Laurence Labayen

80358755

9/21/2019

Lab #2

CS2302 MW 10:30am

Description:

For this lab we were given the task to implement recursive sorting algorithms with both bubble and quick sort. The first part was to recursively implement bubble and quick sort and return the kth element of a given list. Additionally, we were asked to implement a modified version of quick sort the only makes one recursive call to the sublist where the kth element is positioned. Part 2 of the instructions stated that we needed to use what we learned from activation records as a reference. One function was to implement quick sort using a stack instead of recursion and the the other was to implement select_modified_quick using a while loop instead of recursion.

Solution design and Implementation:

select_bubble:

For bubble sort, elements are sorted by swapping adjacent elements to in the correct order. By having a check if the second element is smaller than the first element during an iteration. If that is true, both their positions are swapped in the correct order until the largest element is placed at the end of the list. After the loop is done, select_bubble calls itself without the last and largest element. Finally, it returns the kth element is the sorted list. **Best case for this function is O(n) and worst case is $O(n^2)$**

select_quick:

This function uses a partition function that selects a pivot (last element in this case). Then iteratively checks if the element is less than or equal to the pivot value. If true, current right side is incremented then swapped with the current left element. For the select_quick function, it takes in a list, the beginning of the list, end of list and kth element that is asked by the user. This recursive function stops when start index is more than or equal to end index. Partition function is called for the first time and returns and assigns the pivot location to a variable “pi”. A recursive call to each of the left and right sublists is then called with using the current pivot as the middle (left side being ending at one less than the pivot and right side starting at 1 more than the pivot). Once list is sorted, the element at k is returned. **Best case for this function is O(nlogn)** using the master method, and **worst case is $O(n^2)$** when the sublists are always unbalanced.

select_modified_quick:

In this function, we were asked to implement quicksort but only making one recursive call to the sublist where the kth element is. The partition function is still used and return value is assigned to a variable “pi”. If the kth element is in the same index as the pivot, it returns the pivot value. Otherwise, it will keep searching by calling on the sublist where the kth element is and eventually return the value of the kth element once it gets to the base case. Big O for **this function is O(n)** using the master method.

select_quick_nr:

This function uses stacks instead of recursion to implement quick sort. Using Dr. Fuentes’ approach as a template on the class website, I was able to quickly implement a quicksort algorithm using a stack.

select_quick_while:

This function implements the modified quick sort function described previously using a while loop without using recursion or stacks. Using modified_select_quick's logic, I was able implement an identical function using a while loop. Big-O running time for this function is **O(n)**

Experimental Results:

Using the built-in random module, the user enters the desired size of a random list with elements from 0 to 99.

```
listsize=int(input("Enter list size: "))

k=int(input("Enter kth element: "))

#create a list witht the size entered by user with random elements
list1 = []
for i in range(listsize):
    list1.append(random.randint(0, 99))
```

```
L1 = list1
L2 = list1
L3 = list1
L4 = list1
L5 = list1


#check if user enters invalid input(s)
if k<listsize and k>=0:
    start = time.perf_counter()
    print('\nselect_bubble kth element is: ', select_bubble(L1, len(L1)-1, k))
    end = time.perf_counter()
    print('select_bubble took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_quick kth element is: ', select_quick(L2, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_quick took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_modified_quick kth element is: ', select_modified_quick(L3, 0, len(
    end = time.perf_counter()
    print('select_modified_quick took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_quick_nr kth element is: ', select_quick_nr(L4, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_quick_nr took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_quick_while kth element is: ', select_quick_while(L5, 0, len(L1)-1,
    end = time.perf_counter()
    print('select_quick_while ' + str(round((end - start), 5)) + ' seconds\n')
else:
    print("kth element is more than list size",
          "\nor you entered an invalid number")
```

