DVA245 Lab Assignment 2
Recursion, Time Complexity

Anna Friebe

January 2021

1 Recursion

In the first part of the assignment you will write a recursive program that evaluates whether the input string is a palindrome. A palindrome is a string that you can read from left to right or from right to left and get the same result, such as "Anna" or "Naturrutan".

Download files from Canvas that provide tests and a skeleton of your recursive function.

Recall that you need to

1. Find the base case for the recursion - a simple case of input that can be solved easily without recursion. (In the palindrome case we can think of two base cases.)

2. If the base case has not been reached - call the recursive function with a subset of the input making progress towards the base case

1.1 How to display

You need to run the tests in checkPalindrome.py and show that they pass.

2 Time complexity

In the second part of the assignment you will write a report on time complexity. In the file `max_subsequence.py` there are three different implementations, `max_subsequence1`, `max_subsequence2` and `max_subsequence3` of a function that:

- Takes a list of numbers as input.
- Returns the maximum sum of a contiguous subsequence of the list.

In Fig. 1 you see an illustration of the `max_subsequence` functionality.

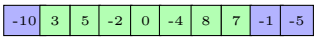

Figure 1: An example list. The output of the `max_subsequence` methods is the sum of the green section of contiguous elements, 17.

2.1 Report content

You shall write a report, where for each of the `max_subsequence` functions you:

- Perform a (Big-Oh) time complexity analysis of the function's code.
- Display a graph of the time taken for the function call for different lengths of the input list.
- Discuss if the graph looks like expected from the analysis.

We are interested in the order of growth of the time - the questions:

- How much longer does it take to run this function when the input list is twice as long?
- What mathematical function describes the running time as a function of the input list size?

The report shall be written in English.

2.1.1 Report sections

The sections to include in the report are:

1. Introduction (short)
2. max_subsequence1
   (a) Algorithm analysis
   (b) Experimental results
   (c) Discussion
3. max_subsequence2
   (a) Algorithm analysis
   (b) Experimental results
   (c) Discussion
4. max_subsequence3
   (a) Algorithm analysis
   (b) Experimental results
   (c) Discussion
5. Conclusion (short)

2.2 Details

The lengths of the lists will need to be different for the different functions you are testing, as they are not equally efficient. Suitable lengths will vary with your system, and whether you are running on repl.it or locally. Use at least five different list lengths for each function, and a range so that the longest list for a function is at least 4 times as long as the shortest. In this way you will be able to see in the graph what happens to the time as the list length doubles twice.

In order to do this, you must write a program that measures the time taken for the calls to the different functions, with different lengths of the input list.

Tips:

- To create lists of different length you may want to use the `sample` function from the `random` module. Note that the input sequence to the `sample` function needs to be at least as long as the length argument. In the following example `sampleList` has length 1001 000 (a list of 1001 items repeated 1000 times), so the listLength parameter can not be longer than 1001 000. Before using randomized functions such as sample, you may want to call the `seed` function. This will cause the randomized functions to give the same result each time they are run, for the same seed value, in this example 17:

```
random.seed(17)
sampleList = list(range(-500, 501))*1000
myList = random.sample(sampleList, listLength)
```

- To measure the time in the function call you can for example use the `time` or `process_time` functions in the `time` module to get the time before and after the function call. Try to find list lengths that give non-zero results, but short enough so that you don't need to wait for a long time. There are many reasons for a single measurement to deviate from the expected. This can be other processes running on the system and interfering, the random content of the list to cause more or less time consuming calculations or memory related overheads in the case of long lists. So don't be too worried if one of the graphs does not look like you expected from the algorithm analysis of the code.

- You can pass a function as a parameter to avoid code duplication, and implement a helper function to measure the time for different function calls with different list lengths.

```
def testTimingHelper(fun, ...):
    ...
    # time measuring
    fun(myList)
    # time measuring
    # return resulting time
```

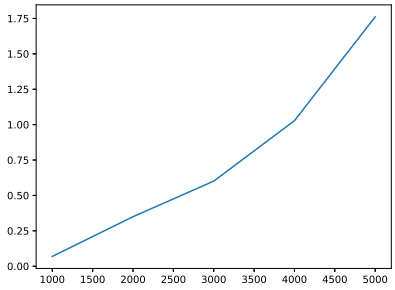

Figure 2: Example graph for one of the max_subsequence functions with list lengths on the x-axis and time in seconds on the y-axis. Here the lists have lengths 1000, 2000, 3000, 4000 and 5000.

```
# in some loop for each list length for max_subsequence1
resultingTime = testTimingHelper(max_subsequence1, ...)
```

You can draw your graphs with python, for example the pyplot interface matplotlib module can be used in this way:

```
from matplotlib import pyplot as plt

plt.figure()
plt.plot(xValues, yValues)
plt.show()
```

An example of a graph visualized with matplotlib is shown in Figure 2.
If you prefer you can draw the graphs in another way.