LAB REPORT #2

Sergio Ortiz
September 20, 2019

Professor – Olac Fuentes
TA – Anindita Nath
CS 2302 Data Structures – Fall 2019

Introduction

For this lab, we were asked to implement several algorithms that would select the k^{th} smallest value in a list. For example, if k = 0, then the algorithms would return the smallest value in the list. The first implementation of this asked us to us bubble sort to sort the list and then simply return the element at index k. The next implementation was nearly identical, but it asked us to use quicksort. The third and final implementation was a bit more interesting. It called for us to use a new algorithm, one that would make a single recursive quicksort call, and based off the partition it created and the pivot it selected, would determine which side of the list it would search for depending on where the value of k landed. Once k was equal to the pivot, the algorithm would return this, and we would be done.

For the second part of the lab we were asked two things, each based off the concept of activation records. The first task consisted of implementing the naturally recursive algorithm quicksort non-recursively, using tasks. The second and final task consisted of implementing the third algorithm mentioned in the previous paragraph merely by using a while loop, no recursion or stacks.

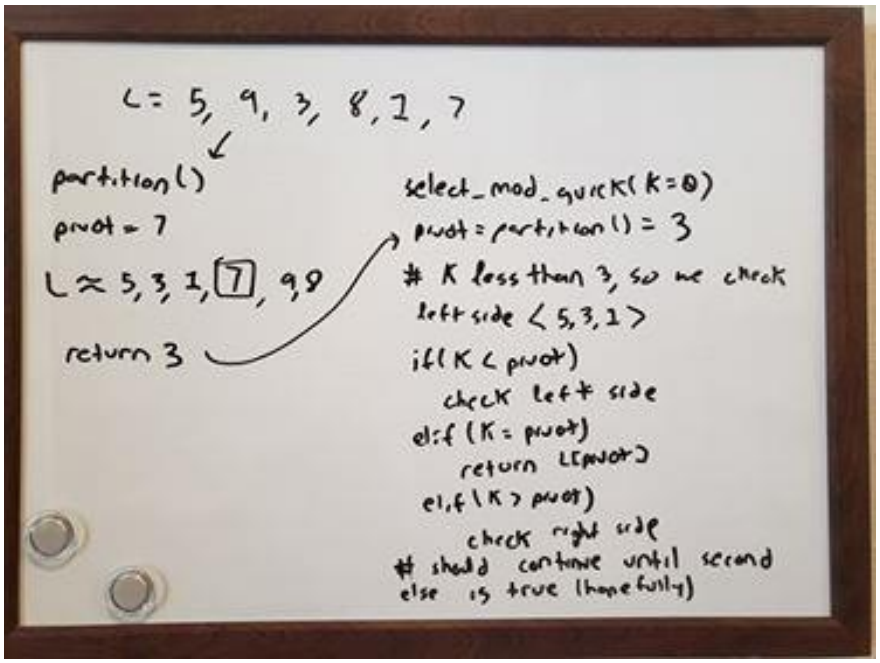
Proposed Solution Design and Implementation

Part #1:

When I began implementing part one, I focused on a bubble sort algorithm. Luckily, this algorithm is quite simple, and its implementation is almost trivial. I have heard some argue that the main function of bubble sort is in academia to introduce students to the idea of sorting. Nonetheless, I implemented this with a pair of nested loops, the outer one controlling the range of comparisons (i.e. the last element will be in its place after one iteration, so we should not check it, the last two after two iterations, and so on) and the inner one doing comparisons between each element and its next one, swapping them if necessary. After having this, the select_bubble() function was simple, merely calling my implementation of bubble sort and then return the element at index k.

For the next algorithm, I began working on an implementation of the quicksort algorithm. Now, I have implemented this algorithm multiple different times, both for academia and for personal research, so I decided to simply copy my implementation over. The only hurdle was that my previous implementation was in java, but the transition was quite seamless. After having this, select_quick() consisted of calling quick sort and returning the element at index k.

The final algorithm was the most interesting and fun, select_modified_quick(). I began my implementation on the whiteboard, scribbling out different scenarios and seeing how they would react.



