Lab08: Quicksort
CSE2050 Fall 2017
Instructor: Jeffrey Meunier & Wei Wei
TA: Jenny Blessing, Param Bidja, Yamuna Rajan & Zigeng Wang

1. Introduction

In this exercise, we will complete two Python functions in order to implement the quicksort sorting algorithm. In the second half of the assignment, you will also compare the time this algorithm takes to run inputs of different sizes.

2. Objectives

The purpose of this assignment is to help you:

- Practice quicksort.
- See the efficiencies of quicksort with large lists in practice.

***Note**: Before you start, if you are not familiar with quicksort you are recommended to review your lecture notes and the additional materials from Dr. Sheehy on common sorting algorithms and divide-and-conquer sorting algorithms.*

3. Background

3.1. Quicksort

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we can simply use the last item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort. Once we've selected a pivot, the **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value. One advantage of quicksort over mergesort is that it is an in-place sorting algorithm, meaning it can sort a list using that exact list – it does not require copying elements into a new list. Also, as you'll see in the skeleton code, quicksort is a divide & conquer algorithm however requires no conquer step! Just calling the the partition step recursively leads to a sorted list. The worst case runtime of quicksort is $O(n^2)$ but with the choice of the right pivot, we can expect an average runtime of $O(nlogn)$. We'll leave a proof of this fact for an algorithms course.
Wikipedia has a nice explanation and gif showing the quicksort process. When using resources to learn about quicksort, **remember – the pivot can be selected using many strategies but we use the last element in the list as the pivot.**

4. Assignment

In this assignment, you are given quicksort(L, left, right) function and most of its partition(L, left, right) helper function, where **L** is the current list to be sorted and **left** and **right** are the indices of the left most element and right most element, respectively, of a sub-list.

4.1. quicksort(L, left = 0, right = None)

First, you should understand what the quicksort function is doing. You do not need to add any code, but understanding what's given will help you complete the following part. See how quick partitions and divides until the right and left indices do not match (or when the right index is to the left of the left index). This step is done recursively. Also notice how quicksort returns the cost of the sorting, meaning the number of comparisons needed for sorting the given list.

Note that the final line of the function, provided for you, contains three variables: **cost_p**, **cost_l**, and **cost_r**. The purpose of these variables is to determine the actual time, or cost, that the quicksort algorithm takes. **cost_l**, and **cost_r** are the result of recursively calling **quicksort**, while **cost_p** is the result of calling the **partition** (discussed in the following section).

4.2. partition(L, left, right)

Now, we will move to the **partition(L, left, right)** function. This is a helper function for the main **quicksort** function, and this is where the partitioning will be done. The goal of the partition function is to place the partition element in its correct place by moving all elements that are less than the pivot to its left and all greater than to its right. For example, suppose we partition the following list (i.e. calling partition([1, 5, 2, 0, 3], 0, 5))

Example:
1 5 2 0 3

We start by selecting the pivot, which will always be the last element in the list based on our implementation. Thus, the pivot is 3. Then we compare the current left and right values with the pivot to determine if they need to be moved, if so swap the current left and right elements.

We start by initializing our left pointer (let's call it i) to the left argument (0 in this case) and the right pointer (let's call it j) to the pivot – 1.

L = [1, 5, 2, 0, 3]
i = 0
j = 3
pivot = 4

So we first compare 1 and 3 (the pivot), and clearly 1 is less than 3 so we proceed with no swap. We increment i.

i = 1

Next we compare 5 (which is at i=1) and 3. 5 is greater than 3 so a swap is required. But we don't swap just yet, first we check if L[j] >= L[pivot] until we reach an index j that does not fulfill this condition. The current j value is 3, and L[3] = 0, which is not greater than or equal to the pivot. We'll swap the value at the current i index and the current j index only if i < j. Thus, we swap 5 and 0.

