Project 01

44-349: Survey of Algorithms

Grade Breakdown

Deliverable	
Code	10
Report First Draft	5
Report Second Draft	10

For this project, you are to implement four algorithms for selecting the element in a list of numbers such that only k elements are smaller than it. For example, in the list A=[7, 4, 2, 14, 75]:

```
select(A, 0) == 2
select(A, 1) == 4
select(A, 4) == 75
```

You will impelment solutions with two different strategies:

1. Sort the list, then return the element at index k
2. Implement the selection algorithm described below

Selection Algorithm

For a more thorough breakdown of this algorithm, see https://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf (section 2.4). A summary of this algorithm follows.

Algorithm Sketch

Choose a random element in the list, and separate the list into three distict groups:

- Al: The set of numbers strictly smaller than the randomly selected element
- Ae: The set of numbers strictly equal to the randomly selected element
- Ar: The set of numbers strictly larger than the randomly selected element

If k < len(Al), recursively call the selection algorithm on Al. If k <= len(Al) + len(Ae), the randomly selected element must be the k^th smallest element, so return it. Otherwise, the k^th smallest element must be in Ar, so recursively call the selection method on Ar, subtracting len(Al)+len(Ae) from k. In pseudocode:

```
e = random value in A
Al = all values in A less than e
Ae = all values in A equal to e
Ar = all values in A greater than e

if k < len(Al)
    return select(Al, k)
elif k<= len(Al) + len(Ae)
    return e
else
    return select(Ar, k - len(Al) -  len(Ae))
```

Note that this algorithm behaves similarly to quicksort: it partitions the array into groups and recurses into those groups. It will perform better if you can partition the array in place (which may be worth your time to investigate).

In the worst case this algorithm is $\Theta(n^2)$ as we may randomly select the minimum or the maximum value in the list every time, resulting in $\sum_{i=0}^{n-1}i$ comparisons. As you explore this algorithm, use the methods discussed in class to determine the expected runtime of this algorithm; it will be required in your report.

Coding Deliverables

Implementation

You may choose any programming language you wish to implement these algorithms; it is up to you to determine the suitability of the language to easily solve the problem.

You are to compare the execution time of the following algorithms:

- Algorithm 1: Sort ($O(n^2)$) and return A[k]
    - Implment your own quadratic sorting algorithm (Insertion, Selection, Bubble, ...). Sort the array and return the k^th element.
- Algorithm 2: Sort ($O(nlg(n))$) and return A[k]
    - Implment your own linear-logrithmic sorting algorithm (Quick, Merge,...). Sort the array and return the k^th element.
- Algorithm 3: Sort (Built In) and return A[k]
    - Sort the array using the builtin sorting algorithm in your programming language and return the k^th element
- Algorithm 4: Select algorithm
    - As outlined above

You must test your code using unit tests (jUnit, nosetests, googletest), or write your own suite of tests to verify that it works. These tests must run separately from your timing code. You must also write code to time ITERS iterations of your algorithms on randomly filled arrays of length N. For example, if I were to write this code in Python, and my selection fnction's name is select, I could write (in Python 3):

```
from time import perf_counter
N = 100
ITERS = 1000
elapsed = 0
for i in range(ITERS):
    L = random_list(N) # function I wrote to generate a
```

```
random list of N elements
    start = perf_counter()
    # It doesn't matter which element you choose, but be
consistent across your timings
    select(L, N//2)
    end = perf_counter()
    elapsed += (end - start)
print('Select algorithm:', elapsed/N, 's')
```

Remember to not include the creation of the random list in your timing. You should always output the number of iterations and the list size you are using for the test run. Example:

```
-- Iterations: 1000
-- List length: 100
Select algorithm: 0.0006288596254307776 s
Mergesort select: 0.003560612862929702 s
Builtin sort sel: 0.00014974827179685236 s
```

You may choose one of two ways to run your code:

Method 1: Single run for one value of ITERS and N

If you choose this method (which would mirror the sample output above), you will run your program several times to generate the timings to include in your report. If you do this, you will either need to write your program so that you can specify ITERS and N either at the command line or through user input; at no time should or will the grader modify your code to generate different timings.

This method has the advantage of allowing you to decide to include different timings in your report without having to re-run your other timings. However, it may make it more difficult to generate your graphics for your report.

Method 2: Single run for all values of ITERS and N

If you choose this method, you will run your timing code and generate all values for your report graphics. This approach has the benefit of making it easier to format your output for inclusion into whatever graph plotting software you want to use (Excel or GNUplot are two excellent choices), so you have to do less data wrangling to generate your graphics.

Submission

Your code must be submitted using the git repositories for the course. You may choose to use gitsubmit or a fully featured git client. You must include instructions for running both your test code and your timing code. See the course website for the rubric.