

Higher Diploma in Science - Computing (Data Analytics)

Computational Thinking with Algorithms (Problem sheet)

Lecturer: Dr. Patrick Mannion

Student: Gareth Duffy

Student number: g00364693

Q1.

The output of the mystery(1) method will be:

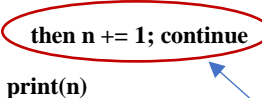
```
def mystery(n): # Takes a single argument in its parameters (Integer or float).
    print(n) # First print of n (Incrementally prints 1,2,3,4)
    if n < 4: # If n is less than 4, continue to the line below. If n is 4 or more we skip the mystery(n + 1) call below.
        mystery(n + 1) # Adds 1 onto n, and pushes this call onto the stack. Now recursion returns to second line.
    print(n) # Second print of n (Incrementally prints 4,3,2,1). This print occurs once n becomes >= to 4.
```

```
mystery(1)
```

```
1
2
3
4
4
3
2
1
```

The reasoning behind the following explanation of the outcome of the method is grounded in my own experimental observations conducted on the algorithm. Initially, when we look at the output above, it appears to be “mirrored”, as if it was processed in a forward then backward iterative loop. However, after a process of trial and error elimination regarding how the algorithm functions, this was found not to be the case. Recursion and not iteration, is the operative method within the algorithm. We will investigate the function by paying close attention to what is going on within the activation frames of the stack, the process of which is outlined and elaborated on below. With a stack diagram, we can represent the state of a program during a function call. This diagram can help a layperson for example to better interpret the recursive operations occurring. Each time a function gets *called*, the interpreter creates a new function activation frame, which contains the functions local variables and parameters. With a recursive function, there can be more than one frame on the stack at the same time as we will see in **fig.1(a & b)** below. Before we view the stack, we should identify a base case before we begin, which in this instance could be:

```
def mystery(n):
    print(n)
    if n<4,
    then n += 1; continue
    print(n)
```



This base case above provides a non-recursive alternative to the method of: **mystery(n + 1)**, which is in fact a recursive function call.

The red arrows in the diagram show where the function calls are pushed onto the stack. To begin with, the stack is initially empty, and **mystery(1)** makes a call to the **mystery()** function. This call pushes **n = 1** onto the stack. Each time the **mystery** method is invoked, the method's activation frame (record/log) is placed on top of the stack. So **mystery(1)** is the first call on the stack and the method traverses to line 2. Here it prints the value of **n** (in this case **1**), and traverses down to the **if** condition on line three. At the **if** condition, it follows **True** that **n** is less than **4** and so the condition is satisfied and **1** is added onto the value of **n** in line four, and called to the stack. This is the first example of recursion happening in the algorithm, i.e. where **mystery()** has essentially called itself inside the function. Now **mystery(2)** is pushed on top of the stack (see Time 3 below). Next the method returns back to line 2 and prints the value of **n** (**2**), and drops as before to the **if** condition to be checked if it is less than **4**. It is, and thus **1** is added to **n**. **mystery(2)** has now become **mystery(3)** and is recursively called and pushed onto the stack. Again, **n** is printed on line two, this time as **3**. Once again, **n** is fed into the **if** condition below and because it is still less than **4**, is increased by one in the **mystery(n + 1)** call below, and pushed onto the stack (Time 5). At line 2, the method prints **n** as **4** and drops to the **if** condition again, but this time the condition is no longer satisfied due to the fact **n** is *not* less than **4**. Instead **n** is equal to **4** so it does not get called to the top of the stack like the previous recursive calls. Instead it skips down to the second **print(n)** clause below. Here, the second print behaves like an *elif* or *else* clause and prints off the value of **n**, i.e. **4** (for a second time). There are now two 4s printed.

Next, the **mystery** method returns to line four where the first four calls are being “held” on the stack, essentially paused there. **n** has no return value now, so the method returns back up the stack and begins to pop the initial calls of the **mystery** function off the stack. Anytime a method returns or exits the activation frame is popped off the stack. Thus, **mystery(4)** is the first to be popped off because it was the last one pushed on (last in first off is the rule). Now, **mystery(3)** which is being held on line four with the first calls of **mystery(1)** and **(2)**, is on the top of the stack (Time 6). The recursive loop has been terminated and remains essentially paused on line 5; **mystery(3)** now gets printed and popped off the stack. This now leaves **2** on top of the stack to undergo the same process as **3** did, and then **1**. Once **1** is printed and called/popped back off the stack, termination of the function occurs. Now all calls have been printed and the stack is back to its original clean slate.

