Advanced Topics in Comp Sci Project—Sorting Algorithm Analysis

ASSIGNMENT OVERVIEW

In this assignment, you'll implement a designated sorting algorithm, do a theoretical analysis of that algorithm's efficiency, collect actual performance data of your implementation, and compare your performance data with what you predicted.

BACKGROUND

Different algorithms have different efficiencies, often expressed using Big-O Notation. Common Big O-values for sort algorithms include

- $O(n^2)$ for a Bubble Sort
- $O(n^2)$ for a Selection Sort
- $O(n^2)$ for an Insertion Sort
- $O(n \log n)$ for a Merge Sort
- $O(n \log n)$ for a Quick Sort
- $O(n \log n)$ for a Shell Sort

These efficiencies don't reflect how quickly the algorithm works on a given machine with a given list of values. Big-O notation indicates how the execution time for an algorithm increases as a function of an increasing number of values.

For example, if a Bubble Sort algorithm on your computer takes 2 millisecond to sort 50 values, how much time would we predict it that it would take to sort 100 values? The number of values n that we're sorting has increased by a factor of 2, and Bubble Sort is an $O(n^2)$ algorithm, so we expect that the amount of time it will take increases by a factor of $(2)^2 = 4$, or about 4×2 milliseconds = 8 milliseconds.

A formal strategy for predicting an algorithm's efficiency is presented at the end of this document.

PROJECT SPECIFICATION

This project consists of four steps that you'll need to complete in preparation for submitting a final document, a formally written research report, submitted electronically as a PDF file. Once you have been assigned a sorting algorithm, you'll need to:

1. Implement the designated sorting algorithm using Python.
2. Develop a theoretical analysis of your sorting algorithm's efficiency using an analysis as outlined in class and at the end of this document.
3. Collect actual performance data of your implementation using Python's `time` module.
4. Compare your measured performance data from (3) with what you predicted in (2).

Your final report will include a formal presentation of the work that you've done, including:

0. **Abstract**
   A short one-paragraph summary of this project, what you did and what you figured out. The abstract is meant to be a complete overview of what happened, including "spoilers" regarding your findings. (In some cases, a reader may not get beyond the abstract, so you want to make sure that everything important gets mentioned, however briefly.)
1. **Introduction**
   An introduction to the report: a general description of what we'll be reading about, and some preliminary information about what sorting is in general, and some information leading into the next section focusing on your algorithm of interest.
2. **The Sorting Algorithm**
   A verbal description of your algorithm's sorting strategy. Additionally, your description of the

algorithm will almost certainly benefit from an appropriate diagram. Consider using either text-based or graphical representations of your sorting process. (See example below.)

3. **Source Code**
   The source code of your Python script implementing that sorting algorithm, with a liberal amount of documentation/comments.
   Present your code using a monospaced, fixed-width font such as `Courier`, `Consolas`, `Ubuntu Mono`, `Andale Mono`, etc.
   Comments should be written so they are clearly distinguishable from code (see example below).
4. **Big-O Analysis**
   The theoretical predicted analysis of the sort's efficiency, explained as much as is allowed after research.
5. **Performance Data**
   Data tables from your actual performance data collected by running your program. If you have a lot of data, don't include it all; a representative sample of your data will do just fine.
6. **Graphical Results**
   A graph of your data, with a regression if possible. If you need to graph *time vs. n log n* or something tricky like that, see below for advice on how to do that.
7. **Discussion**
   A brief evaluation of your results, further explanations or interpretations of your work, your data, or the process, or a drawing attention to things that you think are significant
8. **Summary**
   A summary of everything that happened, located at the end of the report. There should be nothing here that you haven't already mentioned before.
9. **References**
   A series of references to any sources used in the development of your project. Include people you worked with, webpages you looked at, textbooks you read, etc. Your citation style should mirror as much as possible that of the Association for Computing Machinery, at https://www.acm.org/publications/authors/reference-formatting .

The final report, including cover sheet, will probably be from 6-10 pages.

DELIVERABLES

`sorting_algorithm_analysis.pdf`

To submit your assignment for grading, copy `sorting_algorithm_analysis.pdf` to your directory in `/home/studentID/forInstructor/` at *crashwhite.polytechnic.org* before the deadline.

ASSIGNMENT NOTES

- Once you know which algorithm you'll be implementing, you'll need to identify a reference where you can learn about that algorithm. We'll be investigating the original versions of these algorithms, not versions that have been optimized by modifying them to run faster.
- *Write your own Python implementation of your sorting algorithm.* One of the intentions of this assignment is for you to get practice thinking about an algorithm and figuring it out how to implement it. It's easy to find Python versions of these algorithms online and/or in our book, but you shouldn't be using these programs as a reference.

Do not use other people's Python code as a reference in this assignment!

- Examining a pseudocode version of the algorithm may be useful if you get stuck, but your goal is to be able to write the program from scratch. Make sure you've given that a try first before looking at any pseudocode.
- Explaining your algorithm can be challenging, but diagrams can help out. You'll probably want to include a diagram like this one (just text-based, but you could use graphics) to help the reader understand how the algorithm works.