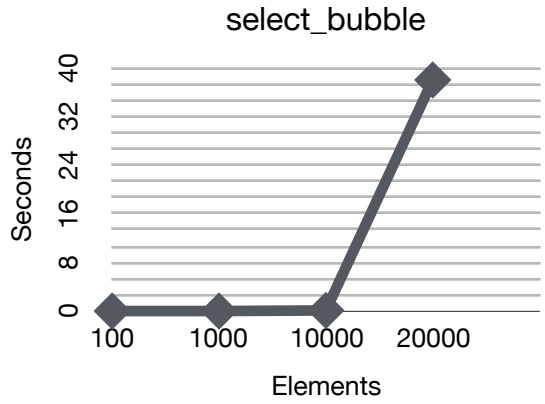
select_bubble:

```
Enter list size: 100

Enter kth element: 50

select_bubble kth element is:  47
select_bubble took 0.00175 seconds
```

```
Enter list size: 1000

Enter kth element: 500

select_bubble kth element is:  49
select_bubble took 0.09023 seconds
```

```
Enter list size: 10000

Enter kth element: 5000

select_bubble kth element is:  49
select_bubble took 9.64982 seconds
```

```
Enter list size: 20000

Enter kth element: 10000

select_bubble kth element is:  49
select_bubble took 38.32157 seconds
```



n	selected kth element	seconds
100	50	0.00175
1000	500	0.09023
10000	5000	9.64982
20000	10000	38.32157

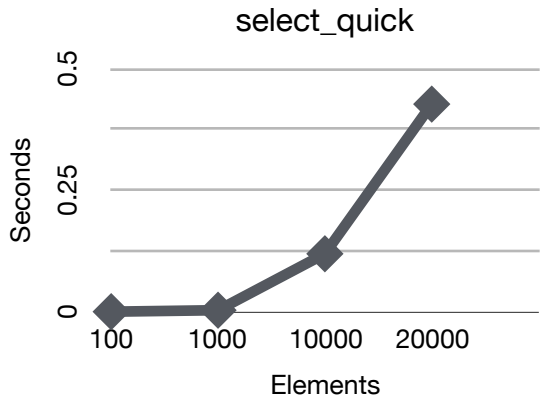
select_quick:

```
Enter list size: 100

Enter kth element: 50

select_quick kth element is:  52
select_quick took 0.00024 seconds
```

```
Enter list size: 1000

Enter kth element: 500

select_quick kth element is:  49
select_quick took 0.00326 seconds
```

```
Enter list size: 10000

Enter kth element: 5000

select_quick kth element is:  51
select_quick took 0.11924 seconds
```

```
Enter list size: 20000

Enter kth element: 10000

select_quick kth element is:  50
select_quick took 0.42689 seconds
```



n	selected kth element	seconds
100	50	0.00024
1000	500	0.00326
10000	5000	0.11924
20000	10000	0.42689

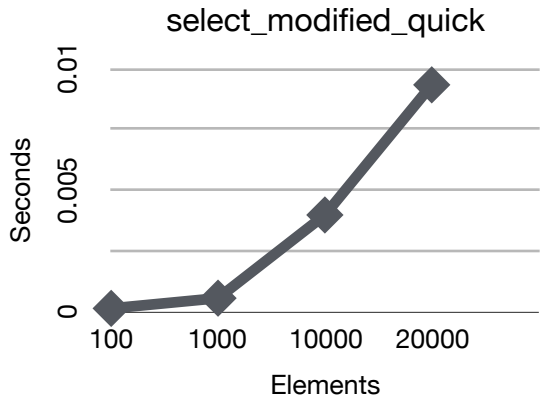
select_modified_quick:

```
Enter list size: 100

Enter kth element: 50

select_modified_quick kth element is:  51
select_modified_quick took 0.00014 seconds
```

```
Enter list size: 1000

Enter kth element: 500

select_modified_quick kth element is:  49
select_modified_quick took 0.00056 seconds
```

```
Enter list size: 10000

Enter kth element: 5000

select_modified_quick kth element is:  49
select_modified_quick took 0.00398 seconds
```

```
Enter list size: 20000

Enter kth element: 10000

select_modified_quick kth element is:  49
select_modified_quick took 0.00933 seconds
```



n	selected kth element	seconds
100	50	0.00014
1000	500	0.00056
10000	5000	0.00398
20000	10000	0.00933