The pseudocode seemed quite simple, so I proceeded to implement it line by line. The method would need some index parameters, to know where to check. To check only the left side, I called the function again, but this time with the end index being pivot – 1. Similarly, to check only the right side, I called the function again but with the start index being pivot + 1.

Part #2:

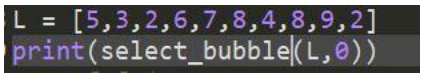
To be completely honest with the reader, I did not have the slightest clue how to implement quicksort using a stack. I understand stacks fine, but as much as I pored over the idea, my mind could not wrap around the notion of using a stack to implement a recursive function. Thankfully, professor Fuentes provided us with some code where he showed the seamless transition from the recursive implementation of the Towers of Hanoi algorithm to the non – recursive one. From here, I decided to copy this code line by line and then tailored it to the needs of the quicksort algorithm. Thank you, professor Fuentes!

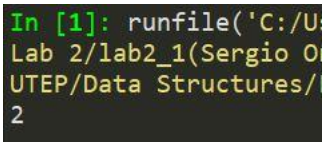
Due to the fact that the second requirement went after the incredibly hard task of implementing quicksort using stacks, I expected it to be excruciatingly hard, even harder than the quicksort one. Therefore, I went into the implementation process with great fear. To my great fortune however, implementing the modified quick select using only a while loop resulted to be quite simple. I began by calling the partition method and assigning its return value to a variable called ‘pivot’. From here, I created a while loop that would iterate as long as pivot was not equal to k. Inside the loop, I copied the three if-else statements from the original implementation, but rather than recursively call the method again, I simply called the partition method and assigned the new value to pivot. This ended up doing the trick.

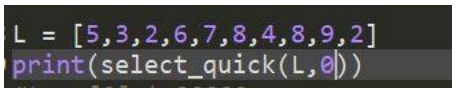
Experimental Results

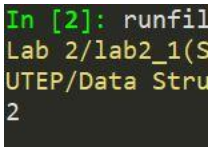
Part #1:

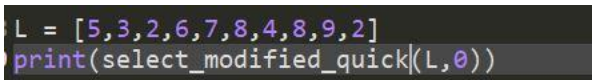
To begin experimenting on part one, first I made sure that all the selectors returned the same output for the same data. I entered the list [5,3,2,6,7,8,4,8,9,2] and checked what the smallest element was.

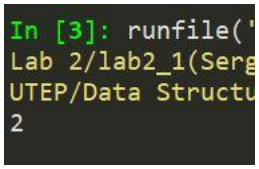
- Select_bubble()
  - 



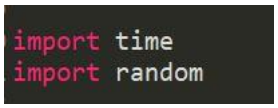
- Select_quick()
  - 



- Select_modified_quick()
  - 



After this I began analyzing each selector separately. First off, the select_bubble(). The runtime of this is O (n^2)since it merely calls bubble sort and then does constant work to return the element at position k. Bubble sort's runtime is O(n^2) because it has a pair of nested loops, each with runtime n). Then, I began running the algorithm with sample data. A quick note on this is that I had to write some code that is not in my submission as it is not part of what the lab asks for. Nonetheless, to do this I first imported the time and random classes.



After this, I created a function that would fill the given list with random integers between 0 and 100, to limit the amount of variables that my experiment has to merely the different selectors. The reason I say this is because I had previously done an experiment where I would see how different sorters reacted to different data and quicksort would take exponentially longer than basic sorters like bubble sort in larger data. After some research I realized that the reason for this was that my arrays were filled with an open range of random ints, and the nature of the random algorithm would return either really large positive numbers or really small negative numbers. When these were selected as pivots, they were essentially moving most of the array to the other side of it during the partition method and attacking the worst case of quicksort. Anyways, I wanted to avoid any unpredicted variables like this, so I kept the random function to a closed range between 0 and 100.