Swap 5 and 0
1 0 2 5 3

Now we compare 0 and 3. No swap needed so we increment i.

i = 2

We compare 2 and 3 to see that no swap is needed, so we increment i

i = 3

Now L[i] = 5 which is greater than 3 so we proceed to finding the next j index that needs to be swapped. Currently, j = 3, which means L[j] = 5. We check to see if 5 > 3 (notice, now we're checking if an element is > pivot because we're in the loop going right to left). 5 is indeed greater than 3 so we decrement j (j = 2). Now we check if 2 > 3, which it not true so we check the swap condition (is i < j). This is false, and our outer loop condition (is i <= j) is also false. This means we are complete. Thus the final result of this partition call is:
[1, 0, 2, 5, 3]

The given code defines the pivot and loop variables i and j. You should add your code for the partitioning process in the given loop (`while i <= j`). Break the partition process down into specific cases (i.e. the case with left element being less than or equal to the pivot, the case with the right element being greater than or equal to the pivot, etc) then translate those cases into code. The implementation is extremely simple, so you should not need to add more than 10-15 lines of code. Simply handle the cases (iteratively) of the left element being less than or equal to the pivot, the right element being greater than or equal to the pivot, and the case of the left index being less than the left index.

4.3 Scaling input size for quicksort

The skeleton code gives lots of code outside the quicksort and partition functions that is meant for you to see the performance of quicksort with different sized inputs. Once you have completed the partition function and tested it on your own to make sure it works, run the given code to see the cost of calling quicksort on different lists of varying sizes. We give you this code to help you see why quicksort is such a powerful sorting algorithm, especially when the input size is large.

Once you have implemented the partition function correctly, running the program should look something like this:

```
Sorting reversely sorted lists...
1
4
13
43
151
559
2143
8383
33151
131839
Scaling factors:
4.0
3.25
3.3076923076923075
3.511627906976744
3.7019867549668874
3.8336314847942754
3.911805879608026
3.9545508767744244
3.9769237730385205
Sorting already sorted lists...
1
4
13
43
151
559
2143
8383
33151
131839
Scaling factors:
4.0
3.25
3.3076923076923075
3.511627906976744
3.7019867549668874
3.8336314847942754
3.911805879608026
3.9545508767744244
3.9769237730385205
Sorting random lists...
1.0
4.0
12.14
34.59
92.44
```

```
232.21
562.95
1308.01
3052.43
6768.11
Scaling factors:
4.0
3.035
2.8492586490939047
2.672448684590922
2.512007788836002
2.4243141983549377
2.323492317257305
2.3336442381938975
2.2172859000861607
```

Notice how the scaling factors are different for different scenarios. An algorithm with $O(n^2)$ running time will have scaling factors approaching 4 when we double the list size as the list size becomes large, while an algorithm with O(nlogn) running time will have scaling factors approaching 2.

When sorting an already sorted list or a reversely sorted list, we see quicksort performs in $O(n^2)$. But in the random case it behaves more like O(nlogn). It's clear that in the worst case quicksort is $O(n^2)$, but on average it is O(nlogn).

5. Submit your work to Mimir

Submit your code to Mimir after you complete your code. Mimir will automatically your submission based on different unit test cases. These test cases **will only test the quicksort function**. Please comment out or delete any testing code at the bottom of your file you may have relating to the scaling (i.e. all the given code outside the quicksort and partition functions). You can submit your code to Mimir up to **30 times** to refresh your existing score before the submission deadline.

6. Due date

This lab assignment is worth **2 points** in the final grade. It will be due by **11:59pm on Wednesday, November 8th, 2017**. A penalty of **10% per day** will be deducted from your grade, starting at 12:00am.

7. Getting help

Start your project early, because you will probably not be able to get timely help in the last few hours before the assignment is due.

- Go to the office hours of instructors and TAs.
    - Prof. Wei Wei: Mon. 2:30 - 3:15pm, Wed. 2:30 - 3:15pm, Thurs. 2:30 - 3:15pm, Fri. 2:30 - 3:15pm @ITE258
    - Jenny Blessing: Fri. 12pm - 2pm @ITE140
    - Param Bidja: Tues. 2pm - 3pm @ITE140
    - Yamuna Rajan: Tues. 11am - 12pm, Wed. 9:30am - 10:30am @ITE140
    - Zigeng Wang: Mon. 3pm - 4pm @ITE140

- Post your questions on Piazza. Instructors, TAs and many of your classmates may answer your questions.
- Search the answer of your unknown questions on the Internet. Many questions asked by you might have been asked and answered many times online already.