CS 2302 Data Structures

Fall 2019

Lab Report #2

Due: September 20th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

Introduction

For this lab we were tasked with creating a program that would sort a given list and return the kth element within the list. The program would use bubble sort, quicksort, a modified quicksort, a quicksort that utilizes stacks instead of recursion, and a modified quicksort that only uses a while loop. For this lab it is essential that we are familiar with recursion as well as tracing a method to make sure the list is being altered according to how we wish to sort it.

Proposed Solution Design and Implementation

Operation #1:

First off, I started my program by creating a main method at the bottom of the program to create the list to sort as well as call upon each program made select_bubble(L, k), select_quick(L, k), and select_modified_quick(L, k). The first method just sorts the method using bubble sort, the second method uses two recursion to completely sort the list, and the third uses only one recursion call to sort the list until the kth element is found.

- **select_bubble(L, k):** When this method gets called it passes the list into another method labeled, "**bubbleSort(L)**." First the list is checked to see if it is empty or of length one in which there is no need to sort and the method returns the list as is. If it has a length of two or more then the program uses two for-loops to go through the list putting the largest value at the end of the list until it is completely sorted. After it is finished sorting the list is returned to the original method where it checks if the list is empty as well as if the list has a kth value. If the list is empty then the method prints, "There are no elements in the list," and return the value -999. If the value of k is less then zero the method prints out, "There is no element in position k." Then it prints out and returns the closest value which would be the first value at position zero. Likewise, if the value of k is greater then the length of the list the program prints out, "There is no element in position k." then prints out and returns the last element of the list. Otherwise if there is kth element in the list it is printed and returned.

- **select_quick(L, k):** When this method gets called it too passes the list onto another method called, "**quicksort(L, lPos, rPos):**," to be sorted. When it is first called the lPos value is zero, and the rPos value is the len(L-1). First the method makes sure the value of lPos is greater than or equal to the value of rPos in which case there is no need to sort and the list is returned. If lPos is less-then rPos then the method continues and passes the variables into another method to sort the list using the first element called, "**partitionQS(L, lPos, rPos):**," to get the mid position. This method collects the value of the first element as a pivot and traverse the list from the lPos until it reaches a value that is greater than the pivot and rPos goes backwards until it reaches a value that is less then the pivot. It then checks to see it the two variables passed

each other in which the while-loop would end and the first element of the list and the element at rPos would swap places. Otherwise the program would continue swapping elements until lPos is greater than or equal to rPos. After the partition is over the rPos is returned as it is the new mid position. The method quicksort is then called two more times with the positions for the left of the list not including the mPos(**quicksort(L, lPos, mPos-1)**). As well as the positions for the right of the list not including the mPos(**quicksort(L, mPos+1, rPos**)). At this point the program repeats until it reaches the base case at which point the list is sorted and the program ends. After the list is sorted the original method checks if the list is empty as well as if the list has a kth value. If empty the method prints, "There are no elements in the list," and returns the value -999. If the value of k is less than zero the method prints out, "There is no element in position k." Then it prints out and returns the closest value which would be the first value at position zero. Likewise, if the value of k is greater than the length of the list the program prints out, "There is no element in position k." then prints out and returns the last element of the list. Otherwise if there is kth element in the list it is printed and returned.

