Computer Science AP/X — CSCI-140/242
Merchants of Venice — Lab 3

01/30/2018

1 Quickselect

There is an algorithm called *Quickselect* (often called *k-select*) which is custom suited for the problem of finding the optimal location for the new merchant. The algorithm is used to find the `k`th smallest element in an unordered list. In our problem, `k` is the index of the median element, which we defined as:

```
k = len(lst) // 2
```

The algorithm is similar to Quicksort, but it does not require a complete sort of the data in order to find the k-th element.

1.1 Quickselect example

Consider a list of 9 locations and finding the index of the 2nd smallest element.

```
data = [65, 28, 59, 33, 21, 56, 22, 95, 56]
k = 2
```

First, choose a random pivot from the list. We will determine the "sorted" location of the pivot element and compare it to `k`:

```
pivot = 56
```

The smaller elements of the pivot are in the left subarray, `smaller`:

```
smaller = [28, 33, 21, 22]
```

A count of the elements that are equal to the pivot are in `count`:

```
count = 2
```

The larger elements are in the right subarray, `larger`:

```
larger = [65, 59, 95]
```

We now know the pivot's exact location is `m` if the list was sorted:

```
m = len(smaller) = 4
```

There are three possible outcomes after the partition:

1. If `k` is between `m` and `m+count`, the `k`-th element has been found. It's the `pivot`.
2. If `m` is greater than `k`, the `k`-th smallest element is in `smaller`. We have eliminated `count + len(larger)` elements, but have not found the `k`-th value. Perform a similar partition on `smaller` using the same value for `k`.
3. Otherwise the `k`-th smallest element is in `larger`. We eliminated `len(smaller) + count` elements. Perform a similar partition on `larger` using `k - m - count` for `k`.

In this example we have the second outcome, `k = 2` is less than `m = 4`, so we perform another partition using the smaller list `smaller = [28, 33, 21, 22]` with index `k=2`. The element in the second position will eventually be found, `22`, at index 1, starting at 0 in the list.

1.2 Quickselect complexity

Quickselect can suffer from the same problem as Quicksort (the pivot is chosen poorly and the partitions are extremely uneven). Because of this, the worst case performance is $O(n^2)$. In practice, working with random data and pivots yields an average/best-case performance of $O(n)$. This is why Quickselect is much faster than Quicksort, which recall is $O(n \log n)$.

Quickselect is usually implemented like Quicksort, using an in-place algorithm, which means when the `k`-th element is found, the list of data will be partially sorted. To preserve the original list, it is okay to create new lists during each partitioning. This method will consume more memory to store the temporary lists during the search, and run a little slower than in-place, but it will still maintain linear complexity for performance.

2 Implementation

Write a program, `merchants.py`, that demonstrates the performance differences between using Quickselect and Quicksort to solve the problem of finding the median location for the new merchant.

2.1 Command Line

The program is run on the command line as:

```
$ python3 merchants.py [slow|fast] input-file
```

If the number of command line arguments is incorrect, display a usage message and exit:

```
Usage: python3 merchants.py [slow|fast] input-file
```

If `slow` is specified, use Quicksort to find the merchant at the median location, otherwise assume `fast` is specified and use Quickselect.

Assume the input file exists, and it contains at least one merchant. The in-lab activity downloaded a series of test files that to use.

2.2 Program Output

When the program runs, it will produce 5 lines of output:

- The search type that was specified on the command line:
  `Search type: [slow|fast]`
- The total number of merchants in the input file:
  `Number of merchants: #`
- The time it took to perform the search, in seconds:
  `Elapsed time: # seconds`
- The merchant at the median:
  `Optimal store location: Merchant(name='XXX', location=#)`
- The sum of the distances from the optimal location, to all the other merchant locations:
  `Sum of distances: #`

2.3 Timing

The calculation of the elapsed timing should only include the time it takes to perform the search. It should not include the time to read in the file, or to display the final output.

Use the `time` module in python to determine the elapsed time, in seconds:

```
>>> import time
>>> start = time.perf_counter()
>>> time.perf_counter() - start
0.0016670000000000018
```

2.4 Optimal Location

As discovered in problem-solving, the optimal location is the median element. If the number of elements in the list is odd, the median is the middle element. If the number of elements in the list is even, the median is the average of the two middle elements.

For this problem we define the median as the element at location `len(lst) // 2`, regardless of the length of the list. This works because with an even length list the sum of distances will be the same anywhere between and including the locations of the middle elements.

2.5 Sample Runs

Here are sample runs using the million merchant data set, `test-1M.txt`. Results should be similar depending on the computer's speed.

2.5.1 Quicksort

```
$ python3 merchants.py slow test-1M.txt
Search type: slow
Number of merchants: 1000000
Elapsed time: 14.809489999999998 seconds
Optimal store location: Merchant(name='Merchant_92030', location=4994729)
Sum of distances: 2501634763753
```

2.5.2 Quickselect

```
$ python3 merchants.py fast test-1M.txt
Search type: fast
Number of merchants: 1000000
Elapsed time: 1.5101100000000005 seconds
Optimal store location: Merchant(name='Merchant_92030', location=4994729)
Sum of distances: 2501634763753
```

3 Grading

This assignment will be graded using the following rubric:

- (15%) Problem-solving
- (10%) In-lab Activity
- (10%) Code Design
- (50%) Functionality:
  - (20%) Quicksort solution
  - (30%) Quickselect solution
- (5%) Version Control: A Git log was submitted with a reasonable number of commits.
- (10%) Code Style and Documentation: Proper commenting of modules, classes and methods (e.g. using docstring's).

4 Submission

The try command to submit is:

```
try csapx-grd lab3-1 merchants.py log.txt
```

Recall that to verify the latest submission has been saved successfully, run try with the query option, -q:

```
try -q csapx-grd lab3-1
```