Principles of Urban Informatics
Assignment 4

Posted on: 09/29/2014
Due Date: 10/06/2014

Data description

In this assignment we are going to solve a series of problems paying attention to efficiency in terms of execution time. Also pay attention to naming your solutions as specified in the following instructions.

Problem 1

In this problem you are going to solve the problem of finding the distinct elements in a list of integers. For example, given the list $[3, 19, 20, 19, 3, 0, 20]$ distinct elements in it are $[3, 19, 20, 0]$ (notice that the order is not important here). Your task is to create three python functions which implement three different solutions to this problem. Each function receives a list of interger as input and outputs a list containing the distinct elements in the input list. The functions are named *solveOnlyLists*, *solveDict* and *solveSorted* and should be implemented using the following descriptions:

(a) *solveOnlyLists*: You should only lists in your solution. You are not allowed to use dictionaries or any other more sofisticated structure and also you are not allowed to use any type of sorting.

(b) *solveDict* Now you should try to improve the performance of the first solution by (possibly) using both lists and dictionaries (don't use any other structures). Again, you are not allowed to use any type of sorting.

(c) *solveSorted* Now you should solve the same problem assuming that the input list is sorted (you do not need to sort it, it is already sorted!). This function should improve the performance of the previous one.

We provide a sample code `http://vgc.poly.edu/files//gx5003-fall2014/week4/src/problem1.py` which contains the three functions you should implement. You should submit the file *problem1.py* with your implementations for the functions mentioned above. You do not need to be concerned with any formatting of the

output, try to solve the problem *correctly* following the specifications above and achieving the best performance.

In order to allow you to assess the performance of your implementation we also provided a script `http://vgc.poly.edu/files//gx5003-fall2014/week4/src/test_problem1.py` that allows you to visualize the performance of your solutions against a baseline functions called *solveWithSet* (which uses sets to solve the problem). To use it, you just run it as *python test_problem1.py*, this has to be in the same folder as the *problem1.py* file. This script records the performance of your implementation using random inputs. In case the matplotlib `http://matplotlib.org/` module is installed in your computer, it also produces a plot similar to Figure 1.

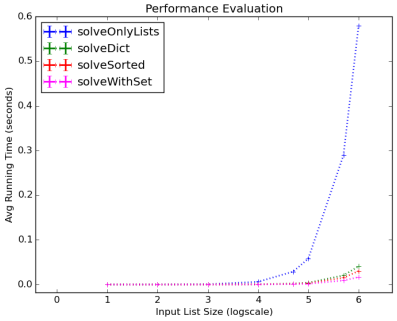

Figure 1: Performance test of the different solutions to problem 1.

If you do not have matplotlib, the sample code prints the plot data so you can use your favorite software to produce the plots. You should observe a pattern similar to Figure 1.

Problem 2

In this problem you are given a list of integer numbers L and your task is to perform searches in this list. The list might contain repeated elements, and in some cases it might not be sorted, so read the instructions of each item with attention. Given the skeleton code in file *problem2.py*:

(a) implement *searchGreaterNotSorted* to return the number n of items in L that are greater than v. Do not assume L is sorted. You are not allowed to use sorting, otherwise your solution will be considered wrong. Examples (n is the value *searchGreaterNoSorting* should return):

- $v = 2$ and $L = [5, 4, 1, 3, 5]$, $n = 4$
- $v = 5$ and $L = [2, 4, 2, 0, -3, 5]$, $n = 0$
- $v = 2$ and $L = [1, 0, 2, -1, 2, 10]$, $n = 1$

(b) implement *searchGreaterSorted* to return the number n of items in L that are greater than v. L is sorted in ascending order. You *cannot* use binary search

in this item, otherwise it will be considered wrong. Examples (n is the value *searchGreaterSorted* should return):

- $v = 2$ and $L = [1, 3, 4, 5, 5]$, $n = 4$
- $v = 5$ and $L = [-3, 0, 2, 4, 5]$, $n = 0$
- $v = 2$ and $L = [-1, 0, 1, 2, 2, 10]$, $n = 1$

(c) implement *searchGreaterBinSearch* to return the number of items n in L that are greater than v. L is sorted in ascending order, and you *must* implement a binary search (a modified one, since now you want the number of elements greater than a given value). Your solution will be considered wrong if you do not use binary search.
Some references to binary search:
`http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch`
`http://en.wikipedia.org/wiki/Binary_search_algorithm`
`https://www.youtube.com/watch?v=JQhciTuD3E8`

(d) implement *searchInRange* to return $n = |\{x \in L, v_1 < x \leq v_2\}|$, i.e., the number of elements in L that are greater than v_1 and less than or equal to v_2. Assume that $v_1 < v_2$, and that L is sorted in ascending order. Your solution must make calls to *searchGreaterBinSearch*, otherwise it will be considered wrong. Tip: two calls to *searchGreaterBinSearch* are enough to solve this problem. Examples (n is the value *searchInRange* should return):

- $v_1 = 1, v_2 = 2$ and $L = [1, 3, 4, 5, 5]$, $n = 0$
- $v_1 = -1, v_2 = 5$ and $L = [-3, 0, 2, 2, 4, 5]$, $n = 5$
- $v_1 = 2, v_2 = 11$ and $L = [-1, 0, 1, 1, 2, 10, 10]$, $n = 2$

To solve this problem, download and complete the skeleton code in file `http://vgc.poly.edu/files//gx5003-fall2014/week4/src/problem2.py`. To verify the execution times of your solution for lists with different sizes, run the script `http://vgc.poly.edu/files//gx5003-fall2014/week4/src/test_problem2.py`. After executing it you should see plots similar to Figure 2.

