Computational Problem Solving CSCI-603
Battle of Bands Lab 5

1 Introduction

Every year a local central Jersey arts paper, The Newbridge Muse, covers Newbridge High School's Battle of the Bands. Beginning in 2003, the Muse ran an online survey that asked their readers to evaluate the acts. Since then, the paper has ran articles discussing the best, worst, and most mediocre bands according to the survey. Spurred on by the high profile press coverage, the Battle of the Bands attracts thousands of acts each year; a feat made possible due to the town's poorly worded laws that technically classifies anyone over the age of thirteen a Newbridge High School student.

For this lab, you will implement a program called `bands.py` which demonstrates the performance differences between using Quickselect and Quicksort to solve the problem of finding the most mediocre band.

2 QuickSort

For this lab, you will implement the QuickSort algorithm as described by the pseudocode from the Problem Solving section. This algorithm sorts the elements *out-of-place*, which means the algorithm copies the elements into extra auxiliary lists as part of the computation rather that rearrange the elements in the given list.

Notice that *QuickSort* can be implemented to sort *in-place*. In that case, it will not create any other lists or data structures, reducing the amount of space in memory required to do the sorting. For this lab, we are not concerned about memory space, so we will implement the algorithm to sort *out-of-place*.

2.0.1 Choosing the pivot

As discussed during the problem-solving session, always choosing the first element as the pivot can make our algorithm perform poorly if the lists are already in order. Instead of assuming the likelihood of already in order lists, we will implement *QuickSort* to be independent of any particular input.

Choose the pivot from the given list at random using the `random.randint` function. That will ensure that QuickSort will perform well on any input with high probability, although that cannot be guaranteed for every possible input.

3 Quickselect

There is an algorithm called *Quickselect* (often called *k-select*) which is better suited for the problem of finding the most mediocre band. 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.

3.1 Quickselect example

Consider a list of 9 numbers and finding the index of the 2nd smallest element (i.e. index 1 for a sorted Python list).

```
data = [65, 28, 59, 33, 21, 56, 22, 95, 56]
k = 1
```

First, choose a pivot from the list. Note that it is a good practice to pick a *random* pivot but picking the first index is sufficient for this lab. We will determine the "sorted" location of the pivot element and compare it to `k`:

```
pivot = 56
```

The elements smaller than 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 would be `m` if the list was sorted:

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

There are three possible outcomes after the partition:

1. If `k` is between `m` (inclusive) and `m+count` (exclusive), 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` but update `k` to be `k - m - count`.

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

3.2 Quickselect complexity

Quickselect can suffer from the same problem as Quicksort, i.e. there is a chance that the pivot does not split the list into similar sized subsets and the problem is not reduced by $\sim \frac{1}{2}$ at each iteration. Therefore, the worst case performance is $\mathcal{O}(n^2)$. In practice, working with random data and pivots yields an average/best-case performance of $\mathcal{O}(n)$. The proof of this is outside the scope of our course but it is relatively easy to derive with an understanding of geometric series. This is why Quickselect is much faster than Quicksort's $\mathcal{O}(n \log n)$ performance.

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. For this lab, 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; however, it will still maintain linear complexity for performance.

3.3 Main Program

The `bands.py` will demonstrate the performance differences between using Quickselect and Quicksort to solve the problem of finding the most mediocre band.

The program will run with an extra command line parameter before the input filename that will determine which algorithm your program will use.

```
$ python3 bands.py [slow|fast] <input-file>
```

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

```
Usage: python3 bands.py [slow|fast] <input-file>
```

If `slow` is specified, use Quicksort to find the band at the median amount of votes otherwise assume `fast` is specified and use Quickselect.

If the number of command line arguments is correct, it's guaranteed the file exists and its content is valid. The input file will be a tab delimited file listing band names followed by the number of votes that they received. The format is `<band-name>\t<votes>`, see Figure 1 for an example.

```
Mother 13 200
Old Skull 1045
LiLiPUT 954
The Gas Station Dogs  345
Wesley Willis 105
```

Figure 1: An example input file.

3.4 Program Output

When the program runs, it will produce 4 lines of output:

- The search type that was specified on the command line:
  ```
  Search type: [slow|fast]
  ```
- The total number of bands in the input file:
  ```
  Number of bands: #
  ```
- The time it took to perform the search, in seconds:
  ```
  Elapsed time: # seconds
  ```
- The band at the median:
  ```
  Most Mediocre Band (name='XXX', votes=#)
  ```

3.5 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
```

4 Input and Output files

You have been provided with some test files to help you verify the correctness of your implementation. Download those files from here. Here is a brief description of what is provided to you:

1. **data**: A folder containing different files with bands information.

2. **output**: A folder containing the expected output when the program run with the **data** files.

4.1 Sample Runs

Here are sample runs using the million band data set, `test-1M.txt`. Everything should match the examples below except for the time. The elapsed time will obviously depend on the system that is running the program; however, you should see that quickselect runs significantly faster than quicksort.

4.1.1 Quicksort

```
$ python3 bands.py slow test-1M.txt
Search type: slow
Number of bands: 1000000
Elapsed time: 6.737467942002695 seconds
Most Mediocre Band: Band(name='The Circumstantiating Stellaria Medias', votes=49996613)
```

4.1.2 Quickselect

```
$ python3 bands.py fast test-1M.txt
Search type: fast
Number of bands: 1000000
Elapsed time: 0.3108070980015327 seconds
Most Mediocre Band: Band(name='The Circumstantiating Stellaria Medias', votes=49996613)
```

5 Grading

This assignment will be graded using the following rubric:

- 20% : results of problem solving
- 15%: Design
- 55%: Functionality
  - 25%: QuickSort algorithm
  - 30%: QuickSelect algorithm
- 10%: Code Style and Documentation

6 Submission

Create a ZIP file named **lab5.zip** that contains all your source code. Submit the ZIP file to the MyCourses assignment before the due date (if you submit another format, such as 7-Zip, WinRAR, or Tar, you will not receive credit for this lab).