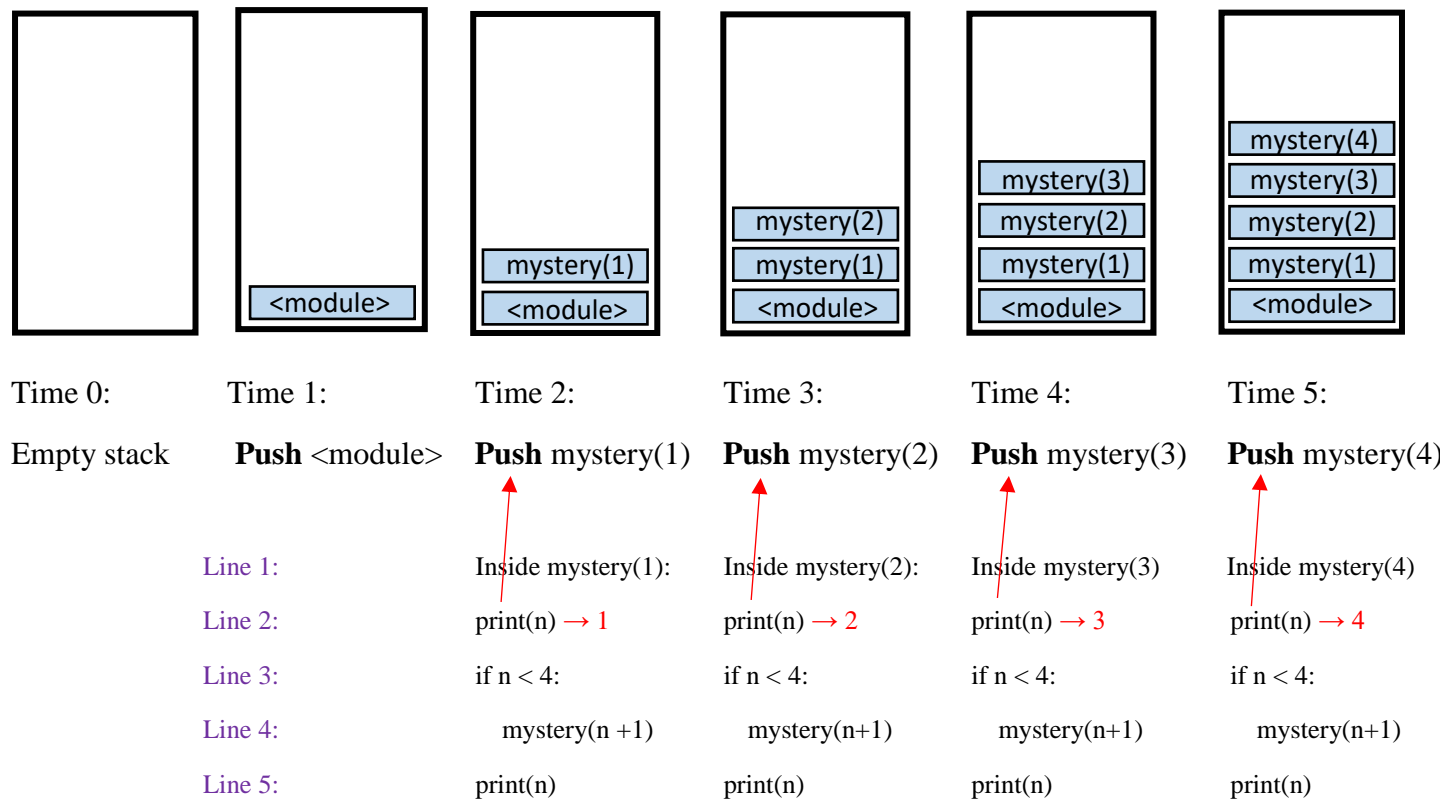


Fig. 1(a) - Stack diagram for mystery() function

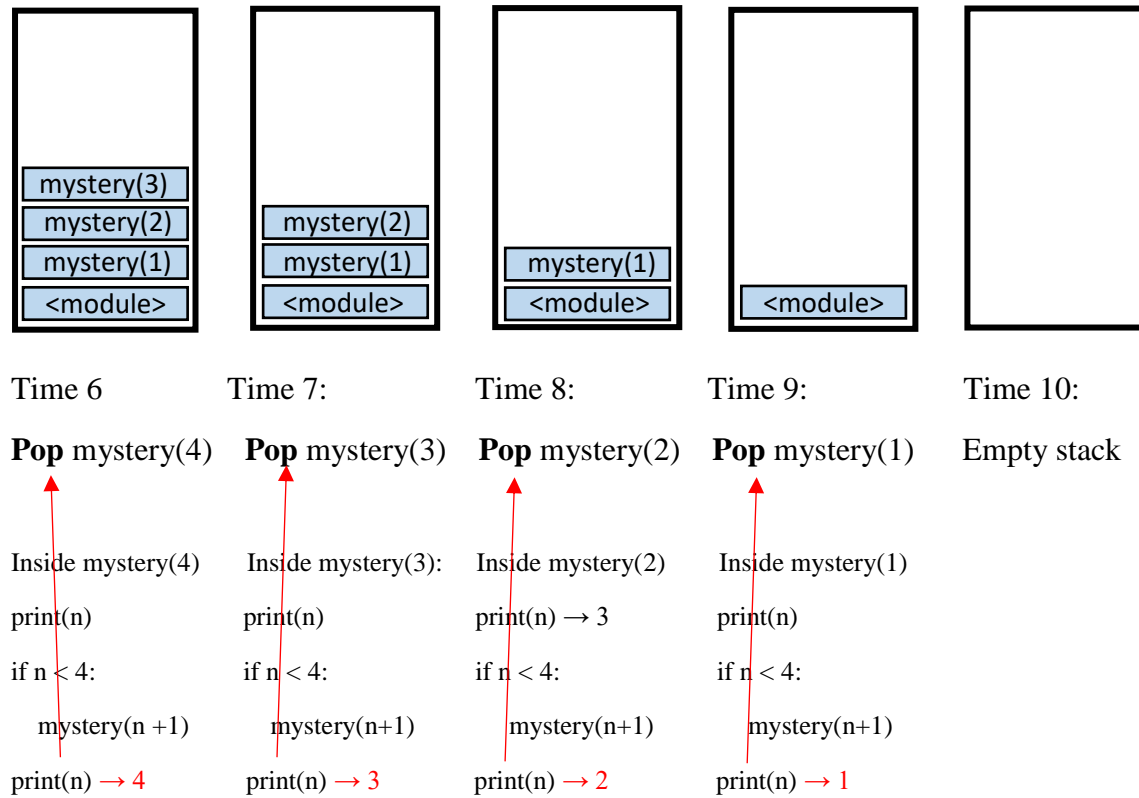


Fig. 1(b) - Stack diagram for `mystery()` function

While conducting exploratory investigations on the function, an interesting observation was made whereby if the second `print(n)` function on line 5 is removed, the output of `mystery(1)` becomes:

```
def mystery(n): # Takes a single argument in its parameters (Integer or float).
    print(n) # First print of n (Incrementally prints 1,2,3,4)
    if n < 4: # If n is less than 4, continue to the line below. If n is 4 or more we skip the mystery(n + 1) call below.
        mystery(n + 1) # Adds 1 onto n, and pushes this call onto the stack. Now recursion returns to second line.

mystery(1)
```

```
1
2
3
4
```

This output shows how with the second print function omitted the recursion is not given the second option to print off *n* once it is popped of the stack (where the number exceeds 3). But why is this the case? This is so due to the fact that there is no second print function anymore which acts like an `elif` or `else` condition. There is also now no alternative option for the algorithm but to return/terminate after it calls and prints `mystery(4)`. Here, the condition ultimately “fails” at **n=4**, recursion stops and the method activation frames now all pop of the stack.

Another interesting observation is that when an inputted number (**n**) of 4 or more is called with the `mystery()` function, it will only print out 2 of the same number i.e. two fours.

```
def mystery(n): # Takes a single argument in its parameters (Integer or float).
    print(n) # First print of n (Incrementally prints 1,2,3,4)
    if n < 4: # If n is less than 4, continue to the line below. If n is 4 or more we skip the mystery(n + 1) call below.
        mystery(n + 1) # Adds 1 onto n, and pushes this call onto the stack. Now recursion returns to second line.
    print(n) # Second print of n (Incrementally prints 4,3,2,1). This print occurs once n becomes >= to 4.

mystery(4)
```

```
4
4
```