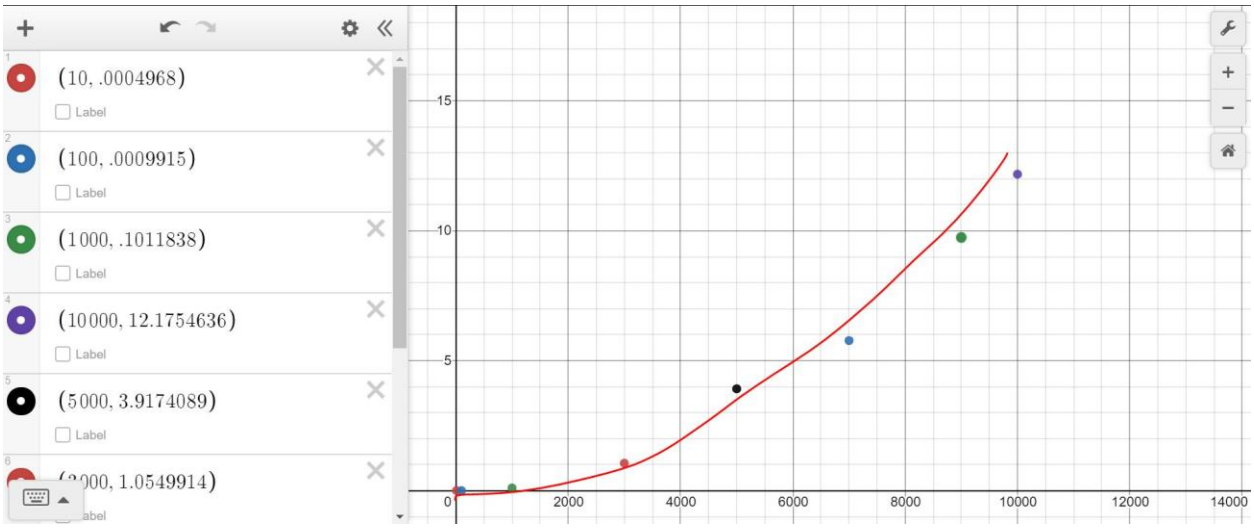
```
def random_numbers(L):
    for i in range(len(L)):
        L[i] = random.randint(0,100)
```

Then, I created a quick snippet of code that would create a list of fixed size and fill it with all zeroes, later calling the random_numbers() to fill it up with random numbers. After this, I began the timer in nanoseconds, since it would round the time down to zero if I used the traditional seconds. Then, I made a print statement that would print the result of calling select_bubble(L,0) and ended the timer. Then, I printed the results.

```
L = [0] * 90000
random_numbers(L)
start = time.time_ns()
print(select_modified_quick(L,0))
end = time.time_ns()
total_time = end - start
print(total_time)
```

Here are the results of the select_bubble with different sized data.

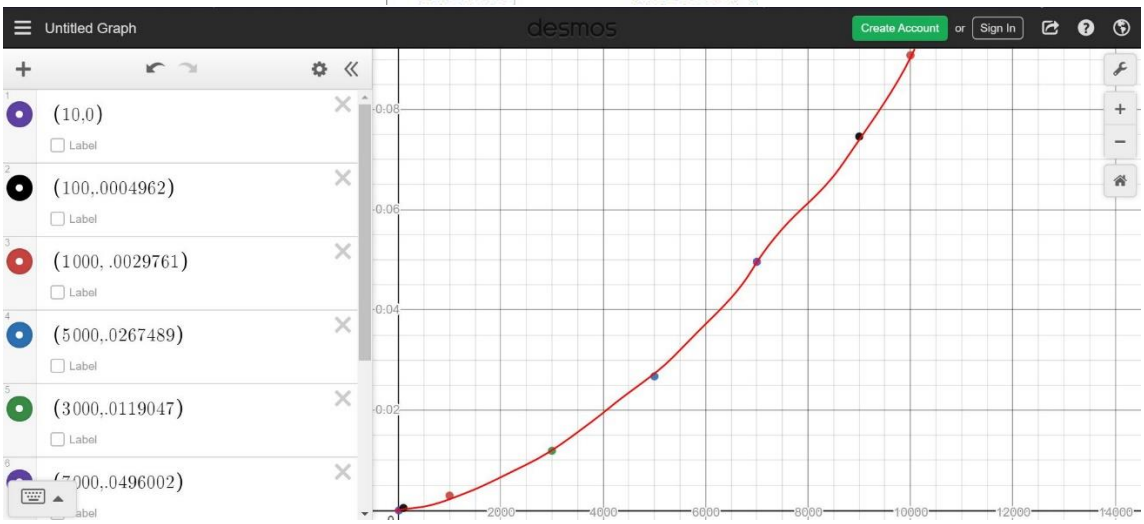
select_bubble(L,k)	
Data size	Runtime(seconds)
10	0.0004968
100	0.0009915
1000	0.1011838
5000	3.9174089
3000	1.0549914
7000	5.7758879
9000	9.7434228
10000	12.1754636



As we can see off the graph, the runtime seems quadratic, so the actual runtime coincides with the theoretical runtime. This algorithm will do n^2 comparisons since it goes through the length of the list twice.

