Divide and Conquer

Divide and Conquer is a paradigm for algorithm design. It is composed of three parts: divide, conquer, and combine. In the divide part you decompose your problem into two or more sub-problems. In the second part you solve each sub-problem separately, while in the third part you combine the solutions of the sub-problems to create the solution for the entire problem.

For example, in the merge sort algorithm that you have seen in class, the original list was divided into two (the divide phase). Then each sub-list was sorted (the conquer phase). The last step combined the two sorted sub-list by merging them into one (the original list that we wanted to sort).

In this lab, we will look at Quicksort and Find-Kth as examples of divide-and-conquer algorithms.

Objectives:

1. Recursion
2. Divide-and-conquer approach
3. More practice using linked lists
4. Converting linked list algorithms to in-place algorithms
5. Using the key function for searching and sorting
6. Pivot selection
7. Splitting a list on a pivot

Part 1: Quicksort using Linked Lists (in-lab)

In part 1, you will implement the Quicksort algorithm using Linked Lists.

You will be provided a `LinkedList` class implementation to use. You will also be provided a `LinkedList` implementation of `mergeSort`, which you can use for reference.

Here is the pseudocode for your function `quickSortLinked(L)`:

1. Take the value of the last element in the input linked list to be your `pivot` (use `peekLast` method of `LinkedList`, which does not remove the item from the list)
2. separate the input list into two lists, calling `(list1, list2) = splitLinkedList(L, pivot)`
3. recursively sort the two lists

4. concatenate the two lists into one (using class method `concatenate`)

The only tricky part in the code above is step 2, where you will need to write a separate function `splitLinkedList` and call it. Given some value `pivot`, this function separates all the items in list `L` into two new lists:

1. `list1` - elements smaller or equal to `pivot`
2. `list2` - elements larger or equal to `pivot`

It initializes two new lists, and traverses the input list. It compares each item to the `pivot` and adds it to one of the lists (use `addLast` method). If it's smaller, it's added to the 1st list; if it's larger, it's added to the 2nd list, if it's equal, add it to SMALLER list (how can you tell which is smaller without having to traverse both lists to find out?).

It returns these two lists as a tuple `(list1, list2)`.

```
L = [25, 20, 5, 10, 30, 17]
LL = LinkedList(L)
pivot = LL.peekLast() # will be 17
(list1, list2) = splitLinkedList(LL, pivot) # list1 and list2 are linked lists
print(list1) # will contain 5, 10, 17 – not necessarily in that order
print(list2) # will contain 20, 25, 30 – not necessarily in that order
```

Part 2: In-place QuickSort

The code for part 1 would also work if we were using regular lists instead of linked lists. However, with the regular list, there is an alternate implementation of Quicksort where all the work is done in the single list, without creating a new list.

Our recursive function will have two additional parameters, `startIdx` and `endIdx`. These would be indices into the list indicating where it starts and ends, originally `0` and `len(L) - 1`.

As we separate the list into two parts, we are keeping both parts IN the original list. Let's say the list has 17 items, 5 smaller than the `pivot`, and 11 larger, and the `pivot` itself. Then after `splitList` (also called `partition` in some texts), 5 smaller items will appear before the `pivot` and 11 larger items will appear after the `pivot`. The `pivot` itself will be in the 6th place (with index 5).

Write the function `splitList(L, pivot, startIdx, endIdx)`. It needs to keep track of two items:

1. index `i` of the first item starting from left that is larger than the `pivot` (initially `startIdx`)
2. index `j` of the first item starting from right that is smaller than the `pivot` (initially `endIdx`)

Swap the items at `i` and `j`. Repeat this process until `i` becomes greater than `j`. Finally swap the item at index `i` with the `pivot` element. Now our `pivot` should be in the correct spot. This means that the 5 elements smaller than the `pivot` will now be to its left and the 11 elements larger than the `pivot` will be to its right.

Note: The function `splitList` should also return the index of the `pivot`.

Now write the recursive function `quickSortInplace(L, starIdxt, endIdx)`, which calls `splitList(L, pivot, startIdx, endIdx)`.

Part 3. Keys

As in the previous lab, we do not want to compare the elements directly. This time, instead of using COMPARE, we will use a different way to make comparisons generic, using a `key`. In this part, we will add a new parameter `keyFunc` to quicksort that will make the sorting generic, similar to what we did in the last lab.

To better understand how keys work, let us examine how the `sort` method works on Python lists.

Assume you have a list `L = [19, 845, 791, 8]`. To sort this list by the item value you would write: `L.sort()`. The expected result is: `8, 19, 791, 845`. However, if we want to sort the items according to the rightmost digit of each item, we can provide a key function `rightMostDigit` that receives an item and returns its rightmost digit.

```
def  rightMostDigit(item):
    return item % 10
```

This function is passed as a key parameter to the `sort` function. Here is how the code snippet would look:

```
L = [19, 845, 791, 8]
L.sort(key = rightMostDigit) # the key function is rightMostDigit
print(L) # output will be [791, 845, 8, 19]
```

This is part of standard Python, try it out in your Python shell!

Note that this algorithm also works to sort in decreasing order. You do not need to change the sort code at all -- just change the key function to return its negative!

```
def  rightMostDigitNeg(item):
  return - 1 * (item % 10)

L = [19, 845, 791, 8]
L.sort(key = rightMostDigitNeg) # the key function is rightMostDigitNeg
print(L) # output will be [19, 8, 845, 791]
```

Try this out in your Python shell, too!