Interesting, but again, why is this so? This is because *n* does not have the opportunity to grow incrementally like 1,2, or 3 would, where they would all satisfy the `if` condition of being less than four, and hence be called onto the stack at the recursive `mystery(n+1)` line, and then subsequently called back off the stack and printed a second time.

Q2.

Q2(a).

The value **1001** is returned by a call to `finder()` when the array `[0, -247, 341, 1001, 741, 22]` is used as input. We can see this below.

```
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  
  
finder([0, -247, 341, 1001, 741, 22])
```

1001

Q2(b).

After exploratory investigation on the function, it was found that the `finder` method determines the characteristic of *largest number* in an array input. Determination of this result is explained and elaborated on in **fig.2** below. As with question 1, the processes will be outlined and explained by means of stack diagram. The red arrows in the diagram show where the function calls are pushed onto the stack. The blue arrows show the filtering process of determining the largest number.

The `finder` program determines the largest number by means of recursion, where the `finder` function initially calls another function (**`finder_rec`**), and then **`finder_rec`** initiates a recursive process to determine the largest number in the inputted array. Initially the function **`finder()`** is called with the data array in its parameters. **`finder()`** calls **`finder_rec()`** which takes in its parameters the inputted data array and the length of the data array (6) minus one which is 5. So here, **`finder_rec()`** is actually calling the element **22** onto the stack. This integer will serve as the starting index position of the array to be looped over.

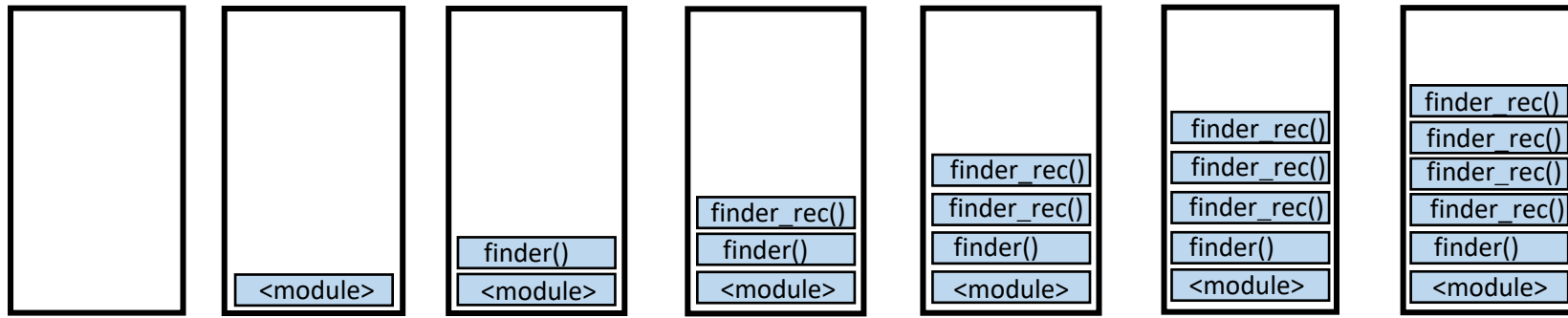
So far, the **`finder()`** function has been called to the stack (the data array) and has traversed to return and call the **`finder_rec()`** function on line 2 and pushes **22** on top of the stack. Next the function **`finder_rec()`** traverses to the **`if`** condition on line 5 (**`if x == 0:`**). Right now, **`x`** is equal to **5** because as we have seen above, it was “set” to **5** in order to represent the positional index starting point of the integers in the data array and push this onto the stack. Remember, the length of the

