q3 (d) Construct an input instance with 5 elements for which this method would exhibit its best-case running time.

Q3 (e) Construct an input instance with 5 elements for which this method would exhibit its worst-case running time.

Q3 (f) Which of the following input instances, `[10,0,5,3,-19,5]` or `[0,1,0,-127,346,125]` would take longer for this method to process, and why?

Answer for Question 3

3(a). The algorithm used for the function `contains_duplicates()` is a simple "brute force" method that checks each combination of numbers and stops as soon as it finds a duplicate. It does this using nested *for loops*. In the outer *for loop* (the one with "i" as the counter), the function starts at the first item in the list and iterates through each item in the list. The inner *for loop* (the loop with "j" as the counter) is then used to compare each item against each value for "i". The function has a safety system built into it to skip over self comparison (i.e. it wont compare the nth item in the list to itself). The other interesting fact about this function is that is is designed to terminate and return a value of "True" if at any stage it finds as comparison.

This means that the if the first 2 items in the list (items at index 0 and 1), are the same, the function will compare the first item with itself, and skip over it. It will then compare the first item in the list to the second item in the list. As they are the same it will terminate after 2 comparisons. This is the best case time complexity for this method and is said to be "constant time" denoted as $O(1)$.

3(b). As described in 3(a) above, the algorithm is a brute force method that compares each combination of numbers in the list and stops once it has found a match. So this means that the worst case scenario for this algorithm is when a list that contains only unique elements is passed to it. In this case, the function will iterate through each item in the list, and for each iteration it will also need to iterate through each item in the list to make a comparison. As there are no duplicates it the worst case scenario it will run to the end making n^2 comparisons (where in the number of unique items in the list. The worst case time complexity is said to depend on the square of the number of elements in the list and is denoted as $O(n^2)$.

3(c). The code in the function from questions 3(a) and (b) was rewritten to include a counter to count the number of comparisons made. As in the original function, it stops once it finds a matching pair, but instead of returning a boolean True or False, it returns the value of the counter at that point. Note that I decided to only count the comparisons between the various elements in the array – I have not counted the operation to compare the indexes in order to avoid self comparisons. The source code for this is included in the submission as `Q3c.py`. This code is shown below:

```python
def dupes(elements):
    # create a variable to count the comparisons made
    count = 0
    # use nested for loops to compate the items in the list
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            # avoid self comparison
            if i == j:
                continue
            # if the elements match, increment the counter and return the final value
            if elements[i] == elements[j]:
                count = count + 1
                return count
            # otherwise increment the counter
            else:
                count = count + 1
    # if no match is found return the final value of the counter
    return count
```

3(d). As described in question 3(a), the best case scenario for this for this function would be when the first 2 items in the list match. So an input instance where this function would exhibit its best case running time would be: list = [1,1,2,3,4]. I have tested this with the code I wrote for question 3(c) and it terminates after 1 comparison. (note that my function is designed to only count the element comparisons, not the index comparisons, so it only starts to count from when it compares the first item in the list to the second one.)

3(e). As described in question 3(b), the worst case scenario for this function is when there are no matching items in the list. So an input instance where this function would exhibit its worst case running time would be: list = [1,2,3,4,5]. I have tested this with the code I wrote for question 3(c) and it terminates after 20 comparisons. (note that my function is designed to only count the element comparisons, not the index comparisons, so it will count a total of (n^2 – n) comparisons – which is 20, when n is equal to 5)

3(f). `[10,0,5,3,-19,5]` should take longer to process than `[0,1,0,-127,346,125].` The reason for this is that the function is set up to loop through the lists from left to right. If you take the first array, the function will loop through the list comparing '10' to everything and it wont find a match (making 5 comparisons), it will then loop through the list comparing' 0' to everything and it wont find a match (making another 5 comparisons). Finally it will loop through the list comparing '5' to everything and it will terminate at the end of the list (as 5 is the last number in the list). The function will have made a total of 15 comparisons before finding a match and terminating.

Considering the case where the second list is passed to the function. The function will start by comparing '0' to every other item in the list. After making 2 comparisons it will terminate at 0 is the first item in the list and the third item in the list.

Note that I have tested this using the function I created for 3(c) and proven these results are expected. See below:

```
def dupes(elements):
    # create a variable to count the comparisons made
    count = 0
    # use nested for loops to compate the items in the list
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            # avoid self comparison
            if i == j:
                continue
            # if the elements match, increment the counter and return the final value
            if elements[i] == elements[j]:
                count = count + 1
                return count
            # otherwise increment the counter
            else:
                count = count + 1
    # if no match is found return the final value of the counter
    return count

a = [10,0,5,3,-19,5]
b = [0,1,0,-127,346,125]
print(dupes(a))
print(dupes(b))
```

Output of the code is:

```
patrickmoore@Patricks-MBP:~/Documents/GMIT/GIT/gmit-cta-problems$ python dupes.py
15
2
```