DVA245 Lab Assignment 2
Recursion, Time Complexity

Anna Friebe

January 2023

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

The resulting code is short, less than 10 lines of code.

1.1 How to display

You need to run the tests in check_palindrome_test.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.

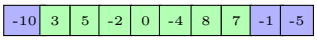

Figure 1: An example list. The output of the `max_subsequence` methods is the sum of the green section of contiguous elements, 17.

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. Remember to label the axes, and specify the time unit for the y-axis.
- 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 Timing tests

Create the helper functions for the timing test from the pseudo code given in this section.

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.

In algorithm 1 you find pseudo code for a function that creates a list of a specified length, *listLength*, and measures the time it takes in the call to a function fun with this list as the argument.

Some lines in algorithm 1 contain python code to simplify for you. Two python modules are used, `random` and `time`. On line 10, the `sample` function from the `random` module is used to create a list of length *listLength* with integers sampled from the range $[-1000000, 1000000]$. The *stop* argument of the python `range` function, 1000001 is not in the interval. The `sample` function samples each item from the input range/ list at most once, so if you want to use lists longer than 2000001 items, you will need to increase the size of the range that is used. The `process_time` function of the `time` module is used to get the time before and after the function call. There are several other functions in this module that could have been used for this purpose.

Algorithm 1 Pseudocode for a test function that creates a list with a given size, and measures the time for a function call with this list as the argument. The list length must not be longer than 2000001.

```
1: function timeInFunctionCall(fun, listLength) returns time of the call to fun.
2:     inputs:
3:         fun, the function to call
4:         listLength, the length of the list to use in the call
5:     local variables:
6:         testList, the list to use in the function call
7:         timePrior, the time before the function call
8:         timeDuration, the time elapsed during the function call
9:
10:    testList ← random.sample(range(−1000000, 1000001), listLength)
11:    timePrior ← time.process_time
12:    fun(testList)
13:    timeDuration ← time.process_time − timePrior
14:    return timeDuration
15: end function
```

In algorithm 2 the function in algorithm 1 is called several times, with the same function argument, but with different list lengths. The resulting times are collected in a list and returned.

When you have implemented the functions in algorithm 1 and algorithm 2

Algorithm 2 Pseudocode for a test function that measures the time for calls to a function with lists of different lengths as the argument.

```
1: function timeInFunctionCalls(fun, listLengths) returns List of times of the calls.
2:     inputs:
3:         fun, the function to call
4:         listLengths, a list of lengths to use in the calls
5:     local variables:
6:         results, the list of times in the function calls
7:
8:     results ← empty list
9:     for each length in listLengths do
10:        results.append(timeInFunctionCall(fun, length)
11:    end for
12:    return results
13: end function
```

you need to find a number of suitable list lengths for each of the functions `max_subsequence1`, `max_subsequence2` and `max_subsequence3`. Then you can call `timeInFunctionCalls` with the function and the list of lengths.

Some advice:

- 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. That means that you can reproduce the same figure for example. In this example the seed is 17:

  ```
  random.seed(17)
  ```

- Try to find list lengths that give non-zero time results, but that are 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.

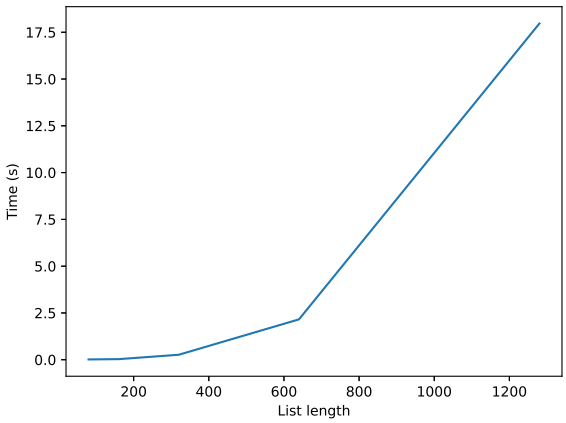

Figure 2: Example graph for one of the max_subsequence functions with list lengths on the x-axis and time in seconds on the y-axis. Here the lists have lengths 80, 160, 320, 640 and 1280.

2.3 Figure creation

You can draw your graphs with python, for example the pyplot interface matplotlib module can be used for this. Matplotlib is installed in the lab rooms. See `https://matplotlib.org/stable/users/installing/index.html` for installation on your system.

Matplotlib can be used in this way:

```
from matplotlib import pyplot as plt

plt.figure()
plt.plot(xValues, yValues)
plt.xlabel("My x axis label")
plt.ylabel("My y axis label (unit)")
plt.show()
```

The list lengths are used for the x values, and the times returned from `timeInFunctionCalls` are used for the y values. An example of a graph visualized with matplotlib is shown in Figure 2.

If you prefer you can draw the graphs in another way.