COMP 2150 - Summer 2022

Homework 1: Recursion, Searching, Sorting

Total Points: 50

Due: Mon., June 20, by 2359 CDT
Please carefully read the submission instructions at the end of the assignment. Remember that whatever you submit must be *your own work*.

Grader: Amy Zhang, `mzhang6@memphis.edu`. Grades will generally be posted within 1-2 weeks of the assignment due date. Questions about grading? Please contact her first.

1. Within a Python file named `recursion.py`, write the following functions:
   (a) (4 points) `power(b, n)`
   Returns the value of b^n for any numeric b and any integer n (including negative integers). Do not use any loops, and do not use any built-in ways of computing powers. This function must be recursive.
   (b) (4 points) `count_positives(stuff)`
   Returns how many positive integers (i.e., > 0) are contained in `stuff`. You may assume that `stuff` is a list of integers. Do not use any loops, and do not use any built-in ways of searching a list. This function must be recursive.
   (c) (4 points) `all_even(stuff)`
   Returns whether or not `stuff` contains only even integers. If `stuff` is empty, this should return `True`. You may assume that `stuff` is a list of integers. Do not use any loops, and do not use any built-in ways of searching a list. This function must be recursive.
   (d) (6 points) `binary_search(stuff, target)`
   Returns the index of `target` within the list `stuff` if it exists, or -1 if it doesn't exist. You may assume that `stuff` is a sorted list of integers, and `target` is an integer. Do not use any loops, and do not use any built-in ways of searching a list. This must be done recursively; you can write a recursive helper function if needed.
   (e) (6 points) After your function definitions in `recursion.py`, write some tests for your functions. Be sure to test at least the following scenarios:
   - `power` - Negative, zero, and positive exponents
   - `count_positives` - List contains all positive integers, list contains all non-positive integers, list contains a mix of positive/non-positive integers, list is empty

- `all_even` - List contains all even integers, list contains all odd integers, list contains a mix of even/odd integers, list is empty
- `binary_search` - Target is in the list, target is too low for the list, target is too high for the list, target is within the range of the list but not present, list is empty

2. As discussed in class, the **quicksort** algorithm starts by **partitioning** the list: a pivot value is selected, and the list is partitioned around that pivot. For this problem we'll assume that

   - The pivot is always the element at index 0 of the original list (i.e., before any rearranging of elements is performed).
   - After partitioning, all the elements to the left of the pivot are $\leq$ the pivot, and all the elements to the right of the pivot are $>$ the pivot.

   In the following parts, assume that `stuff` is a list of integers.

   (a) (7 points) Consider this naive algorithm for partitioning a list `stuff`:

   1. Create two new `low` and `high` lists. These will be used to store the elements of `stuff` that are $\leq$ or $>$ the pivot, respectively.
   2. Loop through all elements of `stuff` besides the pivot. If the element is $\leq$ the pivot, copy it into `low`. If the element is $>$ the pivot, copy it into `high`.
   3. Copy the elements of `low` back into `stuff`, then the pivot back into `stuff`, and finally the elements of `high` back into `stuff`.

   Within a Python file named `partitioning.py`, write a function `partition_naive(stuff)` that implements the above algorithm. The function should return the final index of the pivot, after partitioning is complete. Note that `partition_naive` doesn't need to return the partitioned list because the actions that it performs will affect the original list argument. Make sure that your function works for a list of any length ≥ 1.

   (b) (7 points) The partitioning algorithm from the previous part is not very efficient. Creating the two lists `low` and `high` requires extra time as well as memory. A better way is to perform the partition **in-place**, which means that no new lists are created. Instead, we just modify the elements of the original list directly.

   Suppose we have a list `a` containing the elements `[10, 5, 16, 14, 2, 10, 13]`. As before, we want to use the first element (10) as the pivot. Here's an algorithm to perform an in-place partition:

   1. Create two indices, `L` and `U`. Start `L` from the beginning of the list; start `U` from the end.
   2. Move `L` forward through the list until we find the first element that's $>$ the pivot (or until we run out of elements). Move `U` backward through the list until we find the first element that's $\leq$ the pivot (or until we run out of elements).

```
list:   10   5  16  14   2  10  13
index:   0   1   2   3   4   5   6
                (L)         (U)
```

3. If the list elements at indices `L` and `U` are out of order (i.e., if `L < U`), swap the elements at `L` and `U`. Note that the indices `L` and `U` themselves remain unchanged.

```
list:   10   5  10  14   2  16  13
index:   0   1   2   3   4   5   6
                (L)         (U)
```

