Computer Science I
MqEuRiGcEk

CSCI 141
Homework 8

9/27/2020

1 Problem

You have so far learned about sorting algorithms with different time complexities, e.g., insertion sort with $O(N^2)$ time, merge sort with $O(N \log N)$, and quick sort with $O(N \log N)$ on average. In this assignment, you will first compare the expected time of those sorting algorithms on a same unsorted list, and then you will write a new sorting function that combines strong features of merge and quick sort schemes.

1.1 Part 1: Merge-Quick Sort

What is your favorite sorting algorithm and why? If it is merge sort, you may like its stable performance; if you like quick sort, its superior behavior in many practical scenarios might be the reason. Is there any sorting algorithm that enjoys the advantages of both?

You will write a new sorting algorithm that intertwines the two sorting algorithms in a way that each recursion executes one layer of `merge_sort` plus one layer of `quick_sort`.

We name this hybrid algorithm `merge_quick_sort`, and perform the following steps in each recursion:

(a) Split the given list into equal halves.
(b) Partition each half into three sublists.
(c) Sort each of the sublists and concatenate them
(d) Merge the two sorted halves.

Note that the list is halved in step (a) and this ensures that the recursion depth is at most $\log_2 N$. Since the algorithm performs $O(N)$ operations at each depth, the total computation takes $O(N \log N)$ time.

A visualization of the entire sorting algorithm is on the next page.

How to ***SORT*** — a recursive algorithm

Algorithm origins

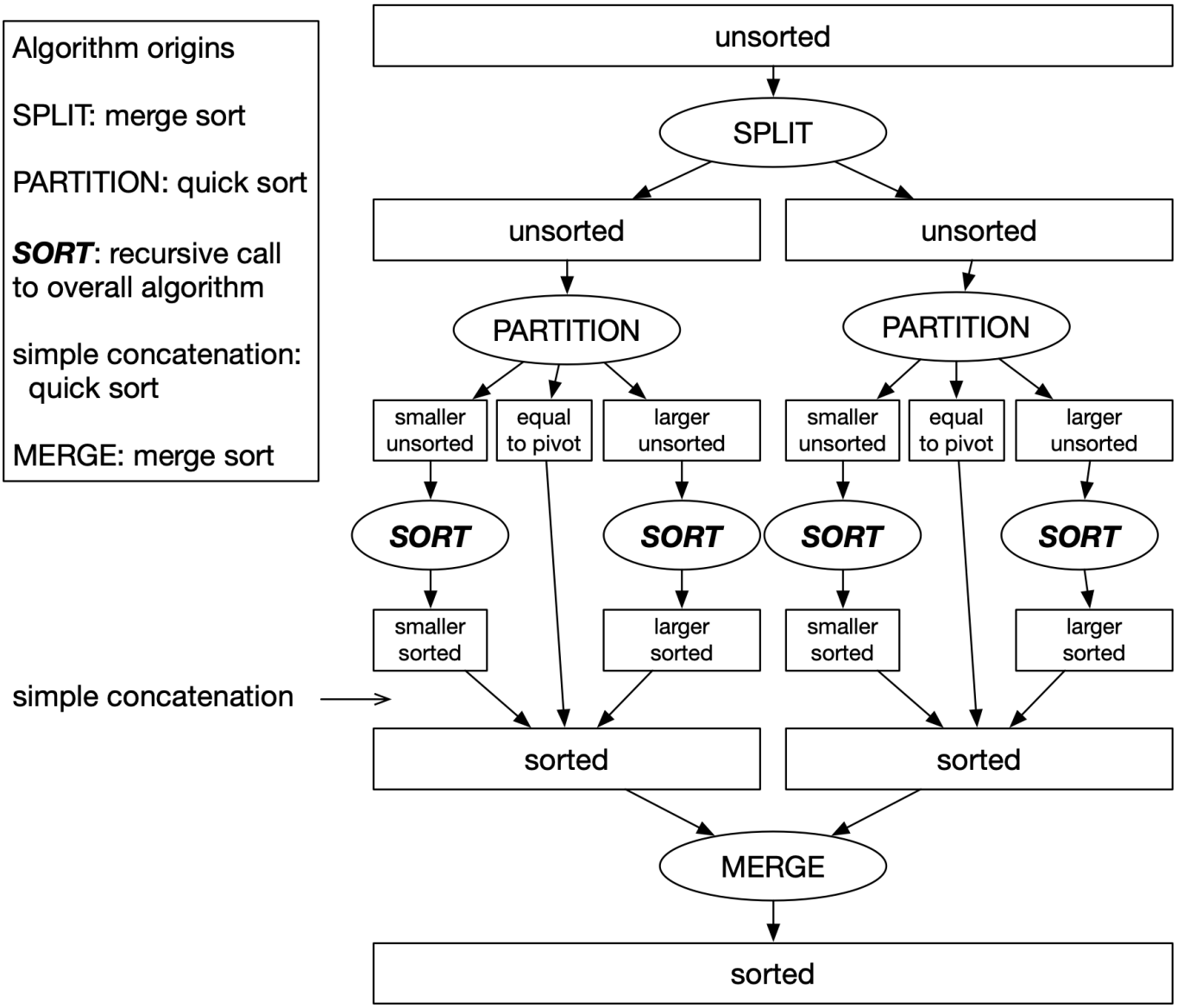
SPLIT: merge sort

PARTITION: quick sort

SORT: recursive call to overall algorithm

simple concatenation: quick sort

MERGE: merge sort



1.2 Getting Started

Download the starter code from:

`https://www.cs.rit.edu/~csci141/Homeworks/08/code.zip`

Unzip the file, navigate into the `Code` directory and either move or copy paste the source files into a new PyCharm project.

