COSC122 (2014) Lab 6.2

Sorting - n log n Algorithms

Quiz Due Date: 5pm Friday, 19 September

Quiz Drop Dead Date: 9pm Monday, 22 September

Goals

This lab will give you some experience with $O(nlog(n))$ sorting algorithms; in this lab you will:

- implement a median-of-three partitioning algorithm;
- implement a version of quicksort to sort certain ranges of a list; and
- use it to quickly find the median value of a list.
- use the merge sort principle to find the common items on two lists.

You should be familiar with the material in Sections 5.35 to 5.36 of the textbook[1] before attempting this lab.

Quick sort

The `quicksort` module contains a number of quicksort methods, stubs, and helpers:

- `quicksort`: Starts the recursive quicksort process for sorting a list. It creates a copy of the list, so that the original isn't modified by `quicksort_helper` (which sorts in-place). The `quicksort` function returns a new sorted list - and doesn't affect the original.
- `quicksort_helper`: Recursively partitions and sorts a list between the `left` and `right` indices.
- `pivot_index`: Returns the index of the pivot point to use (at the moment, it simply returns the left-most value).
- `partition`: Partitions the section of a list between the `left` and `right` indices. Uses `left-pivot` or `Mo3-pivot` - see below for details.
- `pivot_index_Mo3`: Returns the index of the median value of three items from the list. The three items are the left, right and middle.
- `quicksort_range`: Starts the recursive quicksort process for a specific range within a list.
- `quicksort_range_helper`: Recursively partitions and sorts a specific range within a list (you will implement this later in the lab).

The basic quicksort implementation is complete, and you can test it in the shell:

```
>>> from quicksort import quicksort
>>> alist = [4,6,3,9,2,3,1]
>>> quicksort(alist)
[1, 2, 3, 3, 4, 6, 9]
```

[1]Online text: see the Mergesort and Quicksort sections.

Median-of-Three Quick sort

Choosing a good pivot point is crucial to obtaining good quicksort performance. Ideally, you want to partition the list into two sub-lists that are equal in length—which means choosing a pivot that will be sorted to the middle of the list. However, you also want your pivot selection algorithm to be fast ($O(1)$), so you can't spend a lot of time looking around the list for a good value.

In the examples of quicksort you've seen so far, the pivot has always been either the first or last item in the list. If the list of items is randomly arranged (the best case), then this is a perfectly adequate pivot point; however, if our list already happens to be sorted (the worst case), then this pivot would create highly unbalanced sub-lists—degrading quicksort to $O(n^2)$.

A common approach to improving quicksort's partitioning algorithm is known as the *median-of-three* partition: we examine the first, middle, and last elements of the list and pick the median value of these three as our pivot.

As we are finding the median of three items we can simply sort them (using insertion sort makes sense here) and then the middle item is the median. For example, 3 is the median of the following lists `[1,3,6]`, `[3,1,6]`,`[1,6,3]`,`[3,1,3]` and 2 is the median of the following lists `[2,2,3]`,`[2,2,2]`,`[3,1,2]`,`[2,2,1]`.

The provided `quicksort` uses the left item as a pivot by default (eg, using `s = quicksort(alist)` implicitly uses 'left-pivot'). If quicksort is called using `'Mo3-pivot'` then the partition function will call the `pivot_index_Mo3` function to find the index of the pivot point to use— `pivot_i`.

> Your job is to implement the `pivot_index_Mo3` function and test the performance of quick sort when using it to find the pivot index (eg, `s = quicksort(alist,'Mo3-pivot')`).

DocTests are provided so that you can test your median of three function. You can comment out functions you don't want checked to avoid getting masses of irrelevant errors (eg, comment out `quicksort_range` at this stage).

Remember that `pivot_index_Mo3` is returning the index of the median value, not the median value. For example, `pivot_index_Mo3([5,4,3,2,12,14,10],0,4)` should return 0. This call is asking for the index of the item that is the median of the item at index 0, the item at index 4, and the item at index 2 (ie, the middle item). The three values to find the median for are `[5,3,12]` and therefore the median value is 5. The value 5 is at index 0 in the original list and therefore the function should return 0.

`ctrl-/` can be used to comment out a block of text. `ctrl-?` can be used to uncomment a block of text. [2]

Add your own doctest(s) to the quicksort function to ensure that quicksort still works when called with `'Mo3-pivot'`.

> ***Complete question 1 in Lab Quiz 6.2.***

Timing the Difference

The new pivot selection method doesn't change the big-O performance complexity of the quicksort implementation (for lists of random items), but it still gives us a speed-up when sorting *real-world* data.

The `quicksort_trials.py` file provides a graph[3] of the time taken to sort ascending value lists of various lengths, using the default left-pivot quicksort.

[2] In windows, at least

[3] Lab computers have matplotlib installed. If you are running Python 3.3 on your own computer then you will need to make sure you have Numpy and Matplotlib installed. If not, then install Numpy and then Matplotlib —in that order. You can download them from `http://sourceforge.net/projects/numpy/files/NumPy/` and `http://matplotlib.org/downloads.html`. Make sure you download the right version for your operating system and that you download the Python3.3 version. Ask a tutor if you are having trouble.

You should now add in trials for random lists of values and graph them on the same graph - you can add any number of extra series to the plot function (eg, `pyplot.plot(x,y,'bo',x1,y1,'go')` plots two series; one with blue o's and one with green o's - 'bx' would do blue crosses, etc)[4]. The pyplot window may come up in the background and you should also remember to close the pyplot window if you want your program to finish.

