CS-UY 1134: Data Structures and Algorithms
Lab 5: Recursion and Sorting

Friday, June 28, 2019

Instructions

This lab will focus on recursion, with a little bit of practice with sorting.

Try to solve all of the problems by the end of the lab time. As added practice, it is a good idea to try writing your code with pen and paper first, as you will have to do this on the exam.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

As a reminder, you *may* (and are encouraged to!) discuss these problems with your classmates during lab and help each other solve them. However, make sure the code you submit is your own.

What to submit

If you believe you have solved all the problems, let a TA or the professor know and they will check your work. This is a good chance to get feedback on problems similar to what you might see on the exam.

If you don't finish by the end of the lab, you may submit something on NYU Classes by Sunday June 30, 11:55 PM.

Either when a TA has checked your solutions, or by the Sunday submission deadline, submit your solutions to the "Lab 5: Recursion" assignment on NYU Classes.

1 Binary Search

Implement the binary search algorithm as a recursive function. You may want to include `low` and `high` indices as parameters, as we've done with some other recursive functions. In this case, your function should look like:

```
def binary_search(lst, val, low, high)
```

Defined this way, the function takes a list `lst` and searches for the value `val` in the portion of `lst` between the indices `low` and `high` and return the index of `val`.

Note: Do **not** use slicing anywhere in your implementation.

2 Nested Lists Sum

A *nested list* of numbers is a list whose elements are all numbers or other nested lists of numbers. This creates a sort of *hierarchy*. As an exaple, the list `[1, [2, 3], 4, [5, [6, [7, 8]], 9], 10]` is a nested list of numbers.

Write a python function

```
def nested_list_sum(lst)
```

that takes a nested list of numbers and returns the sum of all the numbers in the nested list. For example, if `lst = [1, [2, [3], [4, 5]], [6, 7]]`, the function call `nested_list_sum(lst)` should return 28, since $1+2+3+4+5+6+7=28$.

Note: Recall that the focus of this lab is recursion. You may try to implement this without recursion, and are actually encouraged to — but you will likely find it very difficult. Using recursion will make things much easier.

3 Selection Sort

3.1 Part 1, Iterative

Write the function

```
def selection_sort(lst)
```

that implements the selection sort algorithm covered in class. This function will *modify* `lst` so that the elements are in *increasing order*. Recall that the algorithm works by first finding the smallest value in the list and swapping it into index 0; then finding the next smallest value and swapping it into index 1; and so on.

In your implementation, do not use the built-in python function `min`. You may find it useful to implement your another function `find_min(lst, low, high)` to find the minimum element of the portion of `lst` between the indexes `low` and `high`.

Recall that selection sort should run in time $O(n^2)$. Your implementation should also run in time $O(n^2)$.

3.2 Part 2, Recursion

Next, implement selection sort as a **recursive** algorithm, as a function called `selection_sort_recursive`. In order to do this, use additional parameters by using the function header

```
def selection_sort_recursive(lst, low, high)
```

This function will sort the portion of `lst` between the indices `low` and `high`. An initial call of `selection_sort_recursive(lst, 0, len(lst)-1)` would thus sort all of `lst`.

This should work by first finding the smallest element in the portion of `lst` between the indices `low` and `high`, swapping this element to index `low`, and then (recursivly) sorting the portion of `lst` between the indices `low+1` and `high`.

As with the iterative approach above, do not use the built-in `min` function for the minimum-finding step of the algorithm. You may once again find it useful to implement a function `find_min(lst, low, high)` to find the minimum element of the portion of `lst` between the indexes `low` and `high`. For added recursion fun, implement this subroutine in a recursive way as well, as we've done previously.

4 A New Sorting Algorithm

Recall the `partition` function of Lab 4. This function takes a list `lst` and chooses a value called the *pivot* (in Lab 4, we used `lst[0]` as the pivot) and modifies the list so that all the elements smaller than the pivot appear before all the elements greater than the pivot.

In this problem, we will use the partition algorithm as the key part of a new sorting algorithm. The algorithm will be *recursive*. The idea is as follows. First, we partition the list using the `partition` subroutine. After this step, the list will be partitioned into two *parts* — a left part and a right part. The left part contains all values smaller than the pivot. We sort this part recursively. The right part contains all the elements greater than the pivot (and thus, also greater than all elements in the left part). We sort the right part recursively. After this step, the entire list should be sorted, because we have sorted both the left and the right parts, and all the elements of the left part are smaller than all the elements of the right part.

This sorting algorithm is known as *Quicksort* because it tends to be quite fast in practice. Implement this approach in a method called `quicksort`. You will probably want to include `start` and `end` parameters, as in:

```
def quicksort(lst, start, end)
```

which will sort the portion of `lst` between the indices `start` and `end`. This function will implement the strategy described above, partitioning `lst` into a left part (consisting of all elements smaller than the pivot) and a right part (consisting of all elements greater than the pivot); and then sorting (recursively, using `quicksort`) the left part and then the right part.

As an example, suppose the portion of `lst` we are currently sorting (between the indices `start` and `end`) contains the elements: `[42, 17, 81, 77, 68, 22, 55, 10, 90]`. When calling `quicksort(lst, start, end)`, the partitioning step may rearrange this portion of `lst` (between `start` and `end`) to look like:

```
[22,   17,     10,       42,     68,   77,   55,   81,   90]
start        end_left                                    end
  {    < pivot    }     {pivot}  {         > pivot          }
```

Note that the left part (containing the elements less than the pivot) is between the indices `start` and `end_left` (the elements in the left part are blue), and the right part is between the indices `end_left+1` and `end` (the elements in the right part are red). While `start` and `end` are the starting and ending indices given as parameters, `end_left` will of course not be known in advance and can only be known after the partitioning is complete.

At this point, we can recursively sort the portion of the list between `start` and `end_left` (the left part), which would result in:

```
[10,    17,     22,     42,   68,   77,   55,   81,   90]
start         end_left                                end
```

After recursively sorting the left part, we recursively sort the portion of the list between the indices `end_left+1` and `end`:

```
[10,    17,     22,     42,   55,   68,   77,   81,   90]
start         end_left                                end
```

After recursively sorting the right part, the entire portion of the list from `start` to `end` is sorted!

If you've finished this problem: Congratulations! You've just implemented *quicksort*, one of the most famous and widely used sorting algorithms! This is no small feat. Like the merge sort algorithm from class, it is a *divide and conquer* algorithm, leveraging the power of recursion to sort a list.