FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Code Review Prac 3: Sorting and Complexity
Week 5, Semester 2 2019

Objectives of this prac

- To gain understanding of time complexity of simple (often sorting) algorithms.
- To learn how to compute the running time for a program.
- To practice using assertions and exceptions for testing and input validation.

Important

Remember to submit your code ***and the .pdf files with the requested graphs and explanations*** **via Moodle and to hand in the code review before you leave the lab**. Otherwise you will get zero marks for the code review.
Also remember to **include your documentation as a** ***docstring*** **at the start of your function**. See `https://lms.monash.edu/mod/page/view.php?id=6227193` for an example of good documentation.

Exercise 1

For this exercise you need to create a file called `mean_items.py` where you:

(a) Write a Python function `mean_items(a_list)` that returns the mean (average) of all the items of `a_list`. If `a_list` is empty, `mean_items` should throw an exception. (*You may assume that the items of the list are real numbers.*)
(b) Write a `test_mean_items()` function that tests `mean_items(a_list)`.
(c) Determine the best and worst case time complexity for `mean_items(a_list)` and include this information in the function's documentation (as usual, via docstrings).

Exercise 2

If you include the module:

```
import timeit
```

which is part of the standard library, you can use the call `timeit.default_timer()` to compute the elapsed time as follows:

```
    start = timeit.default_timer()
    # do whatever you are doing that you need to time
    taken = (timeit.default_timer() - start)
```

If you include the module:

```
import random
```

which is also part of the standard library, you can use it to initialise a pseudo-random number generator and then generate a random real number between 0.0 and 1.0 as follows:

```
    random.seed()
    random_number = random.random()
```

Note that you only need to initialise (or seed) the random generator once. Also, you can provide a number as seed (by passing it as an argument to `random.seed()`) to ensure the sequence of random numbers generated from then on is the same in every run.

For this exercise you need to:

1. Create a file called `test_mean_items.py` with a Python function `table_time_mean_items()` that does the following: for each `n = 2, 4, 8, 16` and so on up to at least `4,096`, prints a line to file `output_ex2.csv` (where csv indicates a tabular format file, where each row of values is simply separated by commas) consisting of three things: `n`, the time taken in creating a list of random numbers of length `n`, and the time taken by `mean_items` to compute the mean of this list, separated by commas. To construct the sequence of power-of-two numbers, do not use multiply. Instead, learn how to use the << operator, which is similar to `sll` in MIPS. **Note:** For this code review prac you can use Python's built-in CSV-handling library (see `https://docs.python.org/3/library/csv.html`) if you want (not needed though!)
2. Execute `table_time_mean_items()` and use the information in the newly created file `output_ex2.csv` to make a graph. Create an `explanation_ex2.pdf` file that contains the graph and an explanation of its shape and include it in your submission. Is it what you expected? **NOTE**: you can create the graph by opening the file in Excel and using its graphing tools. Alternatively, and **highly recommended**: you can give *matplotlib* (`http://matplotlib.org/users/pyplot_tutorial.html`) a chance. If you have installed the Python tools following the video on Moodle, *matplotlib* should be part of your installed Python libraries. A short example which reads a CSV (comma-separated value) file and plots the result is available in Moodle.

Exercise 3

This is the code review task and the code here should be appropriately commented

In Tute 5 you discussed *shaker sort* – a variant of bubble sort which, instead of repeatedly walking left-to-right swapping elements, alternates between left-to-right and right-to-left passes.

The version of bubble sort we saw in class stops as soon as the list is sorted. This can be improved by using the position of the last swap to move the mark further to the left (rather by just one position to the left every time). Shaker sort can be improved in the same way, keeping marks for the start and end of the unordered region, and moving one mark on each pass.

For this exercise you need to create a file called `shaker_sort.py` where you:

1. Write a Python function `shaker_sort(a_list)` that implements *shaker sort*, using the improvements described above. Make sure you include the best and worst case time complexity in the documentation for this new function.
2. Write a Python function `table_time_shaker_sort()` that does the following: for each `n = 2, 4, 8, 16,` and so on up to at least `1,024`, prints a line to file `output_ex3.csv` consisting of three things: `n`, the time taken in creating a list of random numbers of length `n`, and the time taken in sorting the list using `shaker_sort)`.
3. Execute `table_time_shaker_sort()` and graph the data in the newly created file `output_ex3.csv`. Create an `explanation_ex3.pdf` file that contains the graph and an explanation of its shape and include it in your submission. Is the shape what you expected? Discuss this in the file.
4. Write a Python function `table_avg_time_shaker_sort()` that does the following: for each `n = 2, 4, 8, 16,` and so on up to at least `1,024`, prints a newline of file `output_ex3_avg.csv` three things: `n`, the time taken in creating 100 lists of random numbers of length `n`, and the average time taken in sorting these 100 lists using `shaker_sort)`.
5. Execute `table_avg_time_shaker_sort()` and graph the data in the newly created file `output_ex3_avg.csv`. Add the graph and an explanation of its shape to your `explanation_ex3.pdf` file. Is the shape what you expected? Discuss this in the file.
6. Extend the function `table_time_shaker_sort()` to also report the time taken to perform shaker sort on a sorted list `sorted_list` of elements 1 to n. (You do not need to report the time taken to build the list.)
7. Further extend the function `table_time_shaker_sort()`, this time to report the time taken to perform shaker sort on a *reversed* sorted list `reversed_list` of elements from `n` to `1`.
8. `table_time_shaker_sort()` should now be writing a file containing 5 columns, separated by commas. Again graph the data in the updated file `output_ex3.csv`, add the graph and an explanation of its shape to your `explanation_ex3.pdf` file. Is it what you expected? Discuss this in the file.

Advanced Question 1

For those who have been coding for a while and like a bit of a challenge, here are some advanced questions. These questions do not need to be completed to get marks for the code review.

If you are having fun and would like to continue, write a new file `max_sum_interval.py` with the function:

```
find_max_sum_interval(a_list, n)
```

to find the Max Sum Interval of the a_list: you want to find indices `i_min` and `i_max` such that

```
a_list[i_min]+a_list[i_min+1]+a_list[i_min+2]+...+a_list[i_max-1]+a_list[i_max]
```

is as large as possible.

The function just has to loop over all possible values of `i_min` and `i_max` (with `i_min` $\leq$ `i_max` and `i_max` $\leq N$), working out the sum for each, and keeping track of the biggest you've seen so far. (Note that the `a_list` elements are allowed to be negative, so it's not always best to just take the whole `a_list`.) When you have found `i_min` and `i_max`, you should return an `a_list` containing these two indices.

Once again, first estimate your time complexity, then time `find_max_sum_interval` as you did in previous questions and graph your results. Feel free to use a list shorter than before, but make sure it is long enough for the results to make sense. Explain your results. **Important:** Don't forget to write your explanation down and submit it them together with your graphs.

Advanced Question 2

If you are still having fun and would like to continue a bit longer, have a go as the *subset sum problem*, a famous NP-complete problem that has been used in cryptography. The input is the `a_list` again[1], plus another integer called the **target**. You have to determine whether there is a subset (which might not be contiguous) of `a_list` elements that add up to **exactly** the target. The function should return true if such a subset exists, otherwise it should return false.

Your task is to program and time the naive approach in which you just go through each subset in turn, find its sum, and test if it equals the target. The only complicated part of the algorithm is cycling through the subsets. To help you do this, we have provided the following code. Note that you will have to modify it to find the sum of each subset and test whether it equals the target.

```python

def subset_sum(a_list, target):
    """
    The subset a_list marks the elements that are in the current subset.
    If subset[i] = true, then a_list[i] is in the subset, otherwise it isn't.
    There is an extra bit that is used as a sentinel to control the outer loop.
    The subset is initialised to the empty subset: subset[i] == False for all indices
    """
    sentinel = len(a_list)
    subset = [False] * (sentinel + 1)
    sum = 0;

    while not subset[sentinel]:
        print(subset)

        # YOUR CODE GOES HERE

        # This part of the code changes which subset we will look at next.
        # It is equivalent to adding 1 to a binary number.
        i = 0
        while i < len(a_list) and subset[i]:
            subset[i] = False
            i += 1
        subset[i] = True

    return False
```

[1] As you may remember from your maths classes, a set is not ordered and may not contain duplicates. An `a_list` is ordered and may contain duplicates so it isn't the perfect data structure to use here, but we want you to use it rather than use Python sets, which use concepts we have not yet covered in lectures. You will learn more about data structures for representing sets later in the unit.

Once you've written and tested the function, time it as you did in Questions 1 to 3, and graph your results. What is its worst-case complexity, in big-O notation? What about its best-case complexity?