|  |  |
| --- | --- |
| **COMP08033 Computational Thinking with Algorithms, Semester 2 2018/2019** | |
| **Document Title:** | Algorithms Problem Sheet (Python) |
| **Student No.:** | G00364778 |
| **Student Name:** | Gerhard van der Linde |

**Algorithms Problem Sheet (Python)**

# Question 1

## 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.

Running the code:

1. def mystery(n):
2. print(n)
3. if n < 4:
4. mystery(n+1)
5. print(n) 6.

7. mystery(1)

outputs the following results:

λ python Q1\_mystery.py 1

2

3

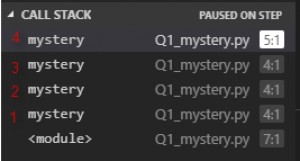
4

4

3

2

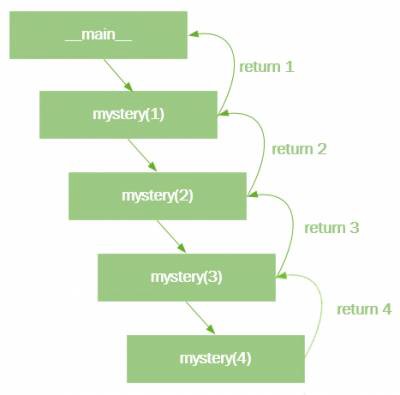
1



The diagram above show the stack on the python debugger output.

At ﬁrst running the code was confusing as the output did not make much sense since the variable *n* seems to decrement mysteriously once equal to four, however on closer inspection the key point to observe in the code sample “Q1\_mystery.py” line 4, the function *mystery* calls itself repeatably inside the function and adds another instance of the function *mystery* on

top of the runtime call stack.



This results in four separate calls to the function *mystery*, piling up on the runtime stack, all holding the last value of *n* in the call to the function *mystery*. The call frames concludes with the print function on line ﬁve and popped from the stack in reverse order [1)](#_bookmark0) and printing the value of *n* that was initially passed into the function *mystery* until the stack is cleared and the function terminates.

# Question 2

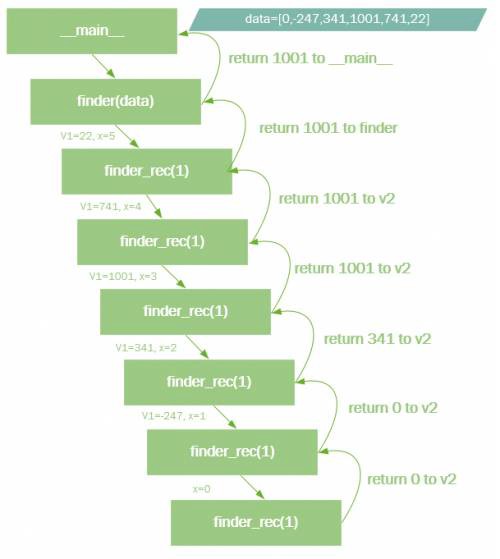
## Q2 (a) What value is returned by a call to ﬁnder when the following array is used as input? (1 mark)

The Value returned is: 1001

## Q2 (b) What characteristic of the input data set does the ﬁnder method determine? How does it determine this result? (3 marks)

The ﬁnder function ﬁnds the biggest number in the data array.

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



The data array above is passed into the ﬁrst ﬁnder function and the passed into ﬁnder\_rec using two values, the data array and the length of the array. The function funder\_rec is then called recursively, calling itself until the all the values in the array is placed on the stack. The last call to ﬁnder\_rec returns data[0] to V2, i.e. the value of 0, then the biggest value between -247 and zero is returned to the next stacked ﬁnder \_rec that compares the returned value with the last value stored in v1 on the stack. The next cycle compares 341 and zero and again returns the biggest of the two. This repeats until the biggest number in the array is ﬁnally returned, i.e. 1001.