- **select_modified_quick(L, k):** Similar to the method **select_quick(L, k)**, this method uses two more methods to sort the list **modifiedQS(L, lPos, rPos)** and **partitionMQS(L, lPos, rPos)**. But the main difference is after the partition method returns the new mid position it goes through 2 cases. If the new mid position is equal to k then the program returns to the original method select_modified_quick(L, k) as the element that is being searched for is already in the correct position even if the entire list is not sorted. If mPos is greater then the value of k then we only make one recursive call(**modifiedQS(L, lPos, mPos-1**)). Then we would only have to sort through the left side because the kth element is on that side. If neither case executes then that means that the value of mPos is less then the value of k in which case the recursive call being made would sort through the right side of the list because the kth value is in there(**modifiedQS(L, mPos+1, rPos**)). Even if the kth value is less then zero or greater then the length of the list method will sort through the entire side and will be addressed in the original method once the list is completely sorted. Again, after the list is sorted the original method checks if the list is empty as well as if the list has a kth value. If empty the method prints, "There are no elements in the list," and returns the value -999. If the value of k is less than zero the method prints out, "There is no element in position k." Then it prints out and returns the closest value which would be the first value at position zero. Likewise, if the value of k is greater than the length of the list the program prints out, "There is no element in position k." then prints out and returns the last element of the list. Otherwise if there is kth element in the list it is printed and returned.

Operation #2:

In this part of the lab we are tasked with modifying the last two methods created in the first operation (select_quick(L, k) and select_modified_quick(L, k)). For select_quick we need to implement the use of a stack to sort the list instead of recursion, and for select_modified_quick I need to rewrite the method to run with a while-loop instead of stacks or recursion.

- **stackQS(L, k):** Just like the method select_quick(L, k) this method uses two more methods (**stackQuicksort_nr(L, lPos, rPos)** and **partitionSQS(L, lPos, rPos)**). First the list and initial values are passes into stackQuicksort_nr(L, 0, len(L)-1). Then the values are pushed into a constructer method called, **classQSQ(object)** to create an object to start the stack. The constructer gives the object three elements (L, lPos, and rPos). After the List has been started a while-loop is created and will continue to run until the stack is empty. Within the loop the stack is popped, and the object is passed to the partitionSQS method to obtain the mid position. Similar to the original quicksort the midpoint is then used to create and add two more positions onto the stack (**stack.append(classSQS(temp.L, temp.lPos, h -1))** and **stack.append(classSQS(temp.L, h+1, temp.rPos))).** The first adds the left side of the middle position to the stack and the second adds the right side. The process will repeated until the list is sorted and the stack is empty. Then after the list is sorted the original method checks if the list is empty as well as if the list has a kth value. If empty the method prints, “There are no elements in the list,” and returns the value -999. If the value of k is less than zero the method prints out, “There is no element in position k.” Then it prints out and returns the closest value which would be the first value at position zero. Likewise, if the value of k is greater than the length of the list the program prints out, “There is no element in position k.” then prints out and returns the last element of the list. Otherwise if there is kth element in the list it is printed and returned.

- **while_modified_quick(L, k):** Keeping the same thought process I used for the select_modified_quick(L, k) method, I passed the list and other variables into another method called **modifiedWQS(L, 0, len(L)-1, k)**. The method then creates a pivot variable and sets the value to the first element of the list. Then using two nested while-loops the program finds the mid position and either returns mPos if it equals k, sorts the left until mPos equals k, or sorts the right until mPos equals k. What ever direction it sorts in once mPos is equal to k then the while-loops exit, and the list is returned to the original method. Then after the list is sorted the original method checks if the list is empty as well as if the list has a kth value. If empty the method prints, “There are no elements in the list,” and returns the value -999. If the value of k is less than zero the method prints out, “There is no element in position k.” Then it prints out and returns the closest value which would be the first value at position zero. Likewise, if the value of k is greater than the length of the list the program prints out, “There is no element in position k.” then prints out and returns the last element of the list. Otherwise if there is kth element in the list it is printed and returned.

Experimental Results:

For each method I tested the same four tests. First if the list was empty, second if the value of k was less then zero, third if the value of k was greater then the length of the list, and finally if the value of k is within the length of the list.