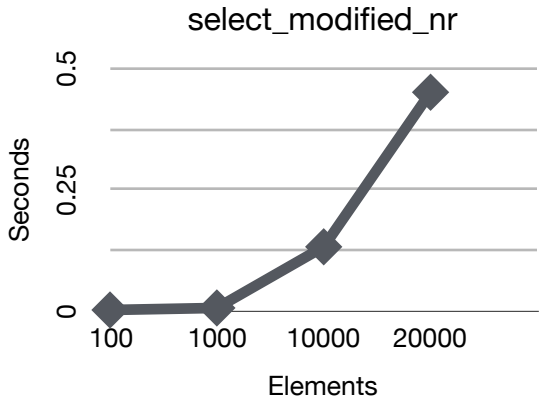
select_quick_nr:

```
Enter list size: 100

Enter kth element: 50

select_quick_nr kth element is:  52
select_quick_nr took 0.00307 seconds
```

```
Enter list size: 1000

Enter kth element: 500

select_quick_nr kth element is:  48
select_quick_nr took 0.00723 seconds
```

```
Enter list size: 10000

Enter kth element: 5000

select_quick_nr kth element is:  49
select_quick_nr took 0.13291 seconds
```

```
Enter list size: 20000

Enter kth element: 10000

select_quick_nr kth element is:  50
select_quick_nr took 0.4506 seconds
```



n	selected kth element	seconds
100	50	0.00307
1000	500	0.00723
10000	5000	0.13291
20000	10000	0.4506

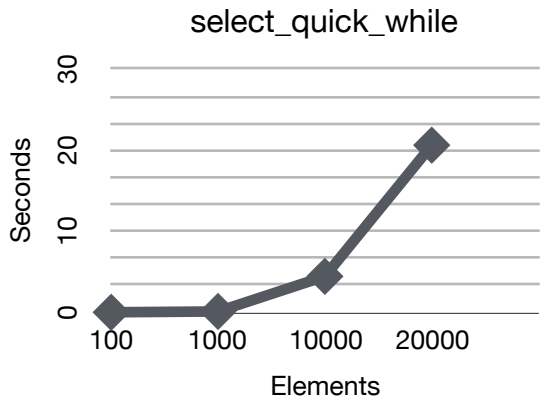
select_quick_while:

```
Enter list size: 100

Enter kth element: 50

select_quick_while kth element is:  53
select_quick_while 0.00014 seconds
```

```
Enter list size: 1000

Enter kth element: 500

select_quick_while kth element is:  51
select_quick_while 0.1023 seconds
```

```
Enter list size: 10000

Enter kth element: 5000

select_quick_while kth element is:  50
select_quick_while 4.43186 seconds
```

```
Enter list size: 20000

Enter kth element: 10000

select_quick_while kth element is:  49
select_quick_while 20.61967 seconds
```



n	selected kth element	seconds
100	50	0.00014
1000	500	0.1023
10000	5000	4.43186
20000	10000	20.61967

Conclusion:

This lab showed me how to implement different sorting algorithms with different requirements. The most difficult one would be implementing quicksort using a stack as it is very different from the normal implementation of quicksort. Also, being able to verify and visually see the analysis helped me see a more concrete view of algorithm running times.