You are being provided with 3 files that you **should not modify**:

- `merge_sort.py`: The lecture merge sort code.
- `quick_sort.py`: The lecture quick sort code (pivot is first element).
- `test_merge_quick_sort.py`: A test unit for the main program, `merge_quick_sort.py`, that you will write.

Create your new main program `merge_quick_sort.py`. For this part you only need to implement the `merge_quick_sort` function. This function takes a list of data as a parameter, and returns a new sorted list of data using the algorithm described above.

Your main program will need to import both of the lecture sort modules so you can reuse them in your implementation. Recall that in order to call a function in a separate module, you must do the following (for example):

```
import merge_sort

# call the split() function in merge_sort and get the two halved lists
left, right = merge_sort.split()
```

1.3 Testing Your Sorting Function

In order to verify that your new sorting function works correctly, run the supplied `test_merge_quick_sort.py` module. There are 8 different tests that are run. You should make sure all tests pass:

```
empty list passed
single element list passed
two element list sorted passed
two element list unsorted passed
random 10 element list passed
random 100 element list passed
random 1000 element list passed
random 10000 element list passed
```

If a test fails it will show you the expected result compared to the result from calling your sorting function.

2 Part 2: Comparing Sorts

Now that you have a working `merge_quick_sort` function, it is time to see how it compares to the original sorts.

Create a new Python source file, `compare_sorts.py`, in the same location as the other source files.

You will now implement a `main` function in `compare_sorts.py`. As with all the other modules, it should have an *if guard* that calls the `main` function (no arguments are passed).

This is how the `main` function should work:

1. Prompt the user for a list size, `N`. You can assume this number is between `0` and `10000` without the need to check.

2. Create a new list of integers, `lst`, of size `N`, with elements in order between `0` and `N-1`.

3. For larger sorted lists, `quick_sort` will exceed the call stack limit. To increase the default limit do:

   ```
   if N + 10 > sys.getrecursionlimit():
       sys.setrecursionlimit(N + 10)
   ```

   Note that in order to do this you must `import sys` beforehand.

4. Perform timed sorts for `quick_sort`, `merge_sort` and `merge_quick_sort` using the sorted data. In order to do the timing you will need to `import time` and get a *timestamp* directly before and after you call each sort. The timing is in seconds and should be converted to milliseconds when displayed. For example:

   ```
   start = time.perf_counter()
   quick_sort.quick_sort(lst)
   elapsed = time.perf_counter() - start
   print('quick_sort (sorted) elapsed time:', elapsed * 1000, 'msec')
   ```

   Note that we are essentially throwing away the sorted data returned by the sorting functions by not assigning the function call results to variables. It is safe to ignore that data at this point, since we are assuming all of the sort functions work.

5. Take your list of sorted data and randomly shuffle the elements. You will need to `import random` and do:

   ```
   random.shuffle(lst)
   ```

   This random shuffling of the list is *in-place* and will sort the passed in `lst`.

6. Again perform the timed sorts for the three sorting functions using the random list of data.

2.0.1 Sample Run

The timings here are *machine dependent*. The only thing you should observe is that `quick_sort` on sorted data runs much slower than the other sorts when testing with `10,000` elements. That is because it is $O(N^2)$ vs $O(NlogN)$ for all the others.

```
list size: 10000
quick_sort (sorted) elapsed time: 5120.129966 msec
merge_sort (sorted) elapsed time: 50.30398099999989 msec
merge_quick_sort (sorted) elapsed time : 53.55109900000077 msec
quick_sort (random) elapsed time: 19.426653000000016 msec
merge_sort (random) elapsed time: 48.093652000000375 msec
merge_quick_sort (random) elapsed time: 37.48606100000007 msec
```

You can also see this effect on `merge_quick_sort`. With sorted data it runs slower than with random data because of the poor pivot selection when partitioning with `quick_sort`.

3 Grading

- 70%: Correct implementation of `merge_quick_sort`.
- 30%: Correct implementation of `compare_sorts.py`.
- Deduction of up to 5%: Each function has a *docstring* containing a sentence describing its purpose. This documentation helps others understand how they may reuse the function. An example is provided on the Course Resources webpage:
  `http://www.cs.rit.edu/~csci141/Docs/style-example-py.txt`
- Deduction of up to 5%: The program is in the correct, standard style, starting with a *docstring* for the whole file. The program file docstring must contain your *full* name.
- Deduction of up to 5%: The program file must be contained in a **zip** file that you upload to the proper drop box.

4 Submission

Zip your `merge_quick_sort.py` and `compare_sorts.py` source files into a zip file name `hw08.zip` and upload it to the MyCourses Assignment dropboox by the due date.