data array is 6 elements long, and in line 2, **x** is represented by the parameter: **len(data)-1**. This means it will deduct 1 from the original value of the *length of the data array* which was 6, and are left with 5. So, [5] is the *last* index position of the data array as the array list is arranged from [0-5]. Therefore, we know **x** represents the index position of each integer in the array. This is why **22** and is the first **finder_rec()** instance called onto the stack (see Time 3).

Moving on, we know that **x** has the (temporary) index value of [5], and thus it does not adhere to the **if** condition of **x == 0**. This means it does *not* return **data[x]** on line 6, but instead drops to line 7, where it assigns **data[x]** i.e. the last element in the array, to the temporary container called **v1**. Because **x** is in index position [5], it assigns **22** to this **v1** container. Now it drops to line 8 and recursively calls **finder_rec()** again, this time *subtracting one from x* to move backward to index [4] (from [5]). Now it pushes **741** to the stack and assigns index **x** i.e. [4] or **741** as **v1**. Next the method moves back up to the **if** condition again and checks again if **x** is equal to 0. Why is it checking to see this? Because [0] is the last element index in the array as it is working from last to first i.e. from index [5] to index [0]. So, when it reaches index [0] it will know to return the element held in **x**. But for now, **x** doesn't equal [0] yet.

Next, as before **finder_rec()** makes another recursive call to itself by pushing the next indexed element [x] which is **3**, onto the stack, in this case **1001**. **1001** now becomes **v1** and index[2] i.e. **1001** is pushed onto the stack at line 8 from the recursive call. This process continues for index [1] (**-247**) and then index [0] (**0**). At time 8, **0** is the last element to be pushed on top of the stack. Now, **x** is finally at index [0] and therefore the **if** condition (**x == 0**) is fulfilled, so **x** is returned and index[0] (element **0**) is popped off the stack. Here, Time 8 has reached no return values and thus marks the beginning of the calls/pops returning back down the stack. Now, **-247** is on top of the stack where it was originally held. It is held in the **v1** container and the previous index element (**0**) is held in the **v2** container. These elements are now ready for comparison on line 9 to see if one is greater than the other.

Now the greater of the two numbers is returned to in order save the larger number as a temporary variable. Here, **v1** is serving as the container for the original function calls (of the array elements), and **v1** is being used to compare to **v2** which saves the larger returned element in order to compare it with the *next v1* element “waiting in line”. This process results in the larger number always being returned/saved/temporarily assigned as **v2**. We can see this process occurring by observing the blue arrows filtering the larger numbers from Time 9 to Time 13 below. Every time the larger number is determined and returned (or exits), the activation frame is popped off the stack. To reiterate, the process compares both containers and returns the larger one as **v2**, because **v1** is still holding the original number called to the stack in line waiting for the next comparison. At time 13, the function performs its last comparison and determines that **1001** is the largest number. Following this, **finder_rec()** has been completed and is thus popped off the stack leaving just the original **finder()** call remaining, which is now also popped off. This algorithm can be considered as an example of exponential recursion, i.e. a method that makes more than two calls to itself.



Time 0: Empty stack
 Time 1: **Push** <module>
 Time 2: **Push** finder()
 Time 3: **Push** finder_rec()
 Time 4: **Push** finder_rec()
 Time 5: **Push** finder_rec()
 Time 6: **Push** finder_rec()

(data)
 x: 5
 v1: 22
 x: 4
 v1: 741
 x: 3
 v1: 1001
 x: 2
 v1: 341

Line 1:
 Line 2:
 Line 3:
 Line 4:
 Line 5:
 Line 6:
 Line 7:
 Line 8:
 Line 9:
 Line 10:
 Line 11:
 Line 12:

Inside 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:

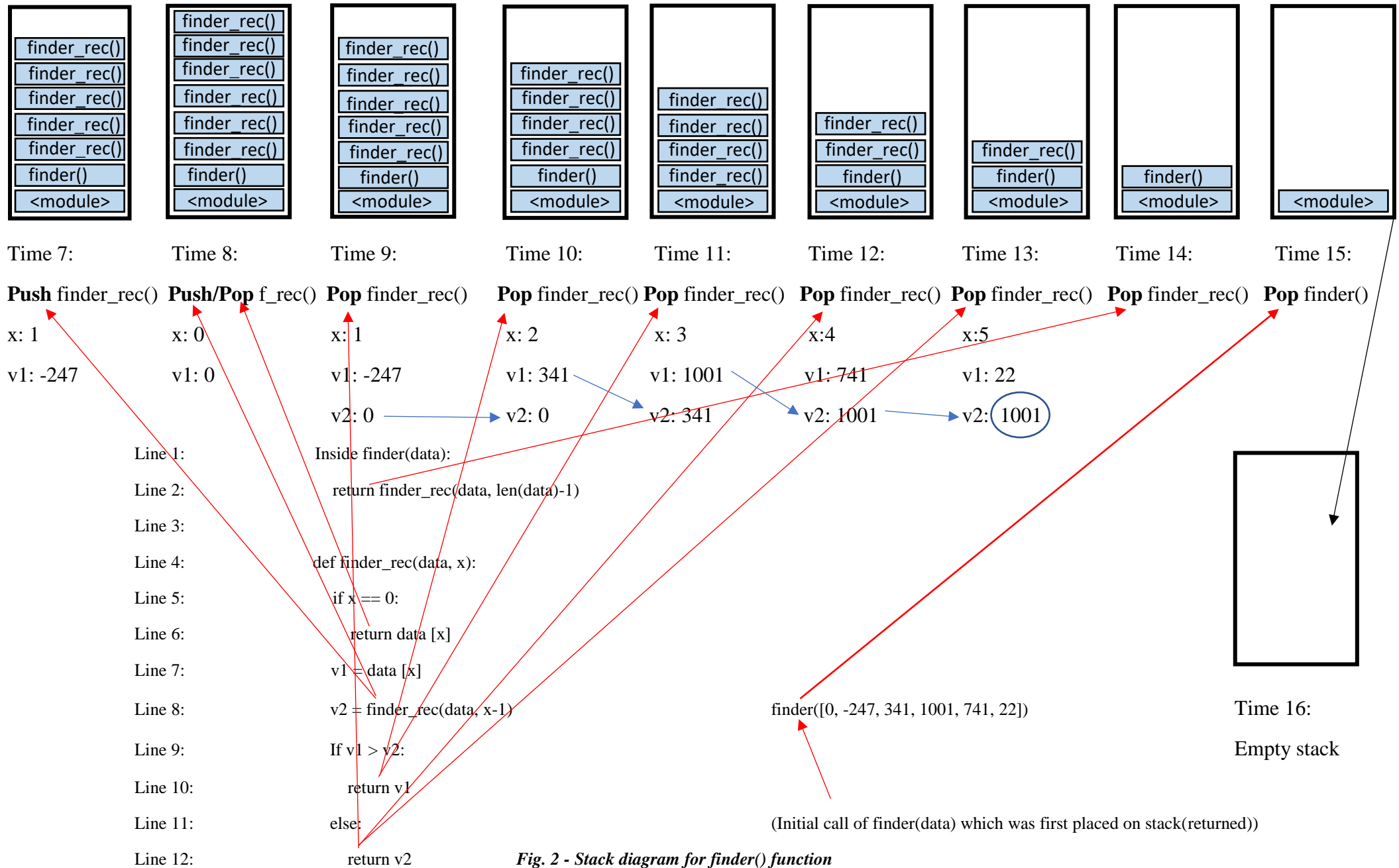
return v1

else:

return v2

finder([0, -247, 341, 1001, 741, 22])

Initial call of finder(data)



Q2(c).

Below is the finder() function with explanatory comments for purpose of clarity.

```
def finder(data): # Takes an array or set of numbers
    return finder_rec(data, len(data)-1) # Here, finder() calls finder_rec(). finder_rec() has two parameters which takes
                                         # the data array as its first argument, and the length of this data array, (which is 6)
                                         # -1 as its second argument (x). Thus, the len(data)-1 parameter receives "5" in order
                                         # to serve as the starting point of the array index (0-5).

def finder_rec(data, x): # Takes two arguments in its parameters: the data array and the index position.

    if x == 0: # "x" is the index position of the data array. When x is equal to index "0", this if condition can be fulfilled.
        return data[x] # Returns the index position of a single number in the array as "x".
    v1 = data[x]        # "v1" becomes a temporary container for the integer in the current index position of "x".
    v2 = finder_rec(data, x-1) # "v2" becomes a temporary container for the next indexed integer in line (x-1), and finder_rec()
                               # is called to the stack.

    if v1 > v2:          # If the number in the v1 container is larger than that in v2, return v1.
        return v1        # Calls/pops v1 off the stack (Second call)
    else:                # Otherwise, return the other number (v2).
        return v2        # Calls/pops v2 off the stack (Second call).
                         # Thus, this process of elimination keeps retaining the larger number until no more larger numbers exist.

finder([0, -247, 341, 1001, 741, 22]) # Answer 1001
```

1001

Q2(d).

Below is my written method which achieves the same result as `finder()` but in an iterative manner (as opposed to recursive):

```
def finder(data):  
    largest = 0 # Provisionally set the "largest" number of the list/array to "0".  
  
    for i in data: # Iterate through the entire data array.  
        if(largest < i): # If "largest" is less than the current iterated number "i"...  
            largest = i # Then assign "largest" this new value of "i" (This continues until the largest number is found).[2]  
    return largest # Largest number is returned once no more comparisons are available for iteration.  
  
print(finder([0, -247, 341, 1001, 741, 22])) # Test any input array  
  
1001
```

Below is another instance of this iterative function but with a different input example:

```
def finder(data):  
    largest = 0 # Provisionally set the "largest" number of the list/array to "0".  
  
    for i in data: # Iterate through the entire data array.  
        if(largest < i): # If "largest" is less than the current iterated number "i"...  
            largest = i # Then assign "largest" this new value of "i" (This continues until the largest number is found).[2]  
    return largest # Largest number is returned once no more comparisons are available for iteration.  
  
print(finder([1916, -1916, 250, 2019, 4, 3.14])) # Test any input array  
  
2019
```

This iterative method was also found to be more efficient than the recursive version. For example, when executed it takes notably less running time and operative steps to complete compared to executing the recursive method.