Appendix:

```
import random
import time


#select_bubble recusively sorts adjacent elements and swaps them
#to the correct ascending position in the list
#It takes k as an input and returns the kth element of the list
def select_bubble(L, end, k):
    swapped=False
    #base case if list is sorted
    if end <= 0:
        return L[k]


    for i in range(0, end):
         #compares adjacent elements
        if(L[i + 1] < L[i]):
             #swap elements
            L[i + 1], L[i] = L[i], L[i + 1]
            swapped=True
     #if elements are swapped, recusively call select_bubble
    if swapped==True:
        #recursive call with last element removed
        return select_bubble(L, end - 1, k)
```

```
    return L[k]


#partition selects pivot and compares elements to be placed on the left or
#right side of the pivot
#input: list, start of list, and end of list
#output: index of pivot
def partition(L, start, end):
    i = (start-1)
    pi=L[end] #pivot is selected to be the last element


    for j in range(start, end):
        if L[j] <= pi: #check if element at j is less than or equal to pivot
            i = i+1 #increment i
            L[i],L[j] = L[j],L[i] #swap elements


    L[i+1],L[end] = L[end],L[i+1] #move pivot to correct position in the list


    return (i+1) #return pivot index


#select_quick is a standard recursive quick sort function. Calls partition and
#uses the returned pivot index to create and sort 2 sublists to the left and
#right of pivot
#input: list, start of list, and end of list, kth index of sorted list
#output: kth element of list
def select_quick(L, start, end, k):
    #base case
    if start < end:
        #calling partition to return pivot of partitioned list
        pi = partition(L, start, end)
```

```
        #left of pivot call
        select_quick(L, start, pi-1, k)
        #right of pivot call
        select_quick(L, pi+1, end, k)

    return L[k]
#select_modified_quick is similar to standard quicksort except, it only
#recusively calls the sublist where the kth element is
#input: list, start of list, and end of list, kth index of sorted list
#output: kth element of list
def select_modified_quick(L, start, end, k):
    #calling partition to return pivot of partitioned list
    pi = partition(L, start, end)

    #check if pi is the kth element. end recursion
    if k == pi:
        return L[pi]
    #check if k is in the left sublist
    if k < pi:
        return select_modified_quick(L, start, pi-1, k)
    #otherwise, k must be in right sublist
    else:
        return select_modified_quick(L, pi+1, end, k)

class stackRecord:
    def __init__(self, L, start, end):
        self.L = L
        self.start = start
        self.end = end
```

```
#select_quick_nr is a non recursive quicksort that uses a stack instead of
#recursion to sort a given list
#input: list, start of list, and end of list, kth index of sorted list
#output: kth element of list
def select_quick_nr(L, start, end, k):
    stack = [stackRecord(L, start, end)]
    #loop until stack is at 0
    while(len(stack) > 0):
        #h gets last element
        h = stack.pop(-1)
        if h.start < h.end:
            #pi gets value of pivot
            pi = partition(h.L, h.start, h.end)

            #append left and right sublists
            stack.append(stackRecord(h.L, h.start, pi - 1))
            stack.append(stackRecord(h.L, pi + 1, h.end))
    return L[k]
#select_quick_while uses iteration using a while loop instead of recursion.
#This function does not sort the list completely but only sorts the sublist
#where kth element is
#input: list, start of list, and end of list, kth index of sorted list
#output: kth element of list
def select_quick_while(L, start, end, k):
    #calling partition to return pivot of partitioned list
    pi = partition(L, start, end)
    #iterate through list until k is equal to pivot
    while(pi != k):
        if k < pi:
            pi = partition(L, start, pi -1)
```

```
        elif k > pi:
            pi = partition(L, pi + 1, end)

    return L[pi]

if __name__=="__main__":

    listsize=int(input("Enter list size: "))

    k=int(input("Enter kth element: "))

    #create a list witht the size entered by user with random elements
    list1 = []
    for i in range(listsize):
        list1.append(random.randint(0, 99))
    #use the same list for each function for consistency
    L1 = list1
    L2 = list1
    L3 = list1
    L4 = list1
    L5 = list1


    #check if user enters invalid input(s)
    if k<listsize and k>=0:
        start = time.perf_counter()
        print('\nselect_bubble kth element is: ', select_bubble(L1, len(L1)-1, k))
        end = time.perf_counter()
        print('select_bubble took ' + str(round((end - start), 5)) + ' seconds\n')
```

```
    start = time.perf_counter()
    print('\nselect_quick kth element is: ', select_quick(L2, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_quick took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_modified_quick kth element is: ', select_modified_quick(L3, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_modified_quick took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_quick_nr kth element is: ', select_quick_nr(L4, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_quick_nr took ' + str(round((end - start), 5)) + ' seconds\n')

    start = time.perf_counter()
    print('\nselect_quick_while kth element is: ', select_quick_while(L5, 0, len(L1)-1, k))
    end = time.perf_counter()
    print('select_quick_while ' + str(round((end - start), 5)) + ' seconds\n')
else:
    print("kth element is more than list size",
        "\nor you entered an invalid number")
```

Academic Honesty Statement:

“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.”
-Laurence Labayen