**Lab Report 2: Sorting and Activation Records**

**I. Introduction**

My program should find the kth smallest element in a list. The user decides what kth smallest number he wants. I should be able to take this input and test different methods to get the result. There are five methods that I should make which are:

1. select bubble(L,k). Using bubble sort  
2. select quick(L,k). Using quick sort  
3. select modified quick(L,k). Usung quick sort but does not finish the sorting in order to bring back the kth element. It only sorts where the k will be, which means only making one recursive call instead of two.  
4. stack quick(L, k). uses stacks on the quicksort instead of recursion.  
5. while modified quick(L,k). Uses only a while loop to sort. No recursion or stacks.

**II. Proposed solution design and implementation**

I made the shell of my program first. So, I made a function to create a random list depending on the size the user inputs, then I made one for the display that would happen every time a test was made. After that I made the menu so that the user can choose which method they want to test. After my user interface was done, I began working on each function.

1. Select Bubble:  
This sort compares every pair of numbers and switches them if they are not in the correct order. This was the easiest to make, as it was just thinking of completing the sorting with for loops and finally returning the element at k.

2. Select Quick:  
This one required more thinking. I decided to use an auxiliary function so that the first call to the method only requires the list and k, not high and low. I struggled getting the partition right, because I used just one method. I had to put conditions to move the indices and swap values that gave the correct input to the recursive case.

3. Select Modified Quick:   
This modified quick sort used an extra condition that the previous did not. If k was higher than the pivot, only the lower half was sent to recursion, if it was higher, the higher half was sent, and if it was the same, then it was returned. This decreases the number of passes that the method will do, because after all, the purpose is to return the kth smallest, not to sort everything.

4. Stack Quick:  
At first, I thought I would be storing the lower values in a stack, and the higher values in another. Then I realized I had to use the stacks for the activation records that recursion does. The stack stores the low and high indices that need to be sorted. So, at first 2 values are stores, and then 4, and so on, until it stops storing and removes, then stores again, and then it only removes, so the while loop finishes, and the list is sorted.

5. While Modified Quick:  
What I did here is realize that a pivot is in its correct order once it goes through partition. So, I changed the place that the pivot was going to be taken from (s) linearly. And while the pivot did not equal k, it went on. To make it quicker, if the pivot was less than k, I made s = pivot+1 instead of the next value. This was a very efficient use of while and quicksort.

**III. Experimental results**

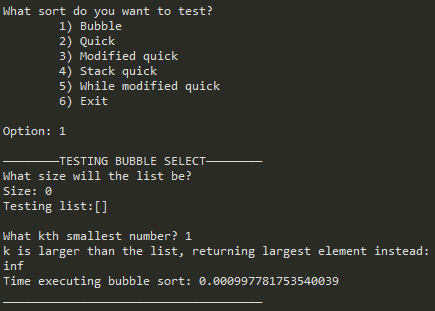
The fact that I made a function so that the user input the length of the list helped me make testing more efficient. I tested with sizes from 0 to 20, and even 7000 (not that there is a limit).

I made it so that when the list was 0, it returned “inf.” And if the user chose a k that was out of bounds, the program did not exit, but instead just took the largest value.

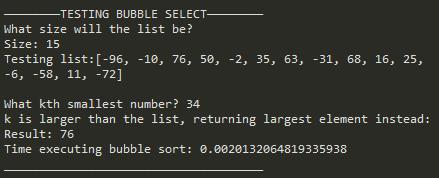
I looked for the smallest values, the largest, the middle, and any random to see if the function worked. A trick I did for checking is that the list that you are looking at is displayed, so if I chose 4 as k, then there should be 4 smaller numbers that the output, and if I chose 0, there shouldn’t be any smaller numbers.

Testing Bubble Sort:

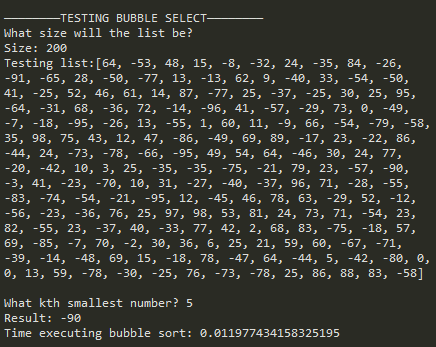
Empty list:



List of 15:

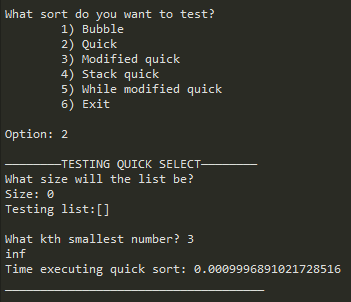


List of 200:

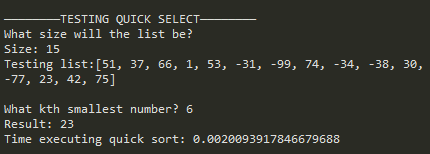


Testing Quick Sort:

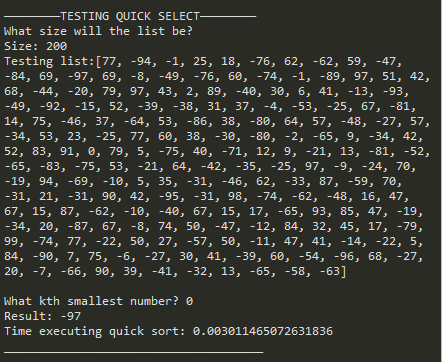
Empty List:



List of 15:

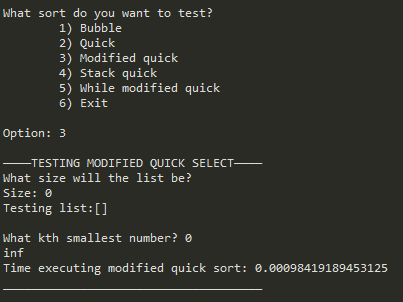


List of 200:

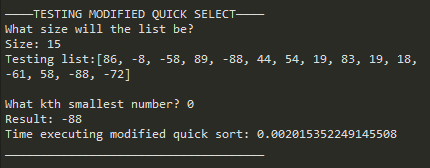


Testing Modified Quick:

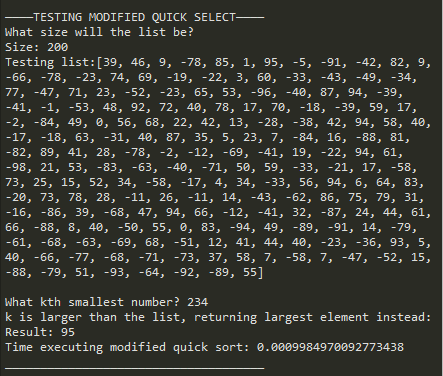
Empty List:



List of 15:

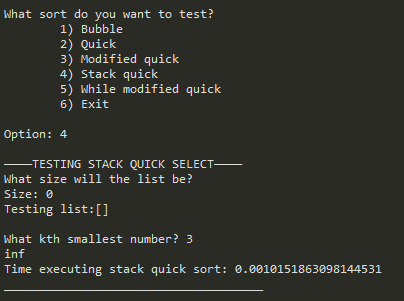


List of 200:

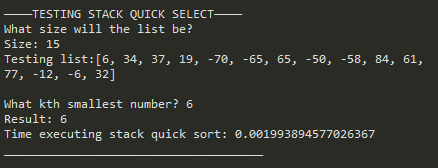


Testing Stack Quick:

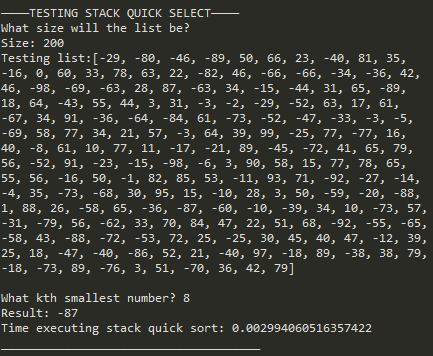
Empty list:



List of 15:

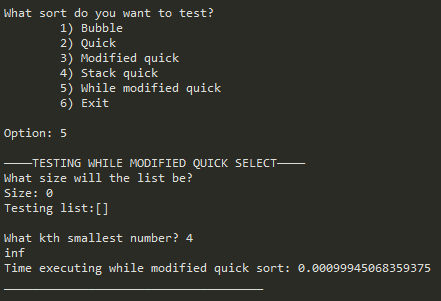


List of 200:

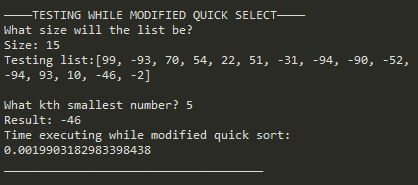


Testing While Modified Quick:

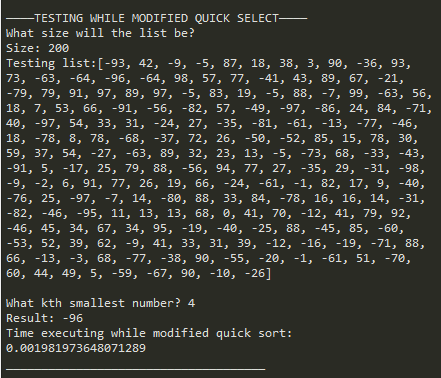
Empty List:



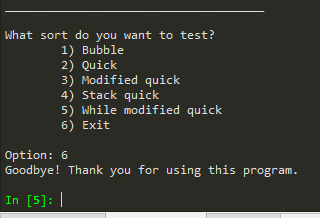
List of 15:



List of 200:



Exit:



**Tables**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Input 0 | Input 15 | Input 200 | O(n) |
| Bubble | 0.0009 | 0.002 | 0.0119 | O(n2) |
| Quick | 0.0009 | 0.002 | 0.003 | O(nlogn) |
| Modified Quick | 0.0009 | 0.002 | 0.0009 | O(logn) |
| Stack Quick | 0.0009 | 0.0019 | 0.0029 | O(nlogn) |
| While M Quick | 0.0009 | 0.0019 | 0.0019 | O(logn) |

**IV. Conclusions**

I learned how to think about stacks differently. This helped me understand activation records on recursion. There are many ways to get to the same result, and many ways that I wouldn’t have thought about solving the problem, but it is very important because after all, the time and efficiency of a program matters. I learned to consider loops, recursion, and stacks when I solve a problem. I also learned that optimizing is very useful. The purpose of this lab was not sorting, it was finding the kth smallest element, so focusing on that helped make the functions quicker. It just means adding conditions and removing steps.

**V. Appendix – Source code**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 2: Sorting

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 9/19/2019

Program description:This program finds the kth smallest element in a list. If k==0, select(L,k) returns the

smallest element in list L, if k==1 it returns the second smallest, and so on. This lets the user decide

which method to use to fin the kth element. The user inputs the size of the list and the program randomly

generates one.

1. select bubble(L,k). Uses bubble sort

2. select quick(L,k). Uses quick sort

3. select modified quick(L,k). Uses quick sort but does not finish the sorting in order to bring back the

kth element. It only sorts where the k will be.

4. stack quick(L, k). uses stacks on the quicksort instead of recursion.

5. while modified quick(L,k). Uses only a while loop to sort. No recursion or stacks.

"""

import math

import random

import time

#PART ONE

#Find smallest kth with bubble sort

def select\_bubble(L,k):

if k>=len(L): #changing kth number as it is out of bounds

print('k is larger than the list, returning largest element instead: ')

k = len(L)-1

if len(L) == 0: #nothing in the list to return

return math.inf

#bubble sort compares every pair of numbers and rearranges them

for i in range(len(L)):

for j in range(0, len(L)-i-1):

if (L[j]>L[j+1]):

L[j], L[j+1] = L[j+1], L[j]

print('Result: ', end ='')

return L[k]

#With quick sort

def select\_quick(L,k):

if len(L) == 0:

return math.inf

if k>=len(L): #changing kth number as it is out of bounds

print('k is larger than the list, returning largest element instead: ')

k = len(L)-1

quicksort(L,0,len(L)-1)

print ('Result: ', end='')

return L[k]

def quicksort(L,l,h):

# base case

if l >= h:

return

# range

low = l

high = h

#last element in range

pivot = L[h]

#new height

h -= 1

#loop to make swaps

while l < h:

#move low index

while L[l]<pivot and l != h:

l += 1

#move right index

while L[h]>=pivot and l != h:

h -= 1

#swap

L[l], L[h] = L[h], L[l]

#next indices

if (l != h):

l +=1

h -=1

#replace pivot

if L[l]<pivot:

l += 1

L[high], L[l] = L[l], L[high]

#partition

quicksort(L,low,l-1)

quicksort(L,l+1, high)

#with modified quick sort

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

if len(L) == 0:

return math.inf

if k>=len(L): #changing kth number as it is out of bounds

print('k is larger than the list, returning largest element instead: ')

k = len(L)-1

modified\_partition(L,0,len(L)-1,k)

print ('Result: ', end='')

return L[k]

def modified\_partition(L,l,h,k):

# base case

if l >= h:

return

# range

low = l

high = h

#last element in range

pivot = L[h]

#new height

h -= 1

#loop to make swaps

while l < h:

#move low index

while L[l]<pivot and l != h:

l += 1

#move right index

while L[h]>=pivot and l != h:

h -= 1

#swap

L[l], L[h] = L[h], L[l]

#next indices

if (l != h):

l +=1

h -=1

#replace pivot

if L[l]<pivot:

l += 1

L[high], L[l] = L[l], L[high]

#partition

if l == k: #k is already in its place so it does not need to do anything else

return

elif l > k: #sort lower half if k is in the lower half

modified\_partition(L,low,l-1,k)

else: #or sort higher half if k is in the higher half

modified\_partition(L,l+1, high,k)

#PART TWO

#quicksort with stacks

def stack\_quicksort(L,k):

if len(L) == 0:

return math.inf

if k>=len(L): #changing kth number as it is out of bounds

print('k is larger than the list, returning largest element instead: ')

k = len(L)-1

qs2(L,0,len(L)-1) #call to actual stack quicksort

print ('Result: ', end='')

return L[k]

def qs2(L, l, h):