Problem 3

In this problem, you are given a list of points (x,y), with the values of x and y ranging from 0 to 1. You will also receive a rectangle ($[x_{min}, y_{min}] \times [x_{max}, y_{max}]$), and must output the number of points that fall inside the rectangle. A point that falls in the border of the rectangle must be considered inside the rectangle.

You have to count the number of points inside the rectangle in the three sub-items below, with each sub-item improving the previous data structure so that the count operation is performed faster.

Like the previous problems, we provide a sample code `http://vgc.poly.edu/files/gx5003-fall2014/week4/src/problem3.py`, which contains a set of functions that you should implement.

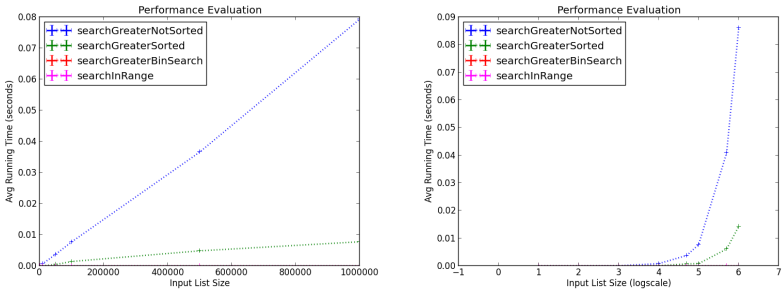

Figure 2: Time performance of different solutions for problem 2, in linear/log scales.

You should submit the file *problem3.py* with your implementations for the functions mentioned below.

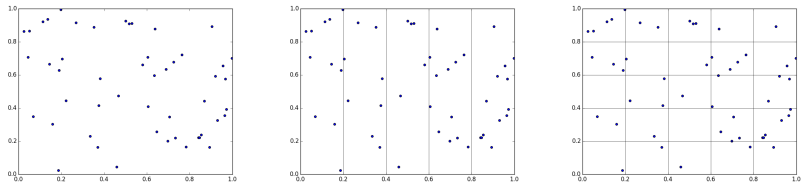

Figure 3: Space division in problems 3a, 3b and 3c.

(a) Solve the problem without worrying about performance. What is the most naive way you could perform this count? The left image in Figure 3 shows all points in the domain, without any kind of space division. For this problem, you must implement the functions *buildNaive* and *queryNaive*. The first function is responsible for building the data structure that will hold the data. It receives one argument (*points*); you must use these points to create a naive data structure that will hold all points. You can ignore the argument *n*.

The function *queryNaive* receives four arguments: (x0, y0, x1, y1). This is the rectangle that specifies the query range. You must use these points to query your naive data structure and return how many points fall inside the rectangle range (or in the boder).

(b) Repeat (a), but now segment dimension x in n partitions, where n is also an input. In other words, if in (a) all points are inside a list l, now you must divide that list l in n smaller lists, so that a list l_i only holds points inside a certain range. The center image in Figure 3 shows all points in the domain, but notice that each partition only holds a fraction of the points.

Similary, you have to implement two functions. The first one (*buildOneDim*) builds the data structure used for query; pay attention that now the build function

receives two arguments: *points*, similar to the naive solution, and *n*, wich is the number of partitions that you must use. The second function (*queryOneDim*) queries in the data structured that you just built and return the number of points that fall inside the rectangle (or in the border).

(c) Repeat (b), but now segment both dimensions x and y in n partitions *each*. The right image in Figure 3 shows all points in the domain, but now notice the partition in x and y.

Similary, you have to implement two functions. The first one (*buildTwoDim*) builds the data structure used for query, using *n* divisions for each dimension. The second function (*queryTwoDim*) queries the data structure that you just created.

Pay attention that the skeleton code also has defined the names of your data structures (*naive*, *onedim* and *twodim*). Do not change the names of those structures or the names of the functions.

We also supply a python file to plot the time of the queries (`http://vgc.poly.edu/files/gx5003-fall2014/week4/src/test_problem3.py`). Your plot should be similar to this:

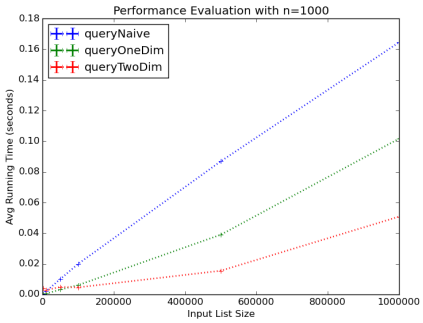

Figure 4: Plot for problem 3.

Note that the naive solution is slower than the two other solutions, and the division of only one dimension is slower than the division of two dimensions.

Questions

Any questions should be sent to the teaching staff (Instructor Role and Teaching Assistant Role) through the NYU Classes system.

How to submit your assignment?

Your assignment should be submitted using the NYU Classes system. Create a zip file with your source code in the files *problem1.py, problem2.py,* and *problem3.py* (**only**

these files). Name the zip file as NetID_assignment_4.zip, changing NetID to your NYU Net ID.

Grading

The grading is going to be done by a series of tests and manual inspection when required. Your functions should return the right values as specified. Try to test your code before submitting to minimize the need for manual inspection of the code, which can be very subjective. Also, we are going to check the performance of the functions; ideally you should obtain a pattern similar to the figures above.