## Q2 (c) Can you add some inline comments to the code above to explain how it works? (2 marks)

def finder(data):

*# call finder\_rec function with data array and the length of the array*

*# and return the final answer from finder\_rec to the calling function*

return finder\_rec(data, **len**(data)-1)

def finder\_rec(data, x):

*# if the arra index is zero, return the value in the first position for comparison*

if x == 0:

return data[x]

*# store the current value from the array in v1 for comparison*

v1 = data[x]

*# store the biggest of the two variables from the last call to finder\_rec in v2*

*# for comparison when the call returns*

v2 = finder\_rec(data, x-1)

*# compare the two values stored in v1 and v2 and return the biggest value for*

*# the next comparison step, the biggest of v1 or v2 will be returned to v2 from*

*# previous call to finder when popping off the stack*

if v1 > v2:

return v1 *# if v1 is bigger than v2 return v1*

else:

return v2 *# if v2 is bigger than v1 return v2*

if name == ' main ':

*# define the array of values to pass in to finder*

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

*# call finder with the dataset created above and return the result to retval*

retval = finder(dat)

*# print retval to the console output*

print('The Value returned is:', retval)

**Q2 (d) Write a method which achieves the same result as ﬁnder, but which uses an iterative approach instead of recursion. (2 marks)**

def return\_the\_biggest(data):

*# assign the first item in the list to the variable "largest"*

largest=data[0]

*# now iterate through the list assigning one value at a time to testval*

for testval in numlist:

*# compare the values and assign the biggest to the variable "largest" if bigger*

if testval > largest: largest=testval

*#finally return the result*

return largest

if name == ' main ':

*# define the list of number to process*

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

retval = return\_the\_biggest(numlist)

print('The biggest value in the list is:',retval)

The output of the code in the terminal is:

The biggest value in the **list** is: 1001

# Question 3

## Q3 (a) What is the best-case time complexity for this method, and why? (2 marks)

The best case time complexity is achieved when the ﬁrst two element in the array is equal.

1. import time 2.

1. def contains\_duplicates(elements):
2. for i in **range** (0, **len**(elements)):
3. for j in **range**(0, **len**(elements)):
4. if i == j: *# avoid self comparision*
5. continue
6. if elements[i] == elements[j]:
7. return **True**
8. return **False**

11.

1. if name == ' main ':
2. testarr=[i for i in **range**(0,5001)]
3. testarr[1]=0 *# set the second element in the array to the value zero so it matches the first.*
4. start = time.time()
5. result = contains\_duplicates(testarr)
6. end = time.time()
7. print("duplicates found: ", result, "- Time taken:", (end-start)\*1000, "ms")

The code executes and completes almost instantaneously when given an array of 5000 points to compare. The reason it completes so fast is that the function returns true as soon as a duplicate is found and stops comparing the rest of the array.

## Q3 (b) What is the worst-case time complexity for this method, and why? (2 marks)

The worst case time scenario is achieved when there are no duplicates in the list.

*# testarr[1]=0*

1. import time 2.

1. def contains\_duplicates(elements):
2. for i in **range** (0, **len**(elements)):
3. for j in **range**(0, **len**(elements)):
4. if i == j: *# avoid self comparison*
5. continue
6. if elements[i] == elements[j]:
7. return **True**
8. return **False**

11.

12. if name == ' main ':

13.

14.

15.

16.

17.

18.

testarr=[i for i in **range**(0,5001)]

start = time.time()

result = contains\_duplicates(testarr) end = time.time()

print("Duplicates found: ", result, "- Time taken:", (end-start)\*1000, "ms")

The timing of the function is represented by the formula



|  |  |  |
| --- | --- | --- |
|  |  | **Actual Test Case timing** |
| **Best Case** |  | < 1 ms |
| **Worst Case** |  | 5713.1910 ms |

So for no duplicates the entire array is stepped through in one loop and scanned against every value in the array as shown in the complexity formula, resulting in an exponential increase in time based on the number of values searched.

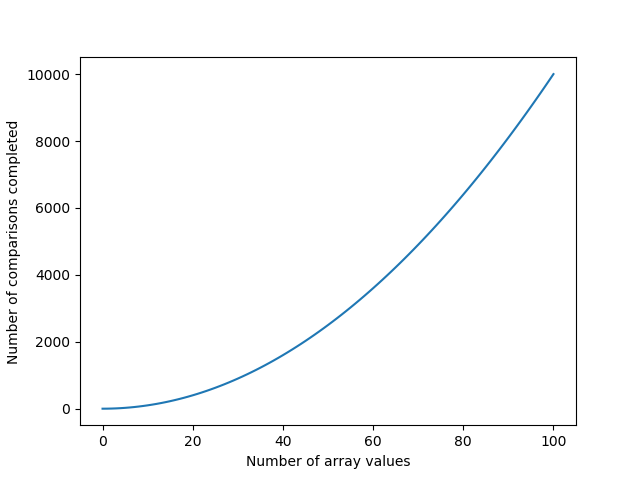


The python code below graphically illustrates the nature of the exponential growth of the code searching for duplicates.

import matplotlib.pyplot as pl

exp=[i\*\*2 for i in **range**(0,101)] pl.plot(exp)

pl.xlabel('Number of array values') pl.ylabel('Number of comparisons completed') pl.show()



## 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. (2 marks)**

1. import time 2.

1. def contains\_duplicates(elements):
2. for i in **range** (0, **len**(elements)):
3. for j in **range**(0, **len**(elements)): 6. steps=(i+1)\*(j+1)
4. if i == j: *# avoid self comparison*
5. continue
6. if elements[i] == elements[j]:
7. *#return True, steps*
8. return steps
9. *#return False, steps*
10. return steps 14.
11. if name == ' main ':
12. testarr=[i for i in **range**(0,5001)] 17. *#testarr[0]=1000*
13. start = time.time()
14. result = contains\_duplicates(testarr)
15. end = time.time()
16. *#print("Duplicates found:", result, "- Execution Time:", (end-start)\*1000, "ms")*
17. print("Duplicates found in stepcount:", result, "- Execution Time:", (end- start)\*1000, "ms")

The code in line 6 calculates the step value by ﬁrst adding one to the zero based indexes, lines 11 and 13 then returns the step values calculated and returns the step values instead of the booleans.

Lines 10 and 12, commented out returns both, i.e. the boolean states and the counters.

## Q3 (d) Construct an input instance with 5 elements for which this method would exhibit its best-case running time. (1 mark)

testarr=[1,1,3,4,5]

Duplicates found in stepcount: 2 - Execution Time: 0.0 ms

The best case scenario as illustrated in the code snippet above with ﬁve element are where the ﬁrst two elements of the ﬁve elements match. The code will then complete in two cycles.

## Q3 (e) Construct an input instance with 5 elements for which this method would exhibit its worst-case running time. (1 mark)

For worst case timing the array with ﬁve element will look like this:

testarr=[1,2,3,5,5]

The terminal output running the array with worst case element will result in 25 cycles.

Duplicates found in stepcount: 20 - Execution Time: 0.0 ms

## 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? (1 mark)**

test1=[10,0,5,3,-19,5]

test2=[0,1,0,-127,346,125]

In test case one values 3 and 6 matches, so the code will complete 18 cycles to ﬁnd the match.

In test case two values 1 and three matches, so only three cycles will be completed before a match is found.

if name == ' main ': test1=[10,0,5,3,-19,5]

test2=[0,1,0,-127,346,125]

start = time.time()

result = contains\_duplicates(test2) end = time.time()

print("Duplicates found in stepcount:", result, "- Execution Time:", (end-start)\*1000, "ms")

The terminal outputs running the code above conﬁrms the results.

Duplicates found in stepcount: 18 - Execution Time: 0.0 ms Duplicates found in stepcount: 3 - Execution Time: 0.0 ms

[1)](#_bookmark0)

[https://www.cs.ucsb.edu/~pconrad/cs8/topics.beta/theStack/02/](https://www.cs.ucsb.edu/%7Epconrad/cs8/topics.beta/theStack/02/)

Last update: **2019/03/10 17:41**