CS2302 Data Structures

Lab Report No. 2

Author: Bryan Ramos
Due: September 20th, 2019
Professor: Olac Fuentes
TA: Anindita Nath

Introduction

For this lab, we were asked to engineer several sorting algorithms that also consist of finding the kth smallest element in a list. If k = 0, the first smallest element in the list is returned by the algorithms, if k = 1 the algorithms will return the second smallest element in the list, and so on. The main objective of this lab was to build familiarity with sorting algorithms like bubble sort and quick sort, how some sorting algorithms can be implemented with recursion, with a stack and with only a while loop and finally understanding their running time.

Proposed Solution Design & Implementation

For the sorting algorithms to function a list is required. During my first attempts at solving the requirements for the lab I would ask the user for the length of the desired list and then prompt them to enter integer values until the length of the list was filled. However, when I began writing my lab report I noticed that was no longer ideal, what if the user enters a list length of 100 or 1000? Furthermore, testing experiment results would be difficult this way. Instead, I revised my design to be able to test list length sizes big or small. With my revision, my program asks for a desired length of a list. The program then populates the list with integer values between 0 and 100.

After a list is created, the user is prompted to enter a value for k - the smallest element in the list to return:

> *If k = 0, the first smallest element in the list is returned by the algorithms, if k = 1 the algorithms will return the second smallest element in the list, and so on.*

At this part of the main function, invalid input from the user is accounted for: say the user enter characters that are not numerical values (letters or symbols). Secondly, the program checks that the k value is within bounds of list, that is, the value is greater or equal to 0 and less than the length of the list.

Part 1:

The main function calls bubble sort passing two parameters: the list and value for k. Bubble sort was the simplest algorithm to implement, it works by repeatedly swapping adjacent elements if they are in the wrong place. The value at position k is returned. After implementing this algorithm, I noticed that whenever a list is sorted, so is the order of smallest elements is too in ascending order. That is, the first value is the smallest, second value is the second smallest and so on.

Next, I implemented a function for quicksort that accepts the list, a low value being 0 and the high value being the last index of the list (length of list minus). The low and high values are used for partitioning. Quicksort is a divide and conquer algorithm. It picks an element in the list as a pivot and partitions the given array around the picked pivot. Within my quicksort method, I check if the high value is greater than the low, if so, I set the partitioning index by calling the partition function and passing it the following parameters: the list, low value, high value. My partition function takes the last element as a pivot, the

pivot is then placed in the correct position and all values smaller than the pivot are placed to the left of the pivot and all values greater than the pivot as placed to the right of the pivot.

Smaller ← Pivot → Larger

Once my partition index is set my quick sort function makes two recursive calls to itself to sort the elements before the partition and after partition. Once the list is sorted, the value at position k is returned, just like in bubble sort.

Next, I implemented a function for modified quicksort that only makes one recursive function call: the first recursive function from the original quicksort algorithm I implemented. After the recursive call, the low value will be set to the partitioning index plus one. As with the bubble sort and regular quicksort functions, the value at position k is returned.

After any one of the above algorithms is called, the value at position k is returned. The value at position k is printed with the proper suffix: “st”, “nd”, “rd”, “th”. Finally, the time it took to sort and find the value at position K is printed to the user, this also helped me keep track of running time for each algorithm for my lab results.

Part 2:

Part 2 required the following: quicksort implementation with a stack and quicksort using only a while loop. I thought it was helpful when Dr. Fuentes showed us the example of implementing Towers of Hanoi using a stack. When going with a stack implementation for quicksort, the recursive calls must be added to the stack backwards. Using the Towers of Hanoi stack implementation example as a guide I created a class for the Stack record. In my function, I create a stack, again using my implementation for stack record. Next, while the length of the stack is greater than zero, I first get the last element from the stack and then partition to get the pivot value and then I append my left and right sub lists. The value at position k is returned.

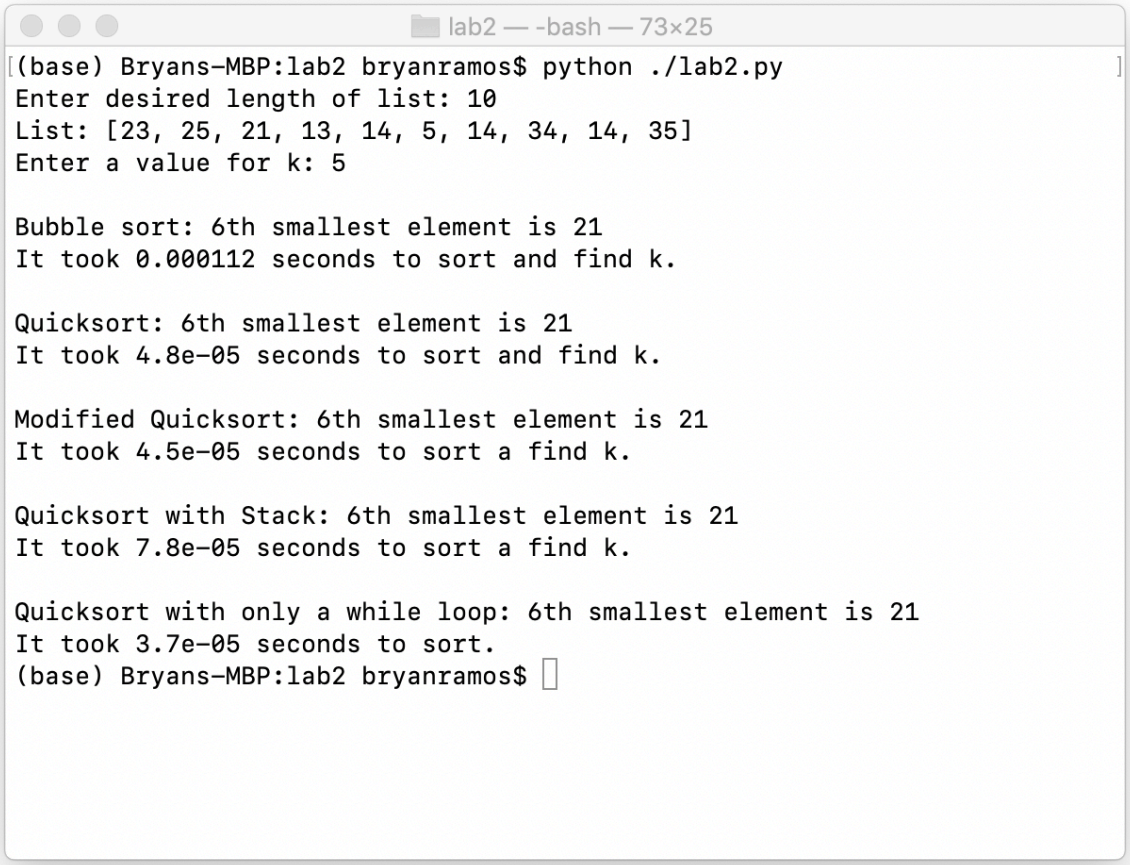
Next, I implemented quick sort by only using a while loop. First, my function must partition to get the pivot value. Next, while the pivot is not the user inputted value for k, I iterate through a while loop. Next I check for two conditions: if k is greater than the pivot, I partition through values from pivot + 1 to the highest, otherwise, I partition through values from the low to one minus the pivot value. Once my loop stops iterating through, I return the value at the pivot, since to terminate the loop the pivot must be the same value as k.

Like with part 1, the value at position k is returned to the main function, printed, the proper suffix is added, and the running time is also printed.

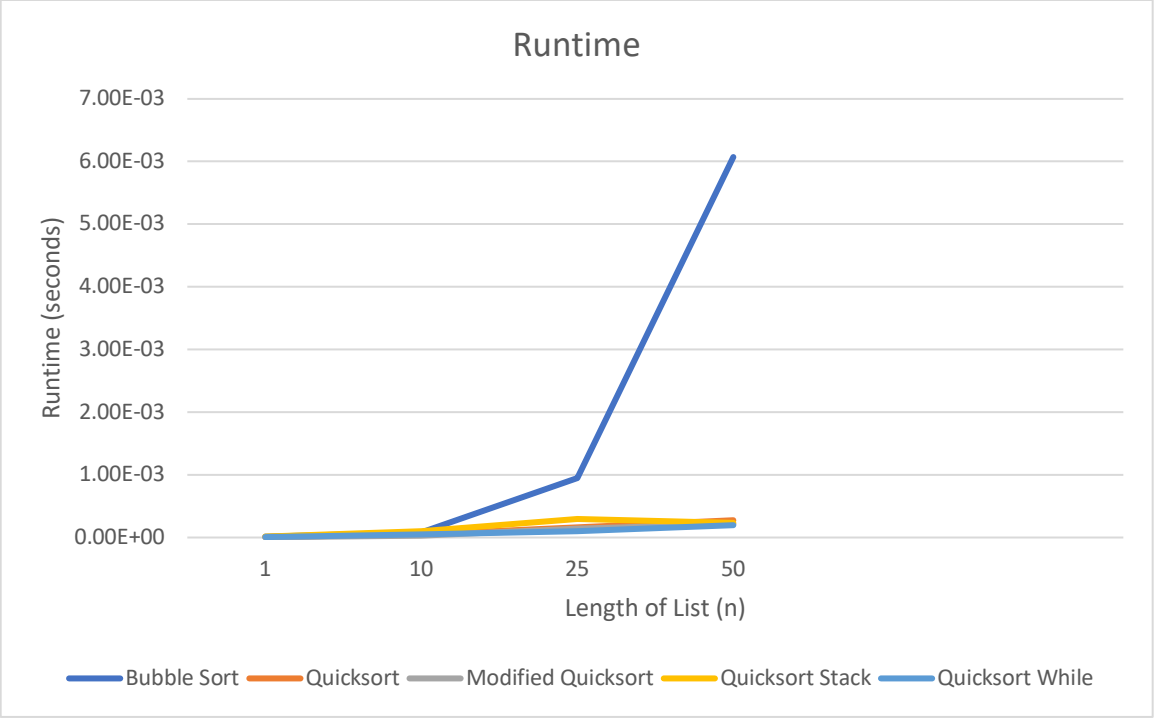
Experimental Results

To ensure the sorting algorithms were working as expected, they must all return the same value at position k.

Here is a sample execution with a list of length 10 and a k value of 5.



As mentioned in my proposed design solution and implementation, initially I asked the user to enter integer values for the list in addition to the desired length of the list. When I got to this step of my report, I realized that was not ideal and instead resorted to populating the list with random values from 0 to 50. Here is how the algorithms performed with lists of different lengths.

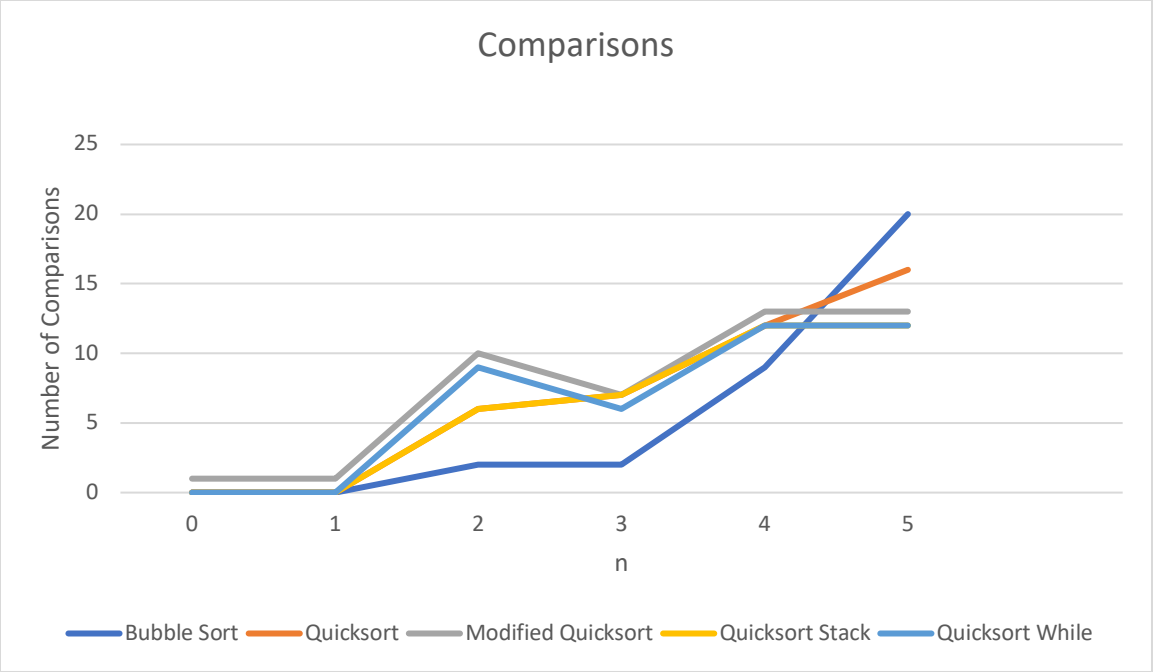


Runtimes and Graph Interpretation:

The length of my lists for testing stop at 50 - after 50 I get a maximum recursion depth error.

On average, bubble sort performed the worst out of all of the sorting algorithms especially as the size of the list grew. The worst-case running time for bubble sort is $O(n^2)$. It sorts through the elements in the list, swapping those in the wrong order. As the length of a list grows, bubble sort will become slower and slower. Notice how even if the length of the list goes to 25 bubble sort takes significantly a lot more than time than the other sorting algorithms. • The worst-case runtime for a quick sort algorithm is O(n2) based on previous work and experiences. The best or average case runtime for quicksort is O(n * logn). Modified quicksort appears to have a runtime of O(n).

n	Bubble Sort
2	4
4	16
8	64
16	256
64	4096



Notice how the number of comparisons for bubble sort is on track to keep increasing faster than the others. It makes sense as with running time, bubble sort also performed the worst.

Conclusion

This lab was very interesting. Implementing sorting algorithms is important as data in the real world must be sorted and organized, of course data on a much bigger scale at times so it's important to understand which algorithm is best for what situation. This lab also helped me understand what factors affect running time the most, especially in recursion, tying in what we learned about running times and how to calculate big o notation in the lecture. I knew stacks could be used for a multitude of things, but I did not know quicksort could be implemented too so it's really nice to know how to implement quicksort with stack, going to try it on other programming languages now. It's nice to know there's more than one way to fix a problem!

Appendix

```
'''
    Course: CS2302 Data Structures Fall 2019
    Author: Bryan Ramos [88760110]
    Assignment: Lab 2 Sorting
    Instructor: Dr. Olac Fuentes
    TA: Anindita Nath
```

```
    Last Modified: September 20th 2019

    Purpose: Implementing sorting algorithms like bubble sort and quick
sort to understand how they function, implement quick sort using a stack
and implement modified with no stack or recursion. Understand the
differences in their run times.
'''


import time as t

import random as rand

import sys


# ---------- PART 1 ----------


# bubble sort

def select_bubble(L, k):

    for i, num in enumerate(L):

        try:

            if L [i+1] < num:

                L[i] = L[i+1]

                L[i+1] = num

                select_bubble(L, k)

        except IndexError:

            pass

    return L[k]


# take the last element as a pivot, the pivot element is placed in the

# correct position, and places all smaller (smaller than pivot) values

# to the left of pivot and all greater elements to the right

# smaller <- pivot -> larger

def partition(L, low, high):

    # smaller element index

    i = low - 1
```

```
    pivot = L[high]

    for j in range(low, high):
        # current element is smaller than the pivot
        if L[j] < pivot:
            # increment index of smaller element
            i = i + 1
            L[i], L[j] = L[j], L[i]
    L[i + 1], L[high] = L[high], L[i + 1]
    return (i + 1)

def quicksort(L, low, high):
    if low < high:
        # partioning index, L[p] is now at right place
        partitionIndex = partition(L, low, high)

        # separately sort elements before partition and after partition
        quicksort(L, low, partitionIndex - 1)
        quicksort(L, partitionIndex + 1, high)

def select_quick(L, k):
    quicksort(L, 0, len(L) - 1)
    return L[k]

def quicksort_modified(L, low, high):
    while low < high:
        partitionIndex = partition(L, low, high)

        # separately sort element before partition and after partition
        quicksort_modified(L, low, partitionIndex - 1)
        low = partitionIndex + 1
```

```
def select_modified_quick(L, k):
    quicksort_modified(L, 0, len(L) - 1)
    return L[k]

# ---------- PART 2 ----------

class stackRecord:
    def __init__(self, L, low, high):
        self.L = L
        self.low = low
        self.high = high

def select_quick_stack(L, low, high, k):
    stack = [stackRecord(L, low, high)]

    while len(stack) > 0:
        # get the last element
        stackElement = stack.pop(-1)
        if stackElement.low < stackElement.high:
            # partitionIndex -> get pivot value
            partitionIndex = partition(stackElement.L, stackElement.low,
stackElement.high)

            # add in left and right sublists
            stack.append(stackRecord(stackElement.L, stackElement.low,
partitionIndex - 1))
            stack.append(stackRecord(stackElement.L, partitionIndex + 1,
stackElement.high))
    return L[k]

def select_quick_while(L, low, high, k):
```

```
    pivot = partition(L, low, high)

    while pivot != k:
        if k > pivot:
            pivot = partition(L, pivot + 1, high)
        elif k < pivot:
            pivot = partition(L, low, pivot - 1)
    return L[pivot]

# ---------- MAIN ----------

def printSuffix(k):
    # suffix to add after the kth element number
    if k[-1] == "1":
        print("st", end="")
    elif k[-1] == "2":
        print("nd", end="")
    elif k[-1] == "3":
        print("rd", end="")
    else:
        print("th", end="")

if __name__ == '__main__':

    sys.setrecursionlimit(1500)

    # empty list to add random int values to
    numsList = []

    # ask for len of list and build a list from user input int values
    length = int(input("Enter desired length of list: "))
```

```
    # add random integer values to list from 0-50
    for i in range(length):
        numsList.append(rand.randint(0, 50))

    # lists to use in each sorting operation will have same values as
numsList
    list1 = numsList
    list2 = numsList
    list3 = numsList
    list4 = numsList
    list5 = numsList

    print("List:", numsList)

    # ask user to the value for k -> must be an integer value
    k = 0
    while (True):
        try:
            k = int(input("Enter a value for k: "))
            break
        except:
            print("\nInvalid! Please enter an integer number.\n")

    # sort using either: bubble sort, quick sort or modified quicksort
    # find kth smallest element in the sorted list

    # k should be less than the len of the list & cannot be less than 0
    if (k >= len(numsList)) or k < 0:
        print("\nError: Value for k out of bounds. List size:",
len(numsList), "\n")
    else:
```

```
        print("")

        # bubble sort
        start = t.time()
        numAtIndex = select_bubble(list1, k)
        stop = t.time()
        print("Bubble sort:", k+1, end="")
        printSuffix(str(k + 1))
        print(" smallest element is", numAtIndex)
        print("It took", round(stop-start, 6), "seconds to sort and find
k.")

        print("")

        # quicksort
        start = t.time()
        numAtIndex = select_quick(list2, k)
        stop = t.time()
        print("Quicksort:", k+1, end="")
        printSuffix(str(k + 1))
        print(" smallest element is", numAtIndex)
        print("It took", round(stop-start, 6), "seconds to sort and find
k.")

        print("")

        # modified quicksort
        start = t.time()
        numAtIndex = select_modified_quick(list3, k)
        stop = t.time()
        print("Modified Quicksort:", k+1, end="")
        printSuffix(str(k + 1))
```

```
        print(" smallest element is", numAtIndex)
        print("It took", round(stop-start, 6), "seconds to sort a find k.")

        print("")

        # quicksort with stack implem.
        start = t.time()
        numAtIndex = select_quick_stack(list4, 0, len(list4) - 1, k)
        stop = t.time()
        print("Quicksort with Stack:", k+1, end="")
        printSuffix(str(k + 1))
        print(" smallest element is", numAtIndex)
        print("It took", round(stop-start, 6), "seconds to sort a find k.")

        print("")

        # quicksort with a while loop
        start = t.time()
        numAtIndex = select_quick_while(list5, 0, len(list5) - 1, k)
        stop = t.time()
        print("Quicksort with only a while loop:", k+1, end="")
        printSuffix(str(k + 1))
        print(" smallest element is", numAtIndex)
        print("It took", round(stop-start, 6), "seconds to sort.")
```

I, Bryan Ramos, 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.