Operation #1:

- **select_bubble(L, k):**

Case 1 (Empty list):
k: 0
returned value: -999

```
---------------------------- Bubble Sort ----------------------------
Sorted List: []
There are no elemnts in the list.
```

Case 2 (Negative k):
k: -1
returned value: 0

```
---------------------------- Bubble Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position -1.
The element in position 0 is, 0.
```

Case 3 (k greater than list length):
k: 10
returned value: 9

```
---------------------------- Bubble Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position 10.
The last element of the list is in position 9 and is, 9.
```

Case 4 (k within list length):
k: 5
returned value: 5

```
---------------------------- Bubble Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The element at position 5 is, 5.
```

- **select_quick(L, k):**

Case 1 (Empty list):
k: 0
returned value: -999

```
---------------------------- Quick Sort ----------------------------
Sorted List: []
There are no elemnts in the list.
```

Case 2 (Negative k):
k: -1
returned value: 0

```
---------------------------- Quick Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position -1.
The element in position 0 is, 0.
```

Case 3 (k greater than list length):
k: 10
returned value: 9

```
---------------------------- Quick Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position 10.
The last element of the list is in position 9 and is, 9.
```

Case 4 (k within list length):
k: 5
returned value: 5

```
---------------------------- Quick Sort ----------------------------
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The element at position 5 is, 5.
```

- **select_modified_quick(L, k):**

Case 1 (Empty list):
k: 0
returned value: -999

```
----------------------- Modified Quick Sort -----------------------
Sorted List: []
There are no elemnts in the list.
```

Case 2 (Negative k):
k: -1
returned value: 0

```
----------------------- Modified Quick Sort -----------------------
Sorted List: [0, 1, 3, 4, 2, 5, 7, 8, 6, 9]
There is no element in position -1.
The element in position 0 is, 0.
```

Case 3 (k greater than list length):
k: 10
returned value: 9

```
----------------------- Modified Quick Sort -----------------------
Sorted List: [5, 3, 6, 7, 2, 1, 4, 8, 0, 9]
There is no element in position 10.
The last element of the list is in position 9 and is, 9.
```

Case 4 (k within list length):
k: 5
returned value: 5

```
----------------------- Modified Quick Sort -----------------------
Sorted List: [1, 3, 0, 4, 2, 5, 7, 8, 6, 9]
The element at position 5 is, 5.
```

Operation #2:

- **stackQS(L, k):**

```
------------------------ Stack Quick Sort -----------------------
```

Case 1 (Empty list):
k: 0
returned value: -999

```
Sorted List: []
There are no elemnts in the list.
```

Case 2 (Negative k): ---------- Stack Quick Sort ----------
k: -1
returned value: 0
```
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position -1.
The element in position 0 is, 0.
```

Case 3 (k greater than list length):
k: 10 ---------- Stack Quick Sort ----------
returned value:
```
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
There is no element in position 10.
The last element of the list is in position 9 and is, 9.
```

Case 4 (k within list length):
k: 5 ---------- Stack Quick Sort ----------
returned value: 5
```
Sorted List: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
The element at position 5 is, 5.
```

- **while_modified_quick(L, k):**

Case 1 (Empty list): ---------- While Loop Quick Sort ----------
k: 0
returned value: -999
```
Sorted List: []
There are no elemnts in the list.
```

Case 2 (Negative k): ---------- While Loop Quick Sort ----------
k: -1
returned value: 9
```
Sorted List: [9, 1, 3, 4, 2, 5, 7, 8, 6, 0]
There is no element in position -1.
The element in position 0 is, 9.
```

Case 3 (k greater than list length):
k: 10 ---------- While Loop Quick Sort ----------
returned value:
```
Sorted List: [9, 3, 6, 7, 2, 1, 4, 8, 0, 5]
There is no element in position 10.
The last element of the list is in position 9 and is, 5.
```

Case 4 (k within list length):
k: 5 ---------- While Loop Quick Sort ----------
returned value: 5
```
Sorted List: [1, 3, 0, 4, 2, 5, 7, 8, 6, 9]
The element at position 5 is, 5.
```