4. Repeat steps 2-3 until `L` becomes $\geq$ `U`.
   In this example, the second iteration of step 2 brings us to this state:

```
list:   10   5  10  14   2  16  13
index:   0   1   2   3   4   5   6
                    (L) (U)
```

   Since `L < U` is still true, step 3 tells us to swap:

```
list:   10   5  10   2  14  16  13
index:   0   1   2   3   4   5   6
                    (L) (U)
```

   The third iteration of step 2 then brings us to:

```
list:   10   5  10   2  14  16  13
index:   0   1   2   3   4   5   6
                    (U) (L)
```

   Since `L < U` is no longer true, we don't perform the swap in step 3. And with that, the loop ends.

5. The final step is to swap the pivot (which, remember, is still sitting back at index 0) with the element at index `U`:

```
list:    2   5  10  10  14  16  13
index:   0   1   2   3   4   5   6
                    (U) (L)
```

   And voila! The list is now partitioned. The pivot is at index 3 (i.e., the final value of `U`), all elements to the left are $\leq$ the pivot, and all elements to the right are $>$ the pivot.

Within your `partitioning.py` file, write a function `partition_in_place(stuff)` that implements the in-place partitioning algorithm on the list `stuff`. As before, the function should return the final index of the pivot, after partitioning is complete. `partition_in_place` doesn't need to return the partitioned list because the actions that it performs will affect the original list argument. Make sure that your function works for a list of any length ≥ 1.

(c) (4 points) Within your `partitioning.py` file, write a function `verify_partition(stuff, pivot_index)` that returns whether or not `stuff` is validly partitioned around the specified `pivot_index`. In other words, the function should verify that all elements before `pivot_index` are $\leq$ the pivot, and all elements after `pivot_index` are $>$ the pivot. You may assume that `pivot_index` is a valid index of `stuff`.

(d) (4 points) Within your `partitioning.py` file, write a function `random_list(size)` that returns a list containing the integers 1, 2, 3, ..., `size`, randomly shuffled. Each integer should appear exactly once in the list, which means the list should contain no duplicate elements.

(e) (4 points) Finally, below your function definitions in `partitioning.py`, write a program that does the following. Call your previously written functions as needed.

- Create two identical large lists. ("Large" is somewhat subjective – make it large enough to see a noticeable difference in your partitioning algorithms, but not so large that you have to wait for a while every time you test your code!)
- Run the naive partitioning algorithm on the first list. Measure and print how many seconds are needed to complete this. Verify that the list is correctly partitioned.
- Run the in-place partitioning algorithm on the second list. Measure and print how many seconds are needed to complete this. Verify that the list is correctly partitioned.

Python tip on timing: One way to get the execution time of a segment of code is to use Python's built-in `process_time()` function, located in the `time` module. This function returns the current time in seconds and can be used as a "stopwatch":

```
import time
start_time = time.process_time()
# Code to time here
end_time = time.process_time()
# Elapsed time in seconds is (end_time - start_time)
```

Code Guidelines

Points can be deducted for not following these guidelines!

- Most importantly, your code *must* run. Code that does not run may receive zero credit, at the TA's discretion.
- Follow Python capitalization conventions for `variable_and_function_names`.
- Use consistent indentation throughout your code. (Python kind of forces you to do this!)
- Include a reasonable amount of comments in your code. "Reasonable" is somewhat subjective, but at the very least include:
    1. A comment at the top of each program summarizing what it does.
    2. Comments that indicate the major steps taken by the program. There are generally at least two or three of these, such as collecting user input or making calculations.

Need Help?

- Email your lecture instructor.
- Use the CS Discord server — however, this is *not* for other people to write code for you.
- The UofM offers free online tutoring through the Educational Support Program (ESP):
  `https://www.memphis.edu/esp/onlinetutoring.php`
  Be sure to schedule sessions well in advance!

Submission Instructions

- Create a zip file containing all your Python source files. You can use any Python development environment you like, as long as you submit the `.py` source files.
- Submit your zip file to the appropriate assignment in your Canvas *lab* section. The assignment is technically due at 2359 on the due date; however, Canvas will continue to accept submissions without penalty up until 0800 the following morning. After that 8-hour "grace period," no submissions are accepted. You may submit as many times as you want up to the deadline.