Insertion Sort algorithm:
1. Initialize a counter that will identify the sorted section of the array (s). All elements up to and including this element are sorted. In our example, we consider an array of 5 elements, identified by the indexes 0-4 shown here.

```
arr = [ 7 ] , [ 2 ] , [ 5 ] , [ 8 ], [ 4 ]
      ------------------------------------       sorted = 0
        0       1       2       3      4
```

2. Move to the next element in the array: we want to insert this value into its correct position in the array. We'll do this by checking it with the previous value, and swapping values backwards until this element arrives at its correct position (as long as this value is less than the position we're considering).

```
arr = [ 2 ] , [ 7 ] , [ 5 ] , [ 8 ], [ 4 ]
      ------------------------------------       sorted = 1
        0       1       2       3      4
insert_point = sorted
while insert_point > 0 and arr[insert_point] < arr[insert_point - 1]:
    # swap values
    arr[insert_point], arr[insert_point - 1] = \
    arr[insert_point - 1],arr[insert_point]
    insert_point = insert_point - 1
```

3. Continue until reaching the end of the array

```
arr = [ 2 ] , [ 5 ] , [ 7 ] , [ 8 ], [ 4 ]
      ------------------------------------       sorted = 2
        0       1       2       3      4

arr = [ 2 ] , [ 5 ] , [ 7 ] , [ 8 ], [ 4 ]
      ------------------------------------       sorted = 3
        0       1       2       3      4

arr = [ 2 ] , [ 4 ] , [ 5 ] , [ 7 ], [ 8 ]
      ------------------------------------       sorted = 4
        0       1       2       3      4
```

- You'll need to collect time data for a number of data sets of different sizes in order to determine the performance of your algorithm. A spreadsheet program may be useful in organizing and graphing this information.
- To empirically determine the performance of the algorithm, you can use the spreadsheet's regression tool, possibly, although an *n log n* performance isn't easily determine. If you suspect that your algorithm's performance is *n log n*, you might want to consider graphing *time* versus *n log n*. What shape would you expect that curve to have if the algorithm is *n log n*?
- You'll be submitting a formal written report for this assignment. Consider using one of the following media for writing up your results:
  - Microsoft Word / Microsoft Excel
  - LibreOffice Writer / LibreOffice Calc (an open source alternative to Microsoft's applications. This document itself was written using LibreOffice.)

- GoogleDocs
- Markdown (a system for including formatting markup in text documents. You'll still need to use a spreadsheet/graphing application to prepare your graphs.)
- HTML (You'll still need to use a spreadsheet/graphing application to prepare your graphs.)
- LaTeX (This is a markup language used primarily by math and physics people to present research. This option is beyond what most people will want to use for this assignment.)

  Regardless of which medium you use for writing up your results, your final submission will be a PDF version of your report.

- As an example of your source code presentation, take a look at the Python code at the end of this document. Note:
  - The font is monospace
  - The comments are clearly visible to one side.
  - Comments may be restructured into multiline comments to preserve the flow of the program.
  - The program is presented as dark text on the white background of the paper, making it clearly readable. (Do not paste colored text on a dark background from VS Code.)

GETTING STARTED

1. Using an appropriate reference, identify the algorithm that you've been assigned to implement. Read and study the algorithm itself, written explanations of how the algorithm works, and possibly graphical presentations of how the algorithm operates.

2. Become familiar enough with the algorithm that you can take a list of values on paper and sort them using the given strategy. A good gauge of your understanding is whether or not you can sit down with someone else and explain the algorithm to them, and show them how it works on paper.

3. The implementation of your sorting algorithm probably won't take much code. Consider including your entire codebase into a single file. For example, a BubbleSort program might include:
   a. a `generate_random_numbers` function to fill an array
   b. a `bubblesort` function that sorts the array
   c. a `display` function that can be used to display the array at any point in time (useful for debugging)
   d. a `main` function that calls the functions and times the `bubblesort` function

EXTENSIONS

1. Create an animated visual (graphical) display of your sorting algorithm. Use Python's `turtle` module or Processing's Python module to implement your animation. (See https://www.youtube.com/watch?v=kPRA0W1kECg for examples of what this can look like.)

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

1. Why, exactly, is it useful for us to study sorting algorithms when Python already has a number of very good implementations of sorts available?

2. The instructor of this course wrote a similar assignment to this one during my Computer Science education. Sadly, copies of that assignment are no longer available. (Look up the word processor *Wordstar* for further info.) What are the advantages and disadvantages of writing this report in Microsoft Office? Google Docs? Plaintext and/or HTML?

3. Is a PDF file a stable format for preserving written documents? Why or why not?

SAMPLE ALGORITHM ANALYSIS

An algorithm's execution time T(*n*) typically changes as a function of the size of the problem *n*. A good way of approximating this time analytically is by counting the number of *assignment operations* that take place during the execution of the algorithm. As well, for our sorting analysis, we'll also be counting the number of *comparisons* that are performed.

Examples of performing a T(*n*) analysis are included in our textbook.

We can get a convenient summary of a T(*n*) function by considering its dominant term, the value that increases faster than everything else in that function. This term controls by an "order of magnitude" the performance of the algorithm, and is written in "Big-O notation" as O(*n*) for the function.