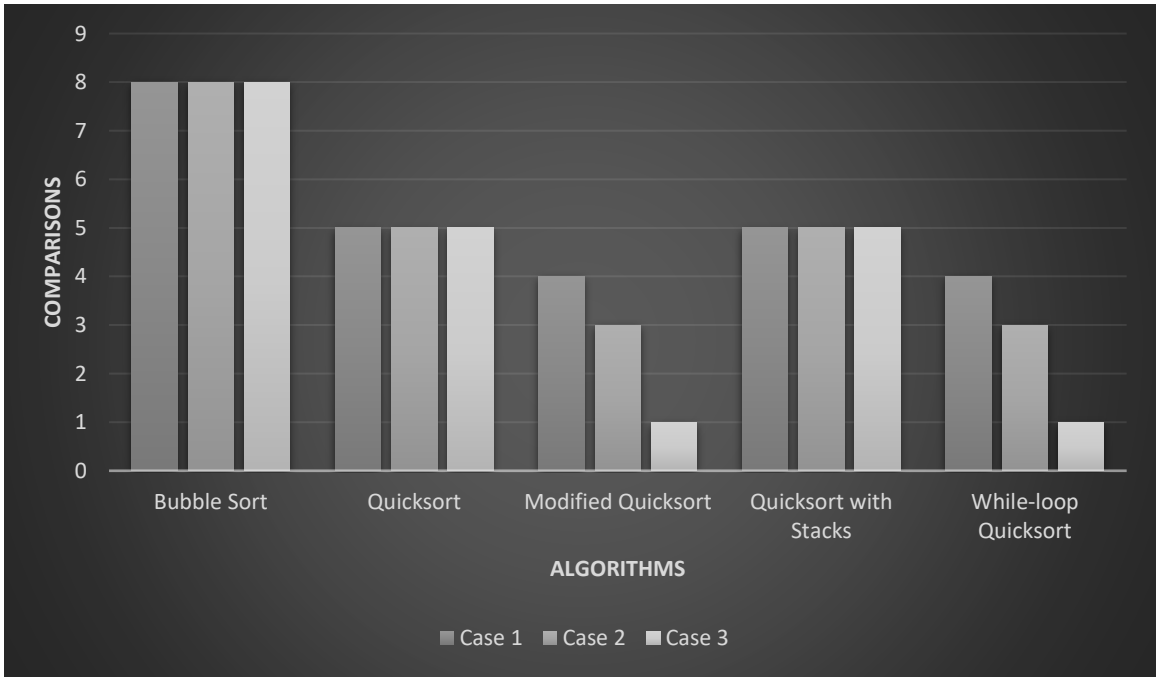
Data

Sorting Algorithm	Running time with respect to **n**
Bubble Sort	$O(n^2)$
Quicksort	O(n*log(n))
Modified Quicksort	O(n*log(n))
Quicksort with Stacks	O(n*log(n))
While-loop Quicksort	O(n*log(n))

Number of Comparisons

Case List = ([9, 3, 6, 7, 2, 1, 4, 8, 0, 5])

Sorting Algorithm	Case 1: (L, 2)	Case 2: (L, 7)	Case 3: (L, 9)
Bubble Sort	8	8	8
Quicksort	5	5	5
Modified Quicksort	4	3	1
Quicksort with Stacks	5	5	5
While-loop Quicksort	4	3	1



- The data show that bubble sort, quicksort, and the quicksort with stacks all have a liner sorting pattern as they sort the whole list of elements no matter what position is being located. As for the modified quicksort and the while-loop modified quicksort can finish in as little as one comparison depending on the value being searched for as well as the order of the list.

Conclusion

In conclusion through completing this lab I was able to better understand how algorithms like quicksort can manipulate list to sort in less comparisons then other algorithms like bubble sort. As well as changing certain parameters can further increase those sorting methods such as in the modified quicksort as it only sorted the list until it knew the kth position was in the correct place.

