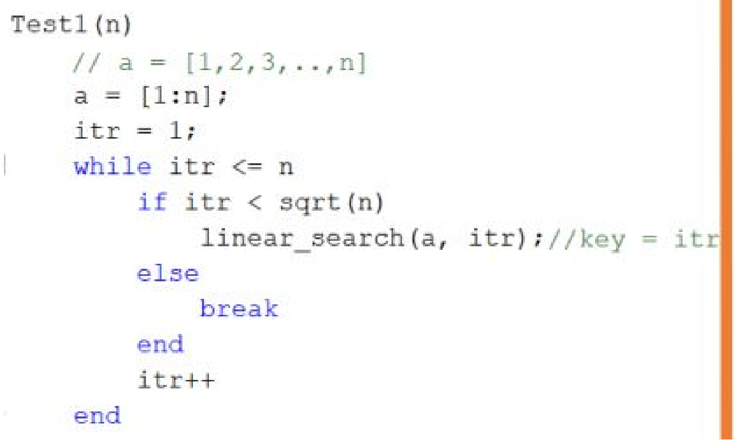
Exam 1:

Due Date: **Tuesday, March 23, 2021 at 4:00pm.**

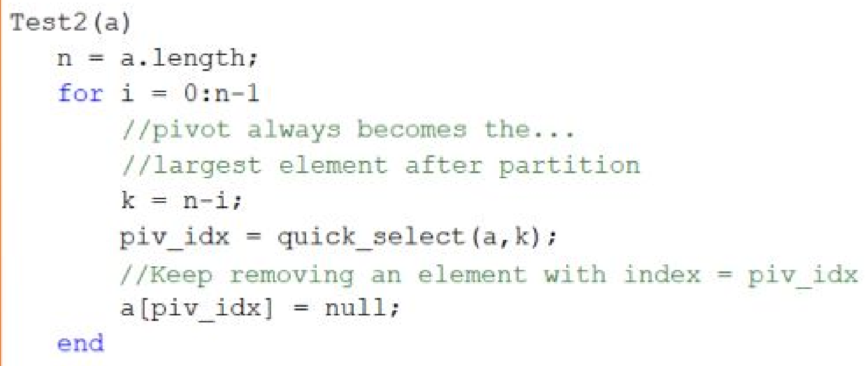
This exam contains three problems asking multiple questions. Please answer each question in detail with clear explanation. :)

Problem 1.

A. What is the growth of the below functions? **Explain in detail and show ALL the work.**



```
Test1(n)
    // a = [1,2,3,..,n]
    a = [1:n];
    itr = 1;
    while itr <= n
        if itr < sqrt(n)
            linear_search(a, itr);//key = itr
        else
            break
        end
        itr++
    end
```



```
Test2(a)
   n = a.length;
   for i = 0:n-1
       //pivot always becomes the...
       //largest element after partition
       k = n-i;
       piv_idx = quick_select(a,k);
       //Keep removing an element with index = piv_idx
       a[piv_idx] = null;
   end
```

For Test1(n):

At first glance, I saw a while loop which iterates from i = 1 to n. I also saw a linear search algorithm which iterates from the first element to the i_th element. If I did not pay attention to the "if" conditional statement, the time complexity would be a simple O(n^2) nested for loop algorithm. **However**, the if statement checks if i < less than the square root of n. This while loop can be rewritten as a "while itr <= $\sqrt{n}$". This also means that linear search will never search higher than $\sqrt{n}$ elements. This can be written as the following summation: $\sum_{i=1}^{\sqrt{n}}\sum_{j=0}^{i}1$. Further simplification → =

$$\sum_{i=1}^{\sqrt{n}} i = \frac{(\sqrt{n})(\sqrt{n}+1)}{2} = \frac{(n+\sqrt{n})}{2} = O(n)$$

For Test2(n):

This function requires more inspection and analysis than the first, as the input size shrinks by one with each recursive call. However, Quickselect is an O(n) function since it compares the pivot value with every element in the array, then shrinks its bounds. This function is called n - 1 times. It can be written as the summation: $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1$. Further simplification → = $\sum_{i=0}^{n-1} i-1 = \frac{(n-1)(n)}{2} = \frac{(n^2-n)}{2} - (n-1) = O(n^2)$

B. Compare the growth of *T_{Test1}(n)* and *T_{Test2}(n)*. **Show all the work.**

Since I know that Test1(n) = O(n) and Test2(n) = O(n^2), I can attempt to prove that Test1(n) = o(Test2(n)). This can be concluded by observing the limit of their quotient.

$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{Test1(n)}{Test2(n)} = \lim_{n\to\infty}\frac{n}{n^2} = \frac{1}{n}$. Since the limit of their quotient goes toward zero, Test1(n) = o(Test2(n)).

C. Let's say you can finish running *Test1 (a.length = 10^4)* in *1* sec. Could you estimate when you finish running *Test2* when *a.length = 100*?

Total Time = 1 second. .

$$\frac{1second}{nlines} = \frac{1second}{10,000lines} = 10^{-4}seconds/line$$

Input size = n = 10,000.

Test1(n) = O(n).

Now that we have the time per instruction, we can adjust the input to 100 and the complexity to n^2.

x = total time

$\frac