Assignment 5 - Radix Sort vs. The World

Robin Dawes

October 31, 2021

The radix sort algorithm sorts a set without ever comparing any of the values. How is this possible?

.

The Radix Sort Algorithm

Let S be a list of n positive integers. Let $B\_0, B\_1, \ldots, B\_9$ be lists, all of which are initially empty.

Traditionally, the B lists are called "buckets", and radix sort is often called bucket sort.

```
for each digit position i, starting with the 1's position:
    for each element x of the set:
        let d be the digit in position i of x
        append x to B_d
    rebuild the list S by concatenating all of the B lists in order
                                                (B_0, then B_1, etc)
    set the B lists back to empty lists.
```

Example of the Algorithm in Operation

Sort the list [843, 952, 17, 199, 59, 33, 13, 904, 1006]

Round 1: Use the "ones" digit to assign values to buckets:

B List	Contents
0	
1	
2	952
3	843, 33, 13
4	904
5	
6	1006
7	17
8	
9	199, 59

S = [952, 843, 33, 13, 904, 1006, 17, 199, 59]

Round 2: Use the "tens" digit to assign values to buckets:

B List	Contents
0	904, 1006
1	13, 17
2	
3	33
4	843
5	952, 59
6	
7	
8	
9	199

S = [904, 1006, 13, 17, 33, 843, 952, 59, 199]

Round 3: Use the "hundreds" digit to assign values to buckets:

Values that do not have a "hundreds" digit go in the 0 bucket.

B List	Contents
0	1006, 13, 17, 33, 59
1	199
2	
3	
4	
5	
6	
7	
8	843
9	904, 952

S = [1006, 13, 17, 33, 59, 199, 843, 904, 952]

Round 4: Use the "thousands" digit to assign values to buckets:

B List	Contents
0	13, 17, 33, 59, 199, 843, 904, 952
1	1006
2	
3	
4	
5	
6	
7	
8	
9	

S = [13, 17, 33, 59, 199, 843, 904, 952, 1006]

None of the numbers has a "ten-thousands" digit so we are finished.

And the list is sorted!

The Assignment

Part 1: Implement Radix Sort

You can find thousands of implementations on the Internet, but I expect you to write your own. At this stage of the course, the description given above should give you all the information you need to develop the algorithm in Python.

Part 2: Implement Merge Sort or Quicksort

I have posted my implementations of Merge Sort and Quicksort - you are welcome to use those with no penalty. But for your own benefit I recommend writing your own without referring to any sources. You may find that Quicksort is trickier to write than Merge Sort - it's easy to go wrong as the "left" and "right" markers come together during the partition phase.

Part 3: Run Both Algorithms on Randomly Generated Lists, and Compare

STEP 1: Generate 100 lists of 100 randomly chosen integers. Apply both of your sorting agorithms (radix sort and whichever other one you chose) to each list and test to make sure the results are the same.

This is a good check - if all the results match, it's very probable that the two algorithms are both correct because it's extremely unlikely they would both go wrong in exactly the same way. Of course if the results don't match, one or both of the algorithms must have an error.

STEP 2: Repeat the following:

- Generate a random list of 10,000 integers from the range [100,000 ... 999,999]
- Make a copy of this list
- Use the **`time`** module to time your radix sort algorithm on one copy of this list
- Use the **`time`** module to time your other sort algorithm on the other copy of this list

See below for a brief demonstration on how to do this.

How many times should you repeat this? Repeat it until you feel confident in saying that for sets of this size and nature, the average time for one of the sorting algorithms is less than the average time for the other. Your conclusion might be something like "In 100 trial runs, Algorithm [redacted] was faster than Algorithm [redacted] 87 times. The average time for Algorithm [redacted] was [redacted] seconds and the

average time for Algorithm [redacted] was [redacted] seconds. These empirical data suggest that for sets of this size and nature, the average time for Algorithm [redacted] is less than the average time for Algorithm [redacted]."

Or it might look like "In 300 trial runs, Algorithm [redacted] was faster than Algorithm [redacted] 48% of the time, while Algorithm [redacted] was faster 46% of the time. In the other 6% of the trials, the running times differed by less than 0.0001 seconds and we judge that to be a tie. The average times for the two algorithms are [redacted] seconds and [redacted] seconds, which are within 2% of each other. It is not possible to say with confidence which has the faster average time."

Advanced statistical analysis is not required! And if the results are close, you can choose the threshold for saying they are too close to decide which is faster.

How You Will Be Graded

The assignment will be marked out of 100. 90% of the grade will be for correctness and 10% of the grade will be for programming style.

The grader will read your code and will run your program to test correctness.

What to Submit

For this assignment, you are required to upload to onQ:

- A Python program containing
    (a) your implementation of Radix Sort
    (b) your implementation of your other selected sorting algorithm
    (c) your test procedure that validates your implementations by comparing results
    (d) your data-gathering procedure that collects the required timing data
- A text file or pdf containing your conclusion regarding the average time required for the two algorithms
- You are NOT required to upload the html page generated by pydoc, because I know some students in the class have not been able to get this to work. Be prepared though, I believe that CISC-124 requires the javadoc (similar to pydoc but

for Java - but I guess that's obvious!) documentation to be handed in.

Remember to put your name and student number at the top of your program file, as well as the statement regarding academic integrity (as specified in Assignment 1). Also, your program must contain appropriate docstring documentation at the beginning of the program and in each defined function.

Due Date

The due date for this assignment is 20211107 (November 7), 11:59 PM. This is not the due date originally posted, but due to circumstances I was not able to post this assignment on October 29 as planned.

Using the Time Module

The Time module lets you take a clock reading. If you take a clock reading just before calling a function and take another just after calling the function, subtracting the first reading from the second reading gives you the elapsed time in seconds.

EXAMPLE:

```
import time

def fun():
        i = 1
        x = 0
        while i <= 1000000:    # loop a million times doing some stuff
                x += 2
                i += 1

start = time.time()
fun()
end = time.time()
print(end - start)
```