Some algorithms have an execution time that changes only as a function of the size of the problem. Other algorithms have an execution time that changes depending on the actual values that are being manipulated. (Some sorting algorithms, for example, are especially efficient at sorting values that are already mostly in order.) In the case of these algorithms, the Big-O notation will be annotated with "best case," "worst case," and "average case" to indicate performance under those conditions.

SAMPLE SOURCE CODE

```
#!/usr/bin/env python3

"""
selection_sort.py
This program uses a Value class to associate a value in an array and a
Turtle object that will represent that value graphically.

This version works relatively quickly, once the Turtle objects have all been
created. There are some issues with the dimensions of the screen and the bars--
it's all a bit hacked together, there--but the basic functioning of the program
is fine, and 1000 values can be displayed and sorted within a minute or so.

@author Richard White
@version 2017-03-24

"""

import random
from turtle import *


##############################

class Value():
    """Defines a Value in terms of its magnitude, and maintains
    a turtle object for displaying a graphical form of that value.
    """
    def __init__(self, value):
        self.value = value
        self.t = Turtle()
        self.t.speed(0)                  # 0 = fastest speed possible
        self.t.penup()                   # don't draw turtle path
        self.t.color("black")            # turtle will be black on white bg
        self.t.hideturtle()              # don't show turtle (yet)
        self.t.shape("square")           # actually a rectangle

    def set_turtle_size(self, height, width):
        """Turtle's size is based on this value, with dimensions sent in
        as parameters from the main program.
        """
        self.t.turtlesize(((self.value + 1) * height), width, 0)

    def get_value(self):
        return self.value

    def the_turtle(self):
        return self.t

    def set_color(self, newColor):
        self.t.color(newColor)

    def __str__(self):
        return "Turtle[value=" + str(self.value) + ",the_turtle=" + str(self.t)
```

```
############################

def initialize(n):
    """Sets up the screen, and creates an array of n Value objects
    in a list arr, where each value has a turtle associated with it.
    """
    WINDOW_WIDTH = 1200
    WINDOW_HEIGHT = 800
    w = Screen()
    w.setup(WINDOW_WIDTH, WINDOW_HEIGHT)
    w.screensize(WINDOW_WIDTH, WINDOW_HEIGHT)
    w.setworldcoordinates(0,0,w.window_width(), w.window_height())
    w.bgcolor("white")
    LINE_WIDTH = w.window_width() / (n)     # Identify line width based
                                            # on number of turtles and
                                            # screen width
    arr = []
    for i in range(n):
        rand_num = int(random.random() * WINDOW_HEIGHT + 1)
        new_value = Value(rand_num)
        '''
        TurtleSize parameter calculations include "magic number" 9000
        and 0.1, determined through experimentation with Turtle graphics
        May need to be adjusted for other monitors, window sizes, etc.
        '''
        new_value.setTurtleSize(WINDOW_HEIGHT / 9000, 0.1)
        arr.append(new_value)
    display_turtles(arr, w, LINE_WIDTH)
    return arr, w, LINE_WIDTH

############################

def display_turtles(arr, w, LINE_WIDTH):
    """Goes through the array of turtles, and puts each
    turtle at the appropriate location on the screen,
    then shows it.
    """
    for i in range(len(arr)):
        arr[i].theTurtle().goto(i * LINE_WIDTH, 0)
        arr[i].theTurtle().showturtle()

############################

def generate_random_numbers(length, range_of_values):
    """Generates a list of "length" integers randomly
    selected from the range 0 (inclusive) to
    range_of_values (exclusive) and returns it to
    the caller.
    """
    nums = []
    for i in range(length):
        nums.append(random.randrange(range_of_values))
    return nums
```

```
############################

def selection_sort(nums, w, LINE_WIDTH):
    """Takes the list "nums" and sorts it using the
    Selection Sort algorithm. Also moves those turtles to their
    new locations in the window.
    """
    for i in range(len(nums)):
        smallest_loc = i
        for j in range(i, len(nums)):
            if nums[j].get_value() < nums[smallest_loc].get_value():
                smallest_loc = j
        nums[i].setColor("red")             # color the turtles we're swapping
        nums[smallest_loc].setColor("green")
        nums[smallest_loc], nums[i] = nums[i], nums[smallest_loc]         # swap
        # Now move the turtles
        nums[smallest_loc].theTurtle().goto(smallest_loc * LINE_WIDTH, 0)
        if i != smallest_loc:
            nums[smallest_loc].setColor("black")
        nums[i].theTurtle().goto(i * LINE_WIDTH, 0)

############################

def main():
    print("Graphical presentation of SelectionSort using turtle module")
    input("Press [Enter] to continue...")

    NUM_OF_VALUES = 300
    print("Please be patient while",NUM_OF_VALUES,"turtles are created...")
    arr, w, LINE_WIDTH = initialize(NUM_OF_VALUES)

    selection_sort(arr, w, LINE_WIDTH)      # perform sort
    w.exitonclick()                         # leave image on screen


if __name__ == "__main__":
    main()
```