CSC148 Summer 2018: Lab 9

Introduction

The goals of this lab are:

- To get you familiar with Merge Sort
- To get you familiar with Quick Sort
- To give you practice with runtime analysis

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

General Lab Notes

1. Make sure you have lab_pyta.txt downloaded and placed in the directory (or directories) where you'll be working.
2. To use PythonTA, include the following code (if you already have a main block, just add the body to the end of it):

   ```python
   if __name__ == '__main__':
       import python_ta
       python_ta.check_all(config="lab_pyta.txt")
   ```

Your lab_pyta.txt should be in the same folder as the .py files you're running. PythonTA will raise errors regarding style, specifying the lines you need to fix. You should get familiar with what the errors mean, and how to fix them: this will be important for your exercises and assignments.

Merge Sort

Recall the merge sort algorithm from lecture: In it, we take a list and split it into 2 halves which we sort recursively. After sorting both halves, we merge the 2 lists together.

Make sure you understand how the merge sort algorithm works. For example, suppose we have the following list:

```
6, 2, 3, 7, 5, 1, 4, 8
```

What halves do we split it into?

What do we expect back if we call mergesort on both halves?

What would we expect back if we merged the two sorted halves?

To convince yourself the merge sort works, try to keep breaking down the halves and then merge them at each step. For example, if you had the list:

```
5, 3, 2, 1
```
The two halves that come from this would be:

5, 3  
2, 1

And, splitting these into further halves would get us:

5  
3  
2  
1

Those are all single items, so there's nothing else to split. When we try to merge [5] and [3], we would expect [3, 5] to be returned. The same goes for [2] and [1], which should return [1, 2]. So if we have:

3, 5  
1, 2

What should we get back from merging these two? Is that a sorted version of our original list?

The trickiest part of mergesort is likely in the 'merge' step. Without referring to the code from lecture, write a function called `merge_lists` which takes in 2 sorted lists and returns a single list with all of the elements from both lists in sorted order. For example:

- `merge_lists([1, 2, 3], [4, 5, 6]) == [1, 2, 3, 4, 5, 6]`
- `merge_lists([1, 3, 5], [2, 4, 6]) == [1, 2, 3, 4, 5, 6]`
- `merge_lists([3], [2, 4]) == [2, 3, 4]`

What's the runtime of your `merge_lists()` function? How many times does it have to look at each item in each list?

Afterwards, write your own implementation of mergesort that uses `merge_lists`.

**Merge Sort (In Place)**

Python's built-in `sort()` method sorts the list that it's called on instead of returning a new list. This means that all recursive calls are also applied to the original list and its halves.

When we modify lists, we do so by assigning to indices in the list. Thus, if we want to modify the original list passed in, we need to pass the original list into the recursive calls. Since we're only working with halves/parts of a list at each recursive call, we need to know which indices should be changed in our current call.

Below is a function that calls on another function to perform in-place mergesort:

```python
def mergesort_in_place(lst: list) -> None:
    
    # Sort lst in non-descending order using merge sort.
    mergesort_in_place_helper(lst, 0, len(lst))
```

And below is the function header and one part of the code for `mergesort_in_place_helper`:

```python
def mergesort_in_place_helper(lst: list, i: int, j: int) -> None:
    
    #
Sort lst[i:j] in non-descending order.

# TODO: Make recursive calls
lst[i:j] = merge_lists(______) # Fill this in

This function takes in 2 indexes representing the start and end of the part of the list we're trying to sort. It calls on the merge_lists you've written previously as well, to help with the merging step.

Fill in the recursive calls that need to be made, as well as what you should pass into merge_lists.

Quick Sort

Recall the quick sort algorithm from lecture: In it, we take a list and split it into 3 parts, 2 of which we sort recursively. After sorting the 2 parts, we add the 3 lists together and return it. The 3 parts are as follows:

- The first contains only items less than our pivot.
- The second contains only items equal to our pivot.
- The third contains only items greater than our pivot.

We can use any criteria to choose our pivot; the more complicated 'choosing our pivot' becomes, the more time our quicksort algorithm will take. Similarly, the worse our pivot is, the worse the runtime of quicksort will be.

Common choices for a pivot are either: The item at index 0 in the list, the item in the middle of the list (i.e. at index len(lst) // 2), or the item at index -1 int hs list.

Make sure you understand how the quick sort algorithm works. Suppose we have the following list:

6, 2, 3, 7, 5, 1, 4, 8

Pick a pivot criteria that you want to use and create the 3 parts. Recursively try to sort the list manually to convince yourself of how quick sort works.

In merge sort, we had to manually merge the lists together. For quick sort, we can simply return the sum of the 3 lists, and call it a day. Why?

Quick Sort (In Place)

Similar to merge sort, we can try to do quick sort in place. This also requires us to work with indices, so consider the following functions:

def quicksort_in_place(lst: list) -> None:
    
    Sort lst in non-descending order using quick sort.
    
    quicksort_in_place_helper(lst, 0, len(lst))
And below is the function header and one part of the code for `mergesort_in_place_helper`:

```python
def quicksort_in_place_helper(lst: list, i: int, j: int) -> None:
    
    # Sort lst[i:j] in non-descending order.
    pivot_index = partition_list(lst, i, j)
    # TODO: Make recursive calls
```

This function takes in 2 indexes representing the start and end of the part of the list we're trying to sort. It calls on a function called `partition_list` which you are to write.

The header for `partition_list` is as follows:

```python
def partition_list(lst: list, i: int, j: int) -> None:
    
    # Partition lst such that all items < our pivot comes before our pivot which comes before all items > the pivot.
    
    Return the index of the pivot.
```

Write the body of `partition_list` and fill in the recursive calls that need to be made in `quicksort_in_place_helper`.

As a hint: you may want to swap items in `partition_list` based on the pivot's value. Start by looking for the first item in the list > pivot, and the last item in the list < pivot, and swap those. For example, if you have the pivot 5 and the list:

```
8, 6, 1, 5, 3, 2, 7, 4
```

You would try to swap 8 and 4 since 8 > 5 and 4 < 6:

```
4, 6, 1, 5, 3, 2, 7, 8
```

Afterwards, look for the next item that's > pivot and the next item < pivot: that would be 6 and 2. Swap those:

```
4, 2, 1, 5, 3, 6, 7, 8
```

Eventually the only thing left to do is to put the pivot in the right position -- putting it into the border between the last item < pivot and last item > pivot.
Runtime Analysis

Re-visit the code we've written throughout the semester and try to find the runtime of various operations. A breakdown of various topics covered and operations of interest are below:

Stacks and Queues

- For a stack implemented using a list:
  - How long does it take to add to the stack if we append to the end of the list?
  - How long does it take to remove from a stack if we're removing from the end of the list?
  - How long does it take to add to the stack if we insert into the front of the list?
    As a hint: Python's `insert()` method shifts all items in a list over. This takes time proportional to the index we're inserting into -- i.e. if we're inserting into position -5, then 5 items need to be shifted.
  - How long does it take to remove from a stack if we're removing from the front of the list?
  - How long does it take to check whether a stack is empty?
- Do the same for queues implemented with a list (adding to the front of the list, the end of the list, removing from each, etc.)
- What about a stack/queue implemented using a dictionary?
- How about a linked list?
  - What if we added to the front/back of the linked list? Removed from the front/back?

Linked Lists

- How long does it take to add to the start of a linked list?
  - What about the end?
  - What if we wanted to insert into a certain position (i.e. some index)?
- How long does it take to search for a value in the linked list?

Trees

Consider the best and worst-case runtimes. What's the best-case scenario for each of these? Worst-case?

- How long does it take to add a subtree to a certain value in a tree?
- How long does it take to make a tree and set another tree as one of its children?
- How long does it take to search for a value in the tree?
- How long does it take to find the height of a tree?
  - What about the arity?
  - Getting all of the leaves?
  - Internal nodes?
  - The largest value?
  - Smallest value?
Do these things change for a binary tree?

**Binary Search Trees**

Consider the best and worst-case runtimes. What's the best-case scenario for each of these? Worst-case?

- How long does it take to add a value into a binary search tree?
- How long does it take to search for a value in a binary search tree?
- How long does it take to find the height of a binary search tree?
  - What about the arity?
  - Getting all of the leaves?
  - Internal nodes?
  - The largest value?
  - Smallest value?

**Other**

For experiments regarding runtime analysis, try working through [Lab 8 from CSC148 Winter 2018](#). In this, you’ll be performing experiments to see the runtime of different algorithms. If you’d like, try to write your own functions to try and time. Do their runtimes match the times you get back?