CMSC 201 Section 40
Spring 2020
Lab 11

Analyzing Sort Routines

Due: Thursday, May 7, 11:59:59 pm

Value: 10 Points

Background

We introduced three sorting algorithms in lecture on April 29. On May 4, we talked about how to analyze their performance - how long it takes each algorithm to sort a given list of numbers. The purpose of this lab is to let you do some more instrumentation of your own, to see the performance of the different routines.

You will start with a base program written by Prof. Arsenault. You will modify this code as instructed to instrument the algorithm performance.

Step 1: Get the base code.

Create your lab11 directory. Copy the program sorting.py from Prof. Arsenault's public directory on gl into that directory.

cp /afs/umbc.edu/users/a/r/arsenaul/pub/sorting.py lab11.py

This will have the effect of changing the name of the program to lab11.py in your files.

Use the cat command to look at the code. You will see that it implements iterative versions of the bubble sort and selection sort routines, and a recursive version of the quicksort routine. The main program generates a list of random integers, and then calls each function to sort the list. The main program "instruments" the sorting by recording the system clock time immediately before the function is called, and recording the system clock time immediately after the function

returns. These times are always reported in seconds, as a floating point number. This gives you a reasonable approximation of how long it took to sort the list.
Run the code to make sure it works, and look at the time reported for each function.

Step 2: Don't just count the time; count the operations

We want to change each of the three functions to count the number of operations that it took to sort the array. So you will have to change the code as follows:

Bubble sort and Selection sort:

We want to count the number of comparisons made between list elements, and the number of times values are swapped.
At the beginning of the function, initialize two new variables:

```
comparisons = 0
```

and

```
swaps = 0.
```

Then edit the code. Make sure that every time two list elements are compared, you add a

```
comparisons += 1
```

statement; and every time two list elements are swapped, you add a

```
swaps += 1
```

statement.
Immediately prior to the return, print out the number of comparisons and the numbers of swaps made.

Quicksort:

This is different because of the recursive calls. For each invocation of quicksort, there is exactly one comparison of each element and exactly one operation appending each element to its new list. There are never any real "swaps" in this Python implementation. So we'll count the number of comparisons and the number of appends, which should be identical.

Since we only let you return one value from a function call, the recursive function quicksort() will have to be changed to return a list, called result. Modify the code so that result[0] is the list that results from sorting the input; result[1] is the number of comparisons; and result[2] is the number of appends. Then return(result).

In the code itself, for the base case, you will return 0 comparisons and 0 appends. The “comparison” of the length of the list being 0 doesn’t really count as comparing two elements against each other.

For the recursive case, count the number of comparisons and the number of appends in the “for” loop where you’re putting elements in “less”, “equal” or “greater”. Then you’ll need to call the function recursively with “less” and assign that to a variable we’ll call “lower”. Then call the function recursively on “more” and assign that to a variable we’ll call “more”. Then build your result - result[0] is lower[0] (the sorted list) + equal + more[0] - in other words, three lists appended together. result [1] and result[2] are found by adding the number of comparisons and appends you got from this call to what you have in “lower” and “more.”

Then you can return the result.

Print out the number of comparisons and swaps or appends for each of the three sort algorithms for a randomly-generated input.

Step 3: Optimize the Code

The bubble sort routine doesn’t stop even when it could. If there were no variable swaps made during one run of the loop, then the list is already sorted. There’s no need to sort any further. That is, if you start with

5 1 2 3 4

The first run through the list will “bubble” the 5 to the end, leaving you with:

1 2 3 4 5

You can see we’re done. There’s no need to do the comparisons on 1 -2 - 3 - 4 and then 1 - 2 - 3 and then 1 - 2. The current code will do those comparisons, but it doesn’t have to.

Your assignment is to modify the bubble_sort code to keep track of the number of swaps made during each iteration of the outer loop. If you do not make a swap during an iteration of the “for i…” loop, you’re done and you can stop the loop, print out and return the results.

Write a new function, optimized_bubble_sort(), which implements this optimized bubble sort algorithm. Your outer loop in this new function should be a while loop, not a for loop. You execute this loop WHILE you haven’t gone through the entire list AND there was a swap in the last iteration of the outer loop.

Generate a random list, and run both the bubble_sort and optimized_bubble_sort functions against the same list (REMEMBER MUTABILITY). Are the results different?