You can generate sorted lists with `range` as per the trial that is provided. Random lists can be generated using the `random.shuffle(mylist)` function:

```
test_list = list(range(800))  # test_list is a sorted list
random.shuffle(test_list) # mixes items in to a random order
```

Generate and print some sorted and random lists if you are unsure what is happening...

> *Note: Running quicksort with lists of greater than about 900 items will hit Python's internal recursion limit when lists are close to the worst case. Try it and see...*

How does the time for sorted lists compare with the tim for random lists?

Now, generate graphs for quicksort with the `'Mo3-pivot'` setting and compare the speeds. You should be able to graph all four runs on one graph, ie, sorted/random vs left-pivot/Mo3-pivot. These graphs should give a feel for how sort time is related to the number of items in the list and the pivot method.

> ***Complete questions 2 to 4 in Lab Quiz 6.2.***

Once you have finished playing with the graphs you should fill out Table 1 with the average time taken to sort 760 items in the four situations given. To calculate the time you should set the number of trials to 100, that is run each sort one hundred times and take the average time. When calculating ratios for the lab quiz you should try to do it in Python so that you don't get any rounding issues (or you can use average times that are written to 5 or 6 decimal places in your ratios).

	Sort time for *random data* *n=760*	**Sort time for** *ascending data* *n=760*
Left-Pivot Quicksort		
Median-of-three Quicksort		

Table 1: Your results from various quicksort runs with n=760 using 100 trials.

> ***Complete questions 5 to 7 in Lab Quiz 6.2.***

Sorting a Range

Sometimes, you won't want to sort the *entire* list of items, but only a small range of items. For example, say we only want to see the first 10 search results (eg, the first page) from a Google search that returns 1,000,000 results — that is, the slice [0 : 10]. We want the items in the [0 : 10] slice (or, range) to be sorted in the order they would be in if we sorted the entire list. This means that the items for that range can come from any part of the list, and they will be placed in such a way as if the whole list was sorted. For example, if you were to sort the range [8 : 10] of the list `[1, 8, 9, 5, 6, 4, 2, 0, 10, 7, 3]` you should receive something similar to: `[0, 1, 2, 3, 6, 4, 5, 7, 8, 9, 10]` Note that the last three items (8–10) are sorted correctly and in the right position (ie, they are in the four top slots), but the rest of the list isn't.

[4]Marker styles use two chars with first being colour and second being type - feel free to experiment. Or, check the interwebs for information on pyplot marker styles

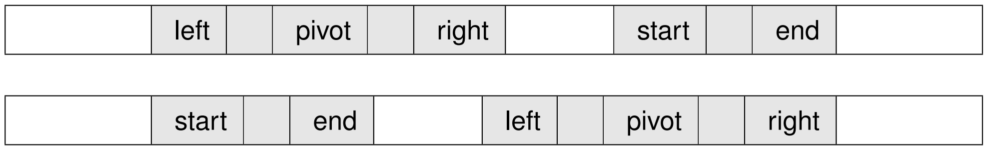

Figure 1: The two cases when `quicksort_range_helper` does not need to be called recursively.

To do this, we'll pass in two extra parameters to our quicksort method: `start` and `end`, which specify the indices of the range to sort. The algorithm is almost the same as the `quicksort` method—it chooses a pivot, and partitions the list around it; but instead of recursively sorting both sides, it only calls the sort for a side if the [`start` : `end`] range overlaps with that side.

It may be easier to think of this in terms of when you should *NOT* recursively sort, shown in Figure 1.

Implement the `quicksort_range_helper` method to sort a specified range of values, using a copy of the `quicksort_helper` code as a starting point and adjusting it so that only necessary recursive calls are made.

`quicksort_range_helper` can use the same `partition` function.

`quick_sort_range` can still be run with `'left-pivot'` or `'Mo3-pivot'`.

For a general definition of Median see the Extras section of this lab - you will need to understand this in order to answer some of the quiz questions.

> ***Complete questions 8 to 10 in Lab Quiz 6.2.***

Finding common items

Suppose you have been given the job of finding students who worked for the volunteer army after both the September and February earthquakes, so you can send them a token of thanks from the Cadbury chocolate company. You have a list of student ids from each army and you want a method of quickly listing ids that are in both lists.

The `common.py` module provides the stub for a `common_items(list_x,list_y)` function. Your job is to implement the function so that it returns a list containing the ids that are common to two lists. You should sort the lists and then use the *merge sort* principle to generate the list of common items. You should include each common item only once in the output list. Doctests are provided to help you check your implementation.

To help you check that your routine works we have provided five list files for testing, ie, `list0.txt` to `list4.txt`.

> ***Complete questions 11 to 14 in Lab Quiz 6.2.***

Extra Exercises

Implement a `quick_median` method to use the `quicksort_range` method to find the median of an unsorted list. You should call `quicksort_range` with the appropriate `start` and `end` values (remember that the range we are interested in is different for odd-sized lists than that of even-sized lists).

> In statistics, the median M of a quantitative data set of size n is the middle number when the values are arranged in ascending (or descending) order. If n is odd, M is the middle number; if n is even, M is the mean of the middle two numbers.

Test the method to see how much faster it is for finding the median value, than sorting the *entire* list and extracting the median.