Q3.

```
def contains_duplicates(elements):  
    for i in range(0, len(elements)):  
        for j in range(0, len(elements)):  
            if i == j: # avoid self comparision  
                continue  
            if elements[i] == elements[j]:  
                return True # duplicate found  
    return False  
  
test = [10,0,5,3,-19,5]  
print(contains_duplicates(test))
```

True

Q3(a).

In the best case, what is the fastest our algorithm can take to run?

The answer is based on the minimum time the algorithm will take to produce outputs, expressed in the size of the inputs. For the `contains_duplicates()` function, we have a relatively small input size ($n = 5$), and it follows that the best-case time complexity in this case would occur if for example, the first two integers in the array were identical, e.g. if index [0] was “1” and index [1] was also “1”. In this example, a duplicate would be found instantly. It follows that the best case time complexity for this method would therefore be constant i.e. **$O(1)$** . Here, no matter how many elements are present in the array, the method will still execute in constant time.

That said, in reality, the best-case rarely occurs and it is worth noting that in certain situations one could put forward the argument that the best-case notation could in fact be different. In the best case an algorithm can run (it’s lower bound), the minimum time it can take to run must be expressed in terms of the size of the inputs. When measuring the performance of an algorithm, we are interested in how the increase in input size will affect the growth rate of operations required. This factor will determine the algorithms time complexity. Considering the example above of the first two integers in the array being 1 and 1, only one search is carried out. However, it should be appreciated that in some other situations, because a minimum of n steps is needed to view all the elements in the array, and the fact that you need to look at each element at least once, it stands to reason that the minimum order of growth might not be constant, but instead $O(n)$, for example if the array is notably longer, or if the duplicates exist very far away from one another in the array, resulting in the function having to perform more operations until it locates them. For example, in the case where an algorithm loops at least once through an array using a for loop, it would typically be given an order of $O(n)$ complexity, because it is of simple linear growth. An algorithm is ‘linear’ when the number of operations increases linearly with the number of elements [1]. Indeed, in scenarios such as this one, constant time with a for loop (and even more so with two for loops) would be difficult to achieve, compared to that comprised of a fixed range of elements as in the above example of only 5 elements.

Q3(b).

In the worst case, what is the longest our algorithm can take to run?

As with the best case scenario, the answer is based on the maximum time the algorithm will take to produce outputs, expressed in terms of the size of the inputs. It follows logically that the worst-case time complexity in this case would occur if no duplicates existed in the array (or worse, having no matches in an extremely long array). In this instance, we would have to iterate over the entire input before we realized the number didn't exist. Having no duplicates present would therefore prevent the algorithm from running efficiently. Remember, there are two iterative nested loops in this algorithm. The algorithm works by looping through all distinct pairs of indices $i < j$, checking to see if any of these pairs match. The first iteration of the outer loop (i), causes $n-1$ iterations of the inner loop, while the second iteration of the out loop causes $n-2$ iterations of the inner loop and so on [3]. Therefore, the worst-case running time is proportional to $O(n^2)$ notation i.e. the performance is directly proportional to the square of the input data.

The execution time in this case depends on the square of the number of elements in the array, and because there are two loops, it needs a higher growth order to execute. The function will essentially loop over each element/index in “**elements**” and then loop through it *again* resulting in it having $O(n*n)$ (or $O(n^2)$) complexity. This means that if the input range changed from 10 to 1000, then the number of operations (total loop iterations) increases as a square of the number of elements [1]. This is known as *Quadratic* growth, because the number of operations becomes the *square* of the number of elements in the array. This class of complexity is common with algorithms that involve inner and outer iterations over the input data such as these nested for loops. We will demonstrate exactly this process and how the total number of operations becomes squared as a result in the next question.

Q3(c).

Below, is the modified version that returns the number of comparisons the method makes between different elements until a duplicate is found.

```
def contains_duplicates(elements):
    count = 0 # Set count variable initially to zero.

    for i in range(0, len(elements)): # "0" is the index position of the integer in the array; "i" iterates over the array.
        for j in range(0, len(elements)): # "j" iterates over array; "i" incrementally moves up and "waits" for j to compare.
            count += 1 # Adds one to the count after a comparison is made. 36 comparisons are made in both inner and outer loops.
            if i == j: # Avoid self comparison.
                continue
            if elements[i] == elements[j]: # If condition to determine if a duplicate is present in the array.
                print(count, "comparisons till duplicate found"); count += 1; continue # Adds one to count if duplicate is found.
                                                    # 2 comparisons are made here.

    print(count, "comparisons made in total") # Prints total comparison count.

contains_duplicates([10,0,5,3,-19,5])
```

< >

```
18 comparisons till duplicate found
34 comparisons till duplicate found
38 comparisons made in total
```

We can see that it took 18 comparisons until a duplicate was found, and 34 for the next. We can also see that a total of 38 comparisons were made while running the function. But what exactly is going on within this process and why does it seem as though there *two sets* of duplicates found?

This doesn't make intuitive sense at first, but the fact is there *aren't* two sets of duplicates. Instead, it boils down to the fact that there are two *loops*. We will investigate what is going on.