I repeated the previous steps with the next algorithm, select_quick(). These are the results.

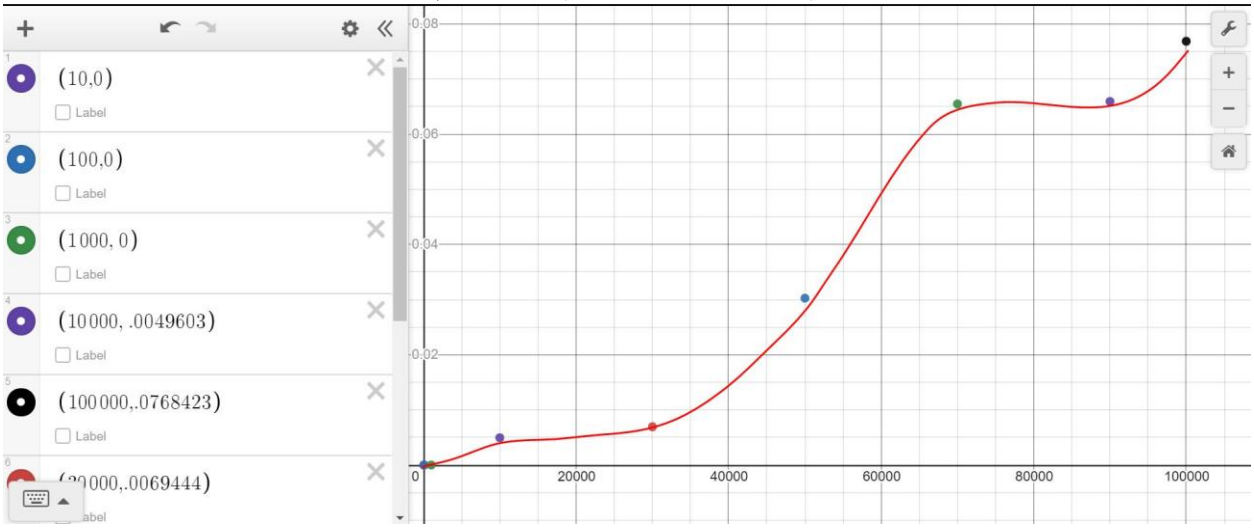
select_quick(L,k)	
Data size	Runtime(seconds)
10	0
100	0.0004962
1000	0.0029761
5000	0.0267489
3000	0.0119047
7000	0.0496002
9000	0.074588
10000	0.0907968
100000	7.4568644



Based off my calculations, I wrote that select_quick() had an asymptotic runtime of O(nlogn). However, I wrote the equation as T(n) = T(n/2) + n, which infers that every partition will be exactly in half. Since this is not the case, I was not sure exactly how to write the equation. Therefore, based off pure intuition and research, I realized that quicksort could reach runtimes as bad as O (n^2), which is what my graph seems to show us. Due to this variation, I am not quite sure if the actual runtime coincides with the theoretical one. If we consider the worst case runtime of n^2 then yes, but quicksort is often coined as a 'divide and conquer' algorithm, implying logarithmic runtime.