In this part of the lab, we will leave `quickSortInplace` from part 2 as-is, and create a new function `quickSort`:

```
quickSort(L, keyFunc, startIdx, endIdx)
```

Call the `splitList` function as `partition`:

```
partition(L, keyFunc, startIdx, endIdx, pivot)
```

At first, `quickSort` is a copy of `quickSortInplace`.

Your sorting function `quickSort` will have a new `keyFunc` parameter. This parameter should receive a predefined function that receives a list item value and returns its key value, as discussed above.

With the key function defined, `sort` will call it every time it need to perform a comparison. For example, the comparison `L[i] < pivot` would be replaced by `key(L[i]) < pivot`. Remember, the `pivot` itself should also be a key!

Here is what you should be able to do now:

```
def rightMostDigitNeg(item):
  return - 1 * (item % 10)

L = [19, 845, 791, 8]
quickSort(L, rightMostDigitNeg, 0, 3)
```

```
print(L) # output will be [19, 8, 845, 791]
```

Note: The default value for keyFunc is identity, which is a function that just returns its argument.

```
def identity(item):
    return item
```

Part 4: Bells and Whistles

In this part, we improve `quickSort`. We make two parameters optional, and change how we choose the `pivot`.

a. Index Parameters

We need to make the parameters `startIdx` and `endIdx` optional, so our call to `quickSort` looks like Python's call to sort. When the user calls `quickSort`, they should not have to pass them in.

The initial value of `startIdx` is `0`, so that should be the default value for it.

The initial value of `endIdx` is `len(L)-1`, but we can't just use that as default value. Consider the following code:

```
L = [1,2,3,45]
def foo(L, startIdx=0, endIdx=len(L)-1):
  print(startIdx, endIdx)

L2 = [1]
foo(L2)
```

foo will print (0, 3) because `endIdx` will be set to the length of `L` rather than `L2`! This is because the value of `endIdx` got set when we defined `foo`, rather than when we actually called it. That's why default argument values must be constants.

So the proper way to do it would be to have some other default value for endIdx such as `-1`:

```
def foo(L, startIdx =0, endIdx =-1):
  if (endIdx == -1):
```

```
        endIdx = len(L)-1
    print(startIdx, endIdx)
```

Now things should work as expected, and calls to our `quickSort` will look more like calls to Python `sort`.

```
def rightMostDigitNeg(item):
  return - 1 * (item % 10)

L = [19, 845, 791, 8]
quickSort(L, rightMostDigitNeg)
    print(L) # output will be [19, 8, 845, 791]
```

b. Pivot Selection

We do not want to take the key of the first element as `pivot`. The quicksort algorithm works best when the two lists you get with `splitList` are about equal in length. With some inputs, such as an ordered list (or a reverse-ordered list), using first element will give you very unbalanced lists.

Instead, sample three elements: 1st, last, and middle, and use the median one as your `pivot` value.

Change `quickSort` code to do this. You should see a big difference in the case when sorting a list that happens to already be sorted. Try the following code:

```
import time

L = [i for i in range(1000000)]
starttime = time.time()
quickSortInplace(L, 0, 999999)  # this is still using old pivot selection
print(time.time() - starttime)

starttime = time.time()
quickSort(L)
print(time.time() - starttime)
```

The time performance of the second call should be much faster, even though `quickSort` is doing the extra work of calling the key function for every comparison!

Part 5. Finding k^{th} smallest element

In this part, we will use the divide-and conquer approach to solve a different problem: find the k^{th} smallest element in a list. For example, in a list `[13, 1, 5, 9, 11, 7]`, the third smallest element is `7`.

It is easy to find it if the list is sorted -- just take the k^{th} value! So we can implement findKth as follows:

```
def findKthSlow(L, k):
    L.sort()
    return L[k]
```

However, there is a faster recursive algorithm, without sorting the list. Its structure should remind you a lot of Binary Search. Its use of pivots is very similar to Quicksort:

```
def findKth (L, k):
    choose a pivot
    split list on the pivot
    if length of list1 >= k:
        recursively call findKth (list1, k)
    else:
        new_k = k - length of list1
        recursively call findKth (list2, new_k)
```

We have provided you `findKthLinked`, a simple `LinkedList` implementation of `findKth`. It has an optional `loud` parameter, by default `FALSE`. When it's `True`, `findKthLinked` prints out the value of `pivot` and how the lists are split at each step. If `k` is larger than `len(L)`, it returns `None`.

Try it out for yourself, for example:

```
L = LinkedList ([13, 1, 5, 9, 11, 7])

# find 3rd smallest element, prints out details along the way
x = findKthLinked(L, 3, true)

print(x) # x is 7
```

Rewrite `findKth` as an in-place algorithm on regular lists. You will be choosing the `pivot` and splitting on it exactly as in `quickSort`. You will be using key function and providing default index values exactly as in `quickSort`.

```
# should return same value as findKthLinked above
x = findKthInplace(testList17, 5)
```

Note that your algorithm should work to find k^{th} largest value! You should not need to change the code for that -- just change the key function:

```
def  negative(item):
  return - 1 * item
x = findKthInplace(testList17, 5, negative) # returns 5th LARGEST
```

How much did we improve the running time of `findKth` by avoiding sort? Try the following code, varying the values for `p` and `k`:

```
import time

k = 1000000 # should be a LARGE number
p = 500000 # should be smaller than k
L = [random(0, k) for i in range(k)]  #initialize L

starttime = time.time()
# note we pass in copy of L, so original is not destroyed
x1 = findKthSlow(L[:], p)
print(time.time() - starttime

starttime = time.time()
# note we pass in copy of L, so original is not destroyed
x2 = findKth(L[:], p)
# x1 and x2 should be the same, but times should be different
print(time.time() - starttime)
```