Appendix

```
'''
    Cs2302 Data Structures
    Issac Rivas (80604101)
    Lab 2
    Dr.Fuentes

'''
#Part 1
#1.Bubble Sort
#-------------------------------------------------------------------------------
def select_bubble(L,k):
    bubbleSort(L)
    print("Sorted List:", L)

    #If list is Empty
    if len(L) == 0:
        print("There are no elemnts in the list.")
        return -999

    #If k is a negative value
    elif k < 0:
        print("There is no element in position", str(k)+".")
        print("The first element in position", str(0)+" is,", str(L[0])+".")
        return L[0]

    #If k is greater then the length of the list
    elif len(L)-1 < k:
        print("There is no element in position", str(k)+".")
        print("The last element of the list is in position", str(len(L)-1)+" and is,", str(L[len(L)-1])+".")
        return L[len(L)-1]

    #If k is within the length of the list
    else:
        print("The element at position", str(k)+" is,", str(L[k])+".")

    return L[k]


def bubbleSort(L):
    #If the list is empty there is no need to sort
    if L is None:
        return L
```

```
    #If the List is of length 1 there is no need to sort
    elif len(L) == 1:
        return L

    #Sorts through the length of the list placing the largest value at the end
    else:
        for x in range(len(L)):
            for y in range(0, (len(L)-x)-1):
                if L[y] > L[y+1]:
                    temp = L[y]
                    L[y] = L[y+1]
                    L[y+1] = temp
    return L



#2.Quicksort
#-------------------------------------------------------------------------------
def select_quick(L,k):
    quicksort(L, 0, (len(L)-1))
    print("Sorted List:", L)

    #If List is empty
    if len(L) == 0:
        print("There are no elemnts in the list.")
        return -999

    #If k is a negative value
    elif k < 0:
        print("There is no element in position", str(k)+".")
        print("The first element in position", str(0)+" is,", str(L[0])+".")
        return L[0]

    #If k is greater then the length of the list
    elif len(L)-1 < k:
        print("There is no element in position", str(k)+".")
        print("The last element of the list is in position", str(len(L)-1)+" and is,", str(L[len(L)-1])+".")
        return L[len(L)-1]

    #If k is within the length of the list
    else:
        print("The element at position", str(k)+" is,", str(L[k])+".")

    return L[k]


def quicksort(L, lPos, rPos):

    #If the left position is greater or equal to the right there is no need to sort
    if lPos >= rPos:
        return

    #Gets the mid point of the array which is where the pivot swaps to
    mPos = partitionQS(L, lPos, rPos)

    #Sorts the two opposite sides of the pivot
    quicksort(L, lPos, mPos-1)
    quicksort(L, mPos+1, rPos)


def partitionQS(L, lPos, rPos):
    pivot = L[lPos]
    leftPos = lPos + 1
    x =  leftPos
    y = rPos
    loop = True

    #While true it will swap values that are above and below the pivot untill
    #leftPos is > y or untill x is < rPos
```

```
    while loop:
        while L[leftPos] <= pivot and leftPos < y:
            leftPos += 1
        while pivot < L[rPos] and x < rPos:
            rPos -= 1
        if rPos <= leftPos:
            loop = False
        else:
            temp = L[leftPos]
            L[leftPos] = L[rPos]
            L[rPos] = temp

    #Swaps the pivot at the leftmost point with the position of the rPos
    temp = L[lPos]
    L[lPos] = L[rPos]
    L[rPos] = temp
    return rPos



#3.Modifed Quicksort
#-------------------------------------------------------------------------------
def select_modified_quick(L,k):
    modifiedQS(L, 0, (len(L)-1), k)
    print("Sorted List:", L)

    #If list is empty
    if len(L) == 0:
        print("There are no elemnts in the list.")
        return -999

    #If k is a negative value
    elif k < 0:
        print("There is no element in position", str(k)+".")
        print("The first element in position", str(0)+" is,", str(L[0])+".")
        return L[0]

    #If k is greater then the length of the list