To begin with, we must be cognizant of the fact that this function is not a recursive one but an iterative one comprised of two nested for loops. Unlike the previous functions, here there is only one initial call of the function to the stack, and one final return of that call to pop it back off the stack once the function has terminated.

To begin, **contains_duplicates()** is called with the array list as its input for the "**elements**" parameter and pushed onto the stack. Next the function drops to the second line to acknowledge that the **count** variable is set to zero. The function then begins to execute on the first for loop (outer loop) which uses "**i**" to iterate over the index array. Here, **i** is set to index [0]. Now the function drops to the inner for loop which uses **j** as its iterator and begins at the same index[0]. This is where the first comparison count is made and thus **1** is added to the count variable. Now the function goes to the first **if** condition to see if **i** is the same as **j** in

order to avoid self-comparison. In this case the elements *are* the same i.e. both iterators are at index[0] and thus both holding the number **10** as their current number. Therefore, the function traverses back to the inner for loop (**j**) and moves to the next index of **j** which is [1] (number **0**). While this is happening the iterator of the outer loop (**i**) remains “paused” on index [0] while it is compared for the second time, and another **1** is added to the count (which is now **count = 2**).

Again, the function drops to the first **if** condition to check for self-comparison, which this time doesn’t register as true, so it drops to the second **if** condition to check to see if **i** and **j** are duplicates. They are not, and the method again returns back up to the inner for loop (**j**) to move up to the next index i.e. index [2]. Again, **i** stays on index [0] as a comparison is made and a count is added. Now the method once again checks the first **if** condition and then the second **if** condition for self-comparison and duplicates respectively. It should be noted that the function is operating in a manner where the inner loop **j** runs before the outer loop **i**, this is the natural order of nested looping. Essentially **j** is doing all the hard work at the moment while **i** “waits” to move up to the next index. The function continues like this until **j** has reached index [5] i.e. the sixth number (**5**) while **i** stays on index [0]. Once this has happened, a total of **6** is on the counter variable (thus six comparisons so far). Now the function moves to the outer loop (**i**) and shifts the index [1] of **i**. Now the inner loop **j** returns back to its own index [0], 1 is added to the count (now at 7) and the process of the inner loop (**j**) hopping along the array while the outer loop (**i**) waits on index [1] for comparisons starts again.

A count of **18** has accrued when the iterators first determine the duplicate at (**i**) index [2] and (**j**) index[5], and when they encounter the duplicate again at (**i**) index[5] and (**j**) index[2], the counter has reached a value of **34**. We can see this fact in the first two outputs above.

To reflect, each time the inner (**j**) completes its iteration of the array, the function moves to the outer loop (**i**) to move that index up one as well. During each of these iterations of **j**, the outer loop stays “paused” on the **i** index, and this results in a total of **6** comparisons made. This process occurs a total of six times due to the fact there are 6 elements in the array. But there is also something else intriguing that happens during these 6 sets of comparisons that can go unnoticed. Because the array of elements is technically looped over twice (by the inner and outer loops), an interesting thing occurs. Just like a clock in a house will show the same time twice in a 24-hour day, so too does the function encounter finding a duplicate **TWICE** during its execution. This is because at one point in the function the iterator **i** is at index [2] while iterator **j** reaches index [5], which results in a duplicate (number 5) being found. This adds one to the count variable as it satisfies the second if condition of “**if elements[i] == elements[j]:**”. But that’s not all, because later, iterator **i** also reaches index [5] while iterator reaches index [2] and so the duplicate is actually found a *second* time. This occurrence therefore adds a total of **2** to the **count** variable plus the other **36** comparisons made on the array across the 6 elements (6 times). Now we can see that the phenomenon of *number of operations increasing as a square of the number of elements* i.e. quadratic growth has occurred within the function. This results in a total comparison count of **38**.

So, the function is very interesting, but what will happen if we change the input to comprise an instance where no duplicates exist within the array?

In this situation, would the function make more or less comparisons than the previous one made? Let's investigate this by changing the second duplicate instance of 5 to a 7 and run the function again:

```
def contains_duplicates(elements):
    count = 0 # Set count variable initially to zero.

    for i in range(0, len(elements)): # "0" is the index position of the integer in the array; "i" iterates over the array.
        for j in range(0, len(elements)): # "j" iterates over array; "i" incrementally moves up and "waits" for j to compare.
            count += 1 # Adds one to the count after a comparison is made. 36 comparisons are made in both inner and outer loops.
            if i == j: # Avoid self comparison.
                continue
            if elements[i] == elements[j]: # If condition to determine if a duplicate is present in the array.
                print(count, "comparisons till duplicate found"); count += 1; continue # Adds one to count if duplicate is found.
                                                    # 2 comparisons are made here.

    print(count, "comparisons made in total") # Prints total comparison count.

contains_duplicates([10,0,5,3,-19,7])
```

36 comparisons made in total

This version and the previous version are written with the exact same code, yet due to the input instance return notably different outputs. This time the output of comparisons is a total of 36. The reason for this is fairly obvious. In this scenario the function doesn't meet the requirements of the second if statement which is only satisfied if a duplicate has been found. We know there is no duplicate in the array this time, therefore, it will not add the final comparisons of the duplicate elements to the count variable. This is also the reason we don't see the same two outputs of 18 and 34 printed like previous example which found the duplicates. In this case the counter under the second if statement will only be activated and printed if the if condition is found to be true, which in this case does is not.