#Auxiliary stack will store 'activation records'

auxL = []

#push initial values of l and h to stack (start and end)

auxL.append(l)

auxL.append(h)

#While there are no more instructions on aux stack

while (len(auxL) > 0):

#Pop h and l

h = auxL.pop()

l = auxL.pop()

#Set pivot element at its correct position in the list

p = part2(L, l, h)

#If there are elements on left side of pivot, then push left side to stack (activation record)

if (p - 1 > l):

auxL.append(l)

auxL.append(p - 1)

#If there are elements on right side of pivot, #then push right side to stack (activation record)

if (p + 1 < h):

auxL.append(p + 1)

auxL.append(h)

return

def part2(L, l, h):

x = L[h]

i = (l - 1);

for j in range(l , h ):

if L[j] <= x:

i = i +1

L[i], L[j] = L[j], L[i]

L[i+1], L[h] = L[h], L[i+1]

return (i+1)

#Quicksort with while no recursion

def while\_modified\_quick(L,k):

if len(L) == 0:

return math.inf

if k>=len(L): #changing kth number as it is out of bounds

print('k is larger than the list, returning largest element instead: ')

k = len(L)-1

print ('Result: ', end='')

return qs3(L,k)

def qs3(L,k):

p = -1 #pivot index

s = 0 #index to make next pivot

while p!=k:

p = part3(L, 0, len(L)-1, s)

if p<k: #goes ahead on list

s = p + 1

else: #goes to the next index

s = s + 1

if s==len(L):

s = 0

return L[k]

def part3(L, l, h, s):

L[s], L[h] = L[h], L[s] #swap s to be the pivot

x = L[h] #last index is pivot

i = (l - 1); #i will be l next, and then keep adding

for j in range(l , h ):

if L[j] <= x:

i = i +1

L[i], L[j] = L[j], L[i] #swap values

L[i+1], L[h] = L[h], L[i+1] #swap values

return (i+1) #returns pivot index

#creates a random list of a size the user inputs

def makeList(size):

L = []

for i in range(size):

L.append(random.randint(-99,99))

return L

#Default print to test a select

def printTest():

print('What size will the list be?', end = '')

size = input('Size: ')

L = makeList(int(size))

print('Testing list:', end = '')

print(L)

return L

#MAIN

stay = True

#Show menu again

while stay:

print('What sort do you want to test?')

print('\t1) Bubble\n\t2) Quick\n\t3) Modified quick\n\t4) Stack quick\n\t5) While modified quick\n\t6) Exit')

whatSort = input('Option: ')#stores option

#bubble select

if whatSort.upper() == 'BUBBLE' or whatSort == '1':

print('\n————————TESTING BUBBLE SELECT————————')

L= printTest()

k = int(input('What kth smallest number? '))

start = time.time()

print(select\_bubble(L,k))

end = time.time()

print('Time executing bubble sort: ' + str(end-start) + '\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n')

#quick select

elif whatSort.upper() == 'QUICK' or whatSort == '2':

print('\n————————TESTING QUICK SELECT————————')

L= printTest()

k = int(input('What kth smallest number? '))

start = time.time()

print(select\_quick(L,k))

end = time.time()

print('Time executing quick sort: ' + str(end-start) + '\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n')

#modified quick select

elif whatSort.upper() == 'MODIFIED QUICK' or whatSort == '3':

print('\n————TESTING MODIFIED QUICK SELECT————')

L= printTest()

k = int(input('What kth smallest number? '))

start = time.time()

print(select\_modified\_quick(L,k))

end = time.time()

print('Time executing modified quick sort: ' + str(end-start) + '\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n')

#stack quick sort

elif whatSort.upper() == 'STACK QUICK' or whatSort == '4':

print('\n————TESTING STACK QUICK SELECT————')

L= printTest()

k = int(input('What kth smallest number? '))

start = time.time()

print(stack\_quicksort(L,k))

end = time.time()

print('Time executing stack quick sort: ' + str(end-start) + '\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n')

#modified quick select

elif whatSort.upper() == 'WHILE MODIFIED QUICK' or whatSort == '5':

print('\n————TESTING WHILE MODIFIED QUICK SELECT————')

L= printTest()

k = int(input('What kth smallest number? '))

start = time.time()

print(while\_modified\_quick(L,k))

end = time.time()

print('Time executing while modified quick sort: ' + str(end-start) + '\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n')

#exit program

elif whatSort.upper() == 'EXIT' or whatSort == '6':

stay = False #to exit the loop

print('Goodbye! Thank you for using this program.')

else:

print('Please enter a number from 1-4 or type one of the options')

stack\_quicksort([1],3)

**VI – Academic Honesty Certification**

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.

x- Elisa Jimenez Todd