The next analysis was the most puzzling, select_modified_quick. For a lot of low values, it would give very similar runtimes before jumping to another set of similar values and then jumping up again. Observe the graph to get a better sense of what I mean.

select_modified_quick(L,k)	
Data size	Runtime(seconds)
10	0
100	0
1000	0
10000	0.0049603
100000	0.0768423
30000	0.0069444
50000	0.030257
70000	0.0654722
90000	0.065933



I calculated the runtime to be O(n), and if we consider the value at 70,000 to be an outlier and that its actual runtime is a bit lower, then our graph almost gives that. However, I do not play around with data to my benefit, so this is what I get. If we ignore the value at 100,000, the behavior seems almost logarithmic. I would say the actual runtime does not match the theoretical one. Also, notice how the intervals of data sizes are much bigger than in the previous experiments. This is because if I used intervals like the previous ones, the runtimes were basically identical which did not allow me to really study the behavior of the algorithm. However, it is worthy to notice that this algorithm is much faster than the other two, since I was traversing pretty large lists and the time never even hit a second.

Part #2:

I was concerned about my implementation of quicksort using stacks, since, quite honestly, I did not know what the code was doing. I had an overall big picture idea of what was going on, but if the code was to fail, I would have no idea how to troubleshoot it. To my great relief however, it worked first try. I tested it by testing it with several lists.

```
if __name__ == "__main__":
    L1 = [6,3,7,3,1,7,8,9,6,4,1,6]
    quick_sort_nr(L1)
    print(L1)
```

```
[1, 1, 3, 3, 4, 6, 6, 6, 7, 7, 8, 9]
```

```
if __name__ == "__main__":
    L1 = [1,2,3,4,5,6,7,8,9]
    quick_sort_nr(L1)
    print(L1)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
if __name__ == "__main__":
    L1 = [1]
    quick_sort_nr(L1)
    print(L1)
```

```
[1]
```

The next part was to check my select_modified_quick() but implemented with a while loop instead of recursion or stacks. I repeated a procedure similar to the one for testing the quicksort.

```
if __name__ == "__main__":
    L = [6,3,7,3,1,7,8,9,6,4,1,6]
    print(while_select(L, 0))
```

```
1
```

```
if __name__ == "__main__":
    L = [1,2,3,4,5,5,6,7,8,9]
    print(while_select(L, 0))
```

```
1
```

```
if __name__ == "__main__":
    L = [1]
    print(while_select(L, 0))
```

```
1
```

Conclusion

This lab was very fun since we had to implement and analyze different algorithms. The main takeaway from this lab was how we can predict the runtime of an algorithm using mathematics and our knowledge, but the machine will always have a mind of its own. Our predictions can be very accurate and amazing, but they can also sometimes not tell the whole story of the behavior of the algorithm. Also, the second part taught us how activation records are essentially behind everything that we code, and how we can replicate this behavior using stacks and while loops. That is what's amazing about computer science because we think certain functions can only be done one way, when in reality, there is an infinitely large amount of solutions.

Appendix

Part #1:

```
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #2 - Part 1
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#September 20, 2019
#This program implements select_bubble(L,k), select_quick(L, k)
#and select_modified_quick(L, k)

#bubble_sort(): will sort a given list using
#the bubble sort algorithm
#input: L - the list to be sorted
#output: list L sorted in acending order
def bubble_sort(L):
    for i in range(len(L)):
        for j in range((len(L) - 1) - i):
            if(L[j] >= L[j+1]):
                L[j], L[j+1] = L[j+1], L[j]
    return L

#partition(): the main part of the quicksort algorithm,
#this method will find a pivot and place it in its correct
#place by moving everything smaller than it to its right and
#eveything bigger to its right
#inputs: L - the list to be sorted
#        start - the start index
#        end - the end index
# output: the index of the pivot
def partition(L, start, end):
    i = (start - 1)
    pivot = L[end]
    for j in range(start, end):
        if(L[j] <= pivot):
            i += 1
            L[i], L[j] = L[j], L[i]
    L[i+1], L[end] = L[end], L[i+1]
    return(i + 1)
```

```
38
39  #select_bubble(): will select the kth smallest
40  #value of a given list
41  #inputs: L - the list to be traversed
42  #        k - the position we are looking for
43  #outputs: the value of L[k]
44  def select_bubble(L,k):
45      bubble_sort(L)
46      return L[k]
47
48  #select_modified_quick(): will find the kth
49  #smallest value of a list using a modified
50  #version of quicksort
51  #inputs: L - the list to be traversed
52  #        k - the position we are looking for
53  #output: the value at L[k]
54  def select_modified_quick(L, k):
55      start = 0
56      end = len(L) - 1
57      return selector(L, start, end, k)
58
59  #select_quick(): will select the kth smallest
60  #element in the list using quicksort
61  #inputs: L - the list to be sorted and traversed
62  #        k - the position we are looking for
63  #output: the value of L[k]
64  def select_quick(L,k):
65      quick_sort(L)
66      return L[k]
67
```

```
68  #selector(): the quickselect algorithm
69  #inputs: L - the list to be traversed
70  #        start - the beginning index
71  #        end - the end index
72  #        k - the index we are searching for
73  #ouput: the value of L[k]
74  def selector(L, start, end, k):
75      pivot = partition(L, start, end)
76      if(k < pivot):
77          return selector(L, start, pivot - 1, k)
78      elif(k == pivot):
79          return L[pivot]
80      else:
81          return selector(L, pivot + 1, end, k)
82
83  #sortq(): the sorting algorithm for quicksort
84  #inputs: L - the array to be sorted
85  #         start - the start index
86  #         end - the end index
87  def sortq(L, start, end):
88      if(start < end):
89          part = partition(L, start, end)
90          sortq(L, start, part - 1)
91          sortq(L, part + 1, end)
92
93  #quick_sort(): the quicksort algorithm
94  #inputs: L - the list to be sorted
95  def quick_sort(L):
96      start = 0
97      end = len(L) - 1
98      sortq(L, start, end)
```

Part #2:

```
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #2 - Part 2
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#September 20, 2019
#This program implements quicksort non - recursively
#as well as select_modified_quick using a while loop

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

#partition(): the main function of quicksort
#gets a pivot from a list, and moves everything
#less than it to the left of it and everything
#greater than it to the right
#inputs: L - a list to be sorted
#        start - the start index of the list
#        end - the end index of the list
#output: the index of the pivot
def partition(L, start, end):
    i = (start - 1)
    pivot = L[end]
    for j in range(start, end):
        if(L[j] <= pivot):
            i += 1
            L[i], L[j] = L[j], L[i]
    L[i+1], L[end] = L[end], L[i+1]
    return(i + 1)

#quick_sort_nr(): a non recursive implementation
#of the naturally recursive algorithm, quicksort
#input: L - the list to be sorted
```

```
37  def quick_sort_nr(L):
38      start = 0
39      end = len(L) - 1
40      quick_sorter_nr(L, start, end)
41
42  #quick_sorter_nr(): the actual implementation
43  #of the non recursive quicksort using a stack
44  #inputs: L - the list to be sorted
45  #        start - the start index of the list
46  #        end - the end index of the list
47  def quick_sorter_nr(L, start, end):
48      stack = [stackRecord(L, start, end)]
49      while(len(stack) > 0):
50          h = stack.pop(-1)
51          if(h.start < h.end):
52              part = partition(h.L, h.start, h.end)
53              stack.append(stackRecord(h.L, h.start, part - 1))
54              stack.append(stackRecord(h.L, part + 1, h.end))
55
56  #while_select(): an implementation of the
57  #select_modified_quick algorithm without using recursion
58  #or stacks
59  #inputs: L - the list to be traversed
60  #        k - the index of the desired element
61  #outputs: the value of L[k]
62  def while_select(L,k):
63      start = 0
64      end = len(L) - 1
65      return while_selector(L, start, end, k)
66
67  #while_selector(): the algorithm for select_modified_quick
68  #using a while loop
69  #inputs: L - the list to be traversed
70  #        start - the beginning index
71  #        end - the end index
72  #        k - the index of the element we are looking for
73  #outputs: the value of L[k]
```

```
74      def while_selector(L, start, end, k):
75          pivot = partition(L, start, end)
76          while(pivot != k):
77              if(k < pivot):
78                  pivot = partition(L, start, pivot -1)
79              elif(k > pivot):
80                  pivot = partition(L, pivot + 1, end)
81          return L[pivot]
82
83
84      if __name__ == "__main__":
85          L1 = [6,3,7,3,1,7,8,9,6,4,1,6]
86          quick_sort_nr(L1)
87          print(L1)
88          L = [4,2,7,3,2]
89          print(while_select(L, 0))
```

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.