Q3(d). Below is an example of an input instance with 5 elements for which this method would exhibit its best-case running time:

In the example below, the “contains duplicates” function is comprised of 5 elements, which increase from zero to three as the index goes from [0] to [4], but with the first two elements instances the same. Each element in the array is a whole number and they increment by one from the second element onward i.e. from the second instance of zero. Because the first two numbers in the array are both zero, a duplicate will be found almost instantly in this case.

As we can see below, the execution speed of the function has been timed using Jupyter Notebook’s **%%timeit** command. The **%%timeit** command will automatically determine the execution time of a script/algorithm that follows its own input position at the beginning of the script. Unlike most timing methods which return a more standard “start time” minus “end time” equation for runtime, the **%%timeit** method is averaged over a number of runs. Essentially, it will time whatever you evaluate multiple times and then return the best, *and* average times (M and SD).

Looking below, the function can be interpreted as taking; 250 microseconds (plus/minus 12500 nanoseconds) per loop, with a mean (M) and standard deviation (SD) of 7 runs at 10000 loops each. To clarify, there are *one million* microseconds in one second, so in this case the 250 microseconds equals 0.00025 seconds. There are *one thousand* nanoseconds in a microsecond.

```
%%timeit
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:
                continue
            if elements[i] == elements[j]:
                return True
    return False

test = [0,0,1,2,3] # Best-case running time version
print(contains_duplicates(test))
```

```
True
250 µs ± 12.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Q3(e). Below is an example of an input instance with 5 elements for which this method would exhibit its worst-case running time:

In the example below, the `contains_duplicates` function is comprised of 5 elements, which increase somewhat exponentially as the index goes from [0] to [4]. Each element in the array is notably distinct from the previous and therefore it will fail to find a duplicate for any of the elements. I have timed the execution speed and it can be interpreted as taking 309 microseconds (plus/minus 70300 nanoseconds) per loop, with a mean (M) and standard deviation (SD) of 7 runs at 1000 loops each. This running time appears to be notably slower than that of the previous input array i.e. [0,0,1,2,3].

```
%%timeit
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:
                continue
            if elements[i] == elements[j]:
                return True
    return False

test = [1,23,456,789,101112] # Worst-case running time version
print(contains_duplicates(test))
```

```
False
309 µs ± 70.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

With this worst case, the fact that the input properties are increasing exponentially as the array moves forward would prevent the algorithm from running efficiently. We know that for any particular value of **n** (in this case the elements of our array), that the amount of work done by the algorithm can vary depending on the instances of size **n**. So, in this case, the number of individual data elements in the data array ostensibly takes longer to process than that of the previous array of [0,0,1,2,3]. It should be noted that in this case we are limited to inputting only 5 elements in the array to demonstrate the worst-case scenario. However, if for example we were permitted to input a larger number of individual elements, we could argue that the worst case time example could be that of having a very long array (say 500 elements) but with a matching number at both the very beginning (index[0]) and very end (index [499]) of the array. Here, the algorithm would be successful yet take a longer runtime due to the fact that the number of operations or work done by the algorithm has increased dramatically due to the fact that the instances of size **n** have also increased dramatically. This input of **n = 500** would affect the square of the input size rather exponentially resulting in the two nested loops having to handle a far heavier workload before the function is executed, due to the fact the number of operations would become the square of the number of inputs.

Q3(f).

First input instance:

```
%%timeit
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:
                continue
            if elements[i] == elements[j]:
                return True
    return False

test = [10,0,5,3,-19,5] # First input instance
print(contains_duplicates(test))
```

True
282 μ s \pm 28.7 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

Second input instance:

```
%%timeit
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:
                continue
            if elements[i] == elements[j]:
                return True
    return False

test = [0,1,0,-127,346,125] # Second input instance
print(contains_duplicates(test))
```

True
245 μ s \pm 8.88 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

Looking at the two outputs, we can see that the first input instance takes longer for this method to process. Why is this?

It follows logically that because the first input array duplicates are in index positions [2] and [5], while in the second array the duplicates are closer together at [0] and [2], this would result in the duplicates in the second array being found faster and the method terminating in a faster execution time than that of the first array. We can see that the first input array executed at a runtime speed 282 microseconds while the second took 245 microseconds. As we have seen from Q3(c) above, we know that the inner loop **j** will reach the indexed numbers in the array before the outer **i** loop does. By this logic, the duplicates in the second example will be found at comparison count **3**, whereas they won't be found until comparison count **18** with the first example.

- There are two files that accompany this Word document: An **.ipynb** file and a **.py** file
- Jupyter Notebook was used for all querying and coding processes (**Problem_sheet_Gareth_Duffy_g00364693.ipynb**)
- A Python file was also created to store all code examples (**Problem_sheet_Gareth_Duffy_g00364693.py**)

References:

- [1]: Mc Donnell, M. (2019). *Algorithmic Complexity in Python*. Retrieved from: <https://www.integralist.co.uk/posts/algorithmic-complexity-in-python/#constant-time>
- [2]: Meehan. S. (2015). *Maximum number in a list using loop in Python*. Retrieved from: <https://stackoverflow.com/questions/31901171/how-to-print-maximum-number-in-a-list-using-loop-in-python>
- [3]. Goodrich, M., T., Tamassia, R., & Goldwasser, M., H. (2013). *Data Structures & Algorithms in Python*. New Jersey: Wiley