**CS 2302 Data Structures**

**Spring 2019**

**Lab Report #2**

Due: September 20, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were asked to implement sorting algorithms and search for the kth smallest element in a list. K would be given by the user to search in the list. It was important to implement sorting algorithms since only then would the kth smallest element be able to be accessed correctly. The main objective of this lab was to understand lists and manipulate sorting algorithms. The sorting algorithms were manipulated by not using recursion as one would usually do or by using stacks in place of recursion.

**Proposed Solution Design and Implementation**

**Part #1**

For Part 1, I first had to translate the source code I had previously written in java for sorting algorithms. This is due to having a lab in CS 2 that required me to write iterative and recursive sorting algorithms. The translation from java to python was seamless with the main issue coming in part 2. Select\_bubble had two parameters which took the list L and the index K. Bubble sort is a fairly simple sorting algorithm that compares an element with the one next one and makes swaps when necessary. The select\_bubble function calls on bubble sort, once the list is sorted it returns the value of the list at index k. Select\_quick also receives two parameters which are list L and index k. Quick sort is an extremely efficient sorting algorithm which chooses a pivot through a partition function call. This partition creates sublists with the elements in each sublist being swapped when necessary. The select\_quick functions call on quick sort. Once the list is sorted it returns the value of the list at index k. The select\_modified\_quickSort function receives the parameters of list L, index of 0, length of the list minus 1, and k. This function serves as a modified version of quick sort because it relies on the function of partition which essentially creates sublists. This modified function compares k to the pivot. If k is less than the pivot, then k is present in the first sublist hence only this sublist needing to be sorted while the other one is simply not needed. If k is greater than pivot than k is in the second sublist. In this case we only need to sort this sublist and not worry about sorting the first sublist. If pivot is the same as k then you return the list at index of the pivot. These three sorting algorithms presented an interesting challenge with a greater understanding for quick sort being cultivated.

**Part #2**

For Part 2 we had to apply extra modifications on the sorting algorithms we had already implemented. The first modification was implementing quick sort using a stack instead of recursion. At this point of the lab I struck gold since I already had code written for quick sort using a stack. The main issue was the translation from java to python. I faced the most difficulties at this point since my code contained a substantial number of operators that pertained to the syntax of java. To make my previous code work in python I had to assign and increase values an old-fashioned way instead of relying on operators. Quick sort using a stack still called partition with the only difference being that elements that were less than the pivot was pushed to the right and elements greater than the pivot were pushed to the right of the stack. The second modification was to implement the select\_modified\_quick function using only a while loop without stacks or recursion. The solution to this modification lies in the partition function call. To make this modification run you call the partition function in a while loop until the value of partition is equal to k. The value of partition is increased until it is the same as k.

**Experimental Results**

**Part #1**

Testing select\_bubble(L,k) with list size [55,11,33] and searching for k = 0

﻿ Output: 11

Runtime: 0.00010995600314345211

Testing select\_bubble(L,k) with list size [55,11,33,66] and searching for k = 1

﻿ Output: 33

Runtime: 7.914500019978732e-05

Testing select\_bubble(L,k) with list size[55,11,33,66,44] nd searching for k = 2

﻿ Output: 44

Runtime: 9.710399899631739e-05

Testing select\_quick(L,k) with list size [55,11,33] and searching for k = 0

﻿ Output: 11

Runtime: 2.079700061585754e-05

Testing select\_quick(L,k) with list size [55,11,33,66] and searching for k = 1

﻿ Output: 33

Runtime: 0.15161673099646578

Testing select\_quick(L,k) with list size [55,11,33,66,44] and searching for k = 2

﻿ Output: 44

Runtime: 4.6111003030091524e-05

Testing select\_modified\_quicksort (L,k) with list size[55,11,33]and searching for k = 0

﻿ Output: 11

Runtime: 1.9874998542945832e-05

Testing select\_ modified\_quicksort L,k) with list size [55,11,33,66] and searching for k = 1

﻿ Output: 33

Runtime: 1.9988001440651715e-05

Testing select\_ modified\_quicksort (L,k) with list size[55,11,33,66,44] and searching for k = 2

﻿ Output: 44

Runtime: 2.1985004423186183e-05

**Part #2**

Testing select\_quickSort\_Stack(L,k) with list size[55,11,33]and searching for k = 0

﻿Output: 11

Runtime: 2.9889997676946223e-05

Testing select\_quickSort\_Stack (L,k) with list size [55,11,33,66] and searching for k = 1

﻿ Output: 55

Runtime: 1.532700116513297e-05

Testing select\_quickSort\_Stack (L,k) with list size [55,11,33,66,44] and searching for k = 2

﻿ Output: 44

Runtime: 2.8262998966965824e-05

Orange – select\_quick

Grey – select\_bubble

Blue – select\_modified\_quick

**Conclusion**

What I learned from this lab is that many of the functions that are implemented in sorting algorithms can be manipulated to portray similar results to what would be the output if you just used the sorting algorithm. The most difficult part was understanding how to implement a stack instead of recursion. The breakthrough in this difficulty was in seeing the similarities between a stack and recursion instead of focusing on the differences. This lab ultimately showed me how important sorting algorithms are and how they can be manipulated to our liking to produce the result we are looking for.

**Appendix**

**﻿**

**﻿from timeit import default\_timer as timer**

**def quicksort\_stack(L,start,end):**

**#creates a temporary stack**

**size = end - start + 1**

**stack = [0] \* (size)**

**#initiate stack**

**top = -1**

**#push start and end to stack**

**top = top + 1**

**stack[top] = start**

**top = top + 1**

**stack[top] = end**

**#while stack is not empty keep popping**

**while top >= 0:**

**#pop start and end**

**end = stack[top]**

**top = top - 1**

**start = stack[top]**

**top = top - 1**

**p = partition( L, start, end-1 )**

**#pushing left of stack**

**if p-1 > start:**

**top = top + 1**

**stack[top] = start**

**top = top + 1**

**stack[top] = p - 1**

**#pushing right of stack**

**if p+1 < end:**

**top = top + 1**

**stack[top] = p + 1**

**top = top + 1**

**stack[top] = end**

**def select\_quickSort\_Stack(L,k):**

**quicksort\_stack(L,0,len(L))**

**return L[k]**

**def partition (L,l,h):**

**x = L[h]**

**i = (l-1)**

**for j in range (l,h):**

**if (L[j]) <= x:**

**i += 1**

**swap(L,i,j)**

**swap(L,i+1,h)**

**return (i + 1)**

**def swap (L,i,j):**

**x = L[i]**

**L[i] = L[j]**

**L[j] = x**

**def quickSort(L,low,high):**

**if low < high:**

**p = partition(L,low,high)**

**quickSort(L, low, p-1)**

**quickSort(L, p+1, high)**

**def select\_quick(L,k):**

**quickSort(L,0,len(L) - 1)**

**return L[k]**

**def bubbleSort (L,n):**

**if n == 1:**

**return L**

**for i in range (n-1):**

**if L[i] > L[i+1]:**

**temp = L[i]**

**L[i] = L[i+1]**

**L[i+1] = temp**

**bubbleSort(L,n-1)**

**def select\_bubble(L,k):**

**bubbleSort(L,len(L))**

**return L[k]**

**def select\_modified\_quickSort(L,low,high,k):**

**if low<high:**

**p = partition(L,low,high)**

**if p == k:**

**return L[p]**

**elif k < p:**

**return select\_modified\_quickSort(L, low, p-1,k)**

**else:**

**return select\_modified\_quickSort(L, p+1, high,k)**

**def select\_modifiedWhile(L,low,high,k):**

**p = partition(L,low,high)**

**while k == p:**

**partition(L,low, p)**

**k = int(input('Enter index of K to find the Kth smallest element in the list: '))**

**start = timer()**

**L1 = [55,11,33,66,44]**

**print (select\_bubble(L1,k))**

**end = timer()**

**print('Runtime:',str(end-start))**

**start = timer()**

**L2 = [55,11,33,66,44]**

**print (select\_quick(L2,k))**

**end = timer()**

**print('Runtime:',str(end-start))**

**start = timer()**

**L3 = [55,11,33,66,44]**

**print (select\_modified\_quickSort(L3,0,len(L3)-1,k))**

**end = timer()**

**print('Runtime:',str(end-start))**

**#start = timer()**

**#L5 = [55,11,33,66,44]**

**#print (select\_modifiedWhile(L5,0,len(L5) - 1,k))**

**#end = timer()**

**#print('Runtime:',str(end-start))**

**start = timer()**

**L4 = [55,11,33,66,44]**

**print (select\_quickSort\_Stack(L4,k))**

**end = timer()**

**print('Runtime:',str(end-start))**

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class