    elif len(L)-1 < k:
        print("There is no element in position", str(k)+".")
        print("The last element of the list is in position", str(len(L)-1)+" and is,", str(L[len(L)-1])+".")
        return L[len(L)-1]

    #If k is within the length of the list
    else:
        print("The element at position", str(k)+" is,", str(L[k])+".")

    return L[k]


def modifiedQS(L, lPos, rPos, k):

    #If the left position is greater or equal to the right there is no need to sort
    if lPos >= rPos:
        return

    #Gets the mid point of the array which is where the pivot swaps to
    mPos = partitionMQS(L, lPos, rPos, k)

    #If the position looking for is the mPos then no need to sort more
    if k == mPos:
        return mPos

    #If position k is less then the midpoint then you just need to sort the left side
    elif k < mPos:
        modifiedQS(L, lPos, mPos-1, k)

    #If position k is greater then the midpoint then you just need to sort the right side
    else:
        modifiedQS(L, mPos+1, rPos, k)
```

```
183
184
185 def partitionMQS(L, lPos, rPos, k):
186     pivot = L[lPos]
187     leftPos = lPos + 1
188     x =  leftPos
189     y = rPos
190     loop = True
191
192     #While true it will swap values that are above and below the pivot untill
193     #leftPos is > y or untill x is < rPos
194     while loop:
195         while L[leftPos] <= pivot and leftPos < y:
196             leftPos += 1
197         while pivot < L[rPos] and x < rPos:
198             rPos -= 1
199         if rPos <= leftPos:
200             loop = False
201         else:
202             temp = L[leftPos]
203             L[leftPos] = L[rPos]
204             L[rPos] = temp
205
206     #Swaps the pivot at the leftmost point with the position of the rPos
207     temp = L[lPos]
208     L[lPos] = L[rPos]
209     L[rPos] = temp
210     return rPos
211
212
213
214 #Part 2
215 #1. Quicksort with Stacks
216 #-------------------------------------------------------------------------------
217 def stackQS(L, k):
218     stackQuicksort_nr(L, 0, (len(L)-1))
219     print("Sorted List:", L)
220
221     #If List is empty
222     if len(L) == 0:
223         print("There are no elemnts in the list.")
224         return -999
225
226     #If k is a negative value
227     elif k < 0:
228         print("There is no element in position", str(k)+".")
229         print("The first element in position", str(0) +" is,", str(L[0])+".")
230         return L[0]
231
232     #If k is greater then the length of the list
233     elif len(L)-1 < k:
234         print("There is no element in position", str(k)+".")
235         print("The last element of the list is in position", str(len(L)-1)+" and is,", str(L[len(L)-1])+".")
236         return L[len(L)-1]
237
238     #If k is within the length of the list
239     else:
240         print("The element at position", str(k)+" is,", str(L[k])+".")
241
242     return L[k]
243
244
```

```
class classSQS(object):
    # Constructor
    def __init__(self, L, lPos, rPos):
        self.L = L
        self.lPos = lPos
        self.rPos = rPos


#Creates a stack that keeps adding on untill the list is sorted
def stackQuicksort_nr(L, lPos, rPos):

    #Creates a stack and adds first left and right position
    stack = [classSQS(L, lPos, rPos)]

    #Runs untill the stack is empty
    while len(stack)>0:
        temp = stack.pop(-1)
        if temp.lPos < temp.rPos:
            #Gets the mPos of the list
            h = partitionSQS(temp.L, temp.lPos, temp.rPos)

            #Adds the left and right of the list to the stack
            stack.append(classSQS(temp.L, temp.lPos, h - 1))
            stack.append(classSQS(temp.L, h + 1, temp.rPos))


def partitionSQS(L, lPos, rPos):
    pivot = L[lPos]
    leftPos = lPos + 1
    x =  leftPos
    y = rPos
    loop = True

    #While true it will swap values that are above and below the pivot untill
    #leftPos is > y or untill x is < rPos
    while loop:
        while L[leftPos] <= pivot and leftPos < y:
            leftPos += 1
        while pivot < L[rPos] and x < rPos:
            rPos -= 1
        if rPos <= leftPos:
            loop = False
        else:
            temp = L[leftPos]
            L[leftPos] = L[rPos]
            L[rPos] = temp

    #Swaps the pivot at the leftmost point with the position of the rPos
    temp = L[lPos]
    L[lPos] = L[rPos]
    L[rPos] = temp
    return rPos



#2.Modified Quicksort with while loop
#-------------------------------------------------------------------------------
def while_modified_quick(L,k):
    if len(L) != 0 and len(L)-1 >= k:
        modifiedWQS(L, 0, (len(L)-1), k)
    print("Sorted List:", L)

    #If list is empty
```

```
    if len(L) == 0:
        print("There are no elemnts in the list.")
        return -999

    #If k is a negative value
    elif k < 0:
        print("There is no element in position", str(k)+".")
        print("The first element in position", str(0)+" is,", str(L[0])+".")
        return L[0]

    #If k is greater then the length of the list
    elif len(L)-1 < k:
        print("There is no element in position", str(k)+".")
        print("The last element of the list is in position", str(len(L)-1)+" and is,", str(L[len(L)-1])+".")
        return L[len(L)-1]

    #If k is within the length of the list
    else:
        print("The element at position", str(k)+" is,", str(L[k])+".")

    return L[k]


def modifiedWQS(L, lPos, rPos, k):
    pivot = L[lPos]
    loop = True
    loop2 = True

    #While mPos dose not equal k
    while loop:
        leftPos = lPos + 1
        x = leftPos
        y = rPos

        #Sets the new mPos
        while loop2:
            while L[leftPos] <= pivot and leftPos < y:
                leftPos += 1
            while pivot < L[rPos] and x < rPos:
                rPos -= 1
            if rPos <= leftPos:
                loop2 = False
            else:
                temp = L[leftPos]
                L[leftPos] = L[rPos]
                L[rPos] = temp

        #Swaps the pivot and the mPos
        temp = L[lPos]
        L[lPos] = L[rPos]
        L[rPos] = temp
        mPos = rPos

        #If mPos equals k then no need to sort more
        if mPos == k:
            loop = False

        #If mPos is less then k you only have to sort the right side
        elif mPos < k:
            pivot = L[mPos + 1]
            lPos = mPos + 1
            rPos = y
```

```
            loop2 = True

        #If mPos is more then k you only need to sort the left side
        elif mPos > k:
            pivot = L[0]
            rPos = mPos - 1
            loop2 = True

    return L[k]



################################################################################
#Main
#-------------------------------------------------------------------------------
if __name__ == '__main__':

    L1 = list([9,3,6,7,2,1,4,8,0,5])
    L2 = list([9,3,6,7,2,1,4,8,0,5])
    L3 = list([9,3,6,7,2,1,4,8,0,5])
    L4 = list([9,3,6,7,2,1,4,8,0,5])
    L5 = list([9,3,6,7,2,1,4,8,0,5])
    print()
    print("------------------------------ Part 1 ------------------------------")
    print()
    print("Unsorted list:", L1)
    print("--------------------------- Bubble Sort ---------------------------")
    print()
    select_bubble(L1,0)
    print()
    print()
    print("Unsorted list:", L2)
    print("--------------------------- Quick Sort ----------------------------")
    print()
    select_quick(L2,0)
    print()
    print()
    print("Unsorted list:", L3)
    print("----------------------- Modified Quick Sort -----------------------")
    print()
    select_modified_quick(L3,0)
    print()
    print()
    print()
    print("------------------------------ Part 2 ------------------------------")
    print()
     print("Unsorted list:", L4)
     print("------------------------ Stack Quick Sort -------------------------")
     print()
     stackQS(L4,0)
     print()
     print()
     print("Unsorted list:", L5)
     print("------------------ While-Loop Modified Quick Sort ------------------")
     print()
     while_modified_quick(L5,0)
     print()
```

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