cycle_5_lecture_notes_complexity

February 7, 2021

Running time and Complexity

Today we'll be taking first steps to both understanding and formally reasoning about the time our programs take to run. We will: 1. Trace the computation of code 2. Plot the running time of code 3. Introduce big-O notation

At the end of this series of lectures and labs, I would like you to be able to: 1. Look at a plot of code running time and say what the likely big-O complexity is 2. Predict a plot from a piece of code 3. Determine the big-O complexity of a piece of code 4. Given the big-O complexities of several pieces of code, know how to combine them if the code is combined

We will then use this knowledge to analyse searching and sorting algorithms in the next units.

Let's look at a few simple functions:

[1]:
```
def findSum(myList):
    sum = 0
    for element in myList:
        sum = sum + element
    return sum

def findSumOdd(myList):
    sum = 0
    for element in myList:
        if element % 2 != 0:
          sum = sum + element
    return sum

def findSumAfter(myList):
    newList = []
    for i in range(len(myList)):
            thisSum = 0
            for j in range(i, len(myList)):
                thisSum = thisSum + myList[j]
            newList.append(thisSum)
    return newList
```

Let's trace *in enormous detail* how many operations the interpreter does when we call this with (I'll do this on paper in the recording):

[2]:
```
findSum([1, 2, 3])
findSumOdd([1, 2, 3])
print(findSumAfter([1, 2, 3]))
```

```
[6, 5, 3]
```

Carefully counting the operations our code needs is one way we can look at how long our code will take to run. We could do this, but it's likely to be a lot of work. We will move toward a less-precise version of this in the form of big-O notation. What we're really interested in is how the time it takes to run our code as the input gets bigger.

[3]:
```
import time

inputLists = []
for i in range(100, 1000, 10):
    inputLists.append([1]*i)

for test in inputLists:
    t = time.process_time()
    findSum(test)
    elapsed_time = time.process_time() - t
    print(str(len(test)) + "  " + str(elapsed_time))
```

```
100  6.999999999979245e-06
110  4.999999999977245e-06
120  1.0000000000010001e-05
130  6.999999999979245e-06
140  4.000000000004e-06
150  5.000000000032756e-06
160  1.2000000000012001e-05
170  4.999999999977245e-06
180  4.999999999977245e-06
190  6.0000000000060005e-06
200  6.0000000000060005e-06
210  8.999999999981245e-06
220  1.0000000000010001e-05
230  1.2999999999985246e-05
240  6.0000000000060005e-06
250  8.000000000008e-06
260  1.4999999999987246e-05
270  1.199999999995649e-05
280  8.000000000008e-06
290  1.2999999999985246e-05
300  8.999999999981245e-06
310  1.2999999999985246e-05
320  1.0000000000010001e-05
330  1.4999999999987246e-05
```

340 1.4000000000014001e-05
350 1.0000000000010001e-05
360 3.0000000000030003e-05
370 1.599999999996049e-05
380 1.4999999999987246e-05
390 1.4000000000014001e-05
400 1.4000000000014001e-05
410 1.4000000000014001e-05
420 1.6000000000016e-05
430 1.5000000000042757e-05
440 1.6000000000016e-05
450 1.4000000000014001e-05
460 3.999999999998449e-05
470 4.999999999999449e-05
480 4.2000000000042004e-05
490 1.6999999999989246e-05
500 2.2000000000022002e-05
510 1.799999999996249e-05
520 2.399999999996849e-05
530 1.6999999999989246e-05
540 1.9000000000046757e-05
550 1.8000000000018e-05
560 1.8000000000018e-05
570 2.0000000000020002e-05
580 1.9000000000046757e-05
590 1.999999999996449e-05
600 2.0000000000020002e-05
610 2.2000000000022002e-05
620 2.2000000000022002e-05
630 4.500000000001725e-05
640 0.000136000000000025
650 2.2999999999995246e-05
660 2.399999999996849e-05
670 2.4999999999997247e-05
680 2.4000000000024002e-05
690 2.399999999996849e-05
700 2.999999999997449e-05
710 2.5000000000052758e-05
720 3.500000000000725e-05
730 2.6999999999999247e-05
740 2.5000000000052758e-05
750 4.099999999995774e-05
760 3.100000000000325e-05
770 0.000160000000000049
780 2.8000000000028002e-05
790 2.6999999999999247e-05
800 2.8000000000028002e-05
810 2.8999999999945736e-05

```
820  2.8000000000028002e-05
830  2.8999999999945736e-05
840  2.9000000000056758e-05
850  2.8999999999945736e-05
860  2.999999999997449e-05
870  2.9000000000056758e-05
880  3.300000000006076e-05
890  2.9000000000056758e-05
900  3.2999999999949736e-05
910  3.2000000000032e-05
920  3.100000000000325e-05
930  3.399999999997849e-05
940  3.100000000000325e-05
950  3.399999999997849e-05
960  3.399999999997849e-05
970  3.399999999997849e-05
980  3.2999999999949736e-05
990  3.199999999992098e-05
```

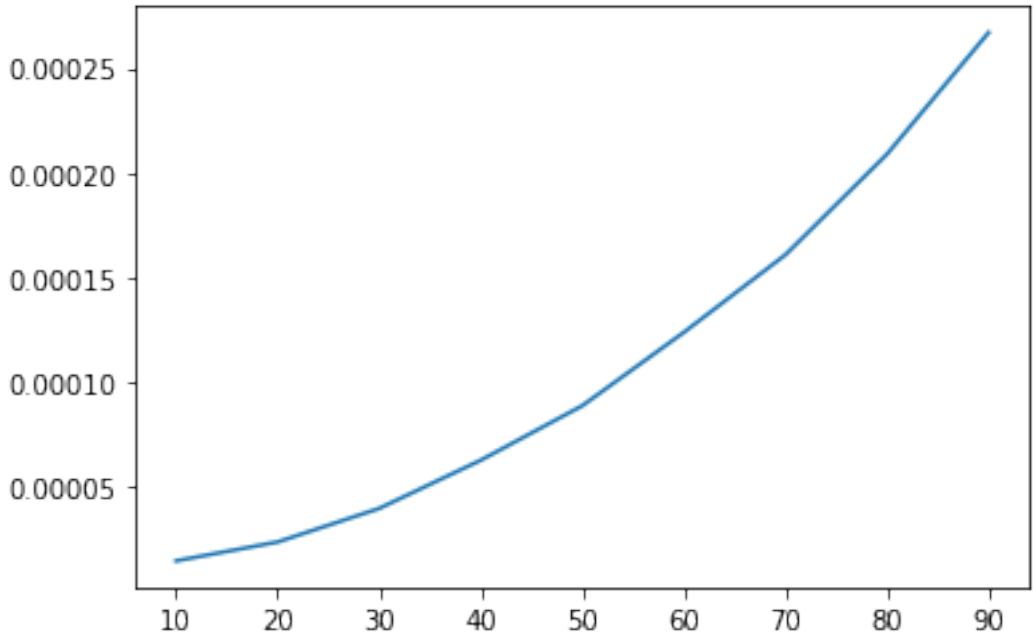
[36]:
```python
import matplotlib.pyplot as plt

def plotRuntimes(func, listOfInputs):
    times = []
    sizes = []
    for inputItem in listOfInputs:
        sizes.append(len(inputItem))
        t = time.process_time()
        func(inputItem)
        elapsed_time = time.process_time() - t
        times.append(elapsed_time)
    plt.plot(sizes, times)
    plt.show()

inputLists = []

for i in range(10, 100, 10):
     inputLists.append([1]*i)

plotRuntimes(findSumAfter, inputLists)
```



If we want to be more analytical, the idea of big-O notation can help us.

What is big-O notation?

If f(n) is a function, and n is the size of the input to some code or algorithm, then we say that our code is O(f(n)) if, for a big enough n, the runtime of our code is upper-bounded by k*f(n)

Let's look at some of our examples. What are the big-O complexities of our functions from before?

<we will work through each in detail in the video, I'll place answers below>

[4]:
```
def findSum(myList):
    sum = 0
    for element in myList:
        sum = sum + element
    return sum


def findSumOdd(myList):
    sum = 0
    for element in myList:
        if element % 2 != 0:
          sum = sum + element
    return sum

def findSumAfter(myList):
    newList = []
    for i in range(len(myList)):
```

```
            thisSum = 0
            for j in range(i, len(myList)):
                thisSum = thisSum + myList[j]
            newList.append(thisSum)
    return newList
```

To find the big-O complexities of our code, we need to argue about how long the code will take as the size of the input increases. After our video-worked examples, we find that the above have: - findSum is O(n) - findSumOdd is O(n) - findSumAfter is O(n2)

Can you think of a piece of code that would be O(1)? How about O(n2)?

Later in the course we'll see an example of O(log n)

Let's talk about how we combine the big-O running times of pieces of code.

First: because we are talking about *very large inputs (asymptotics)*, we only care about the dominant term.

E.g.: if a<b, then O(na) + O(nb) gives us O(nb)

eg: - O(n2) + O(n) gives us O(n2) - O(n4) + O(n5) gives us O(n5)

So! What about if we call our various functions in combination?

Do these as an exercise - what is the big-O complexity of each? - findSum twice - findSum and then findSumOdd - findSumOff and then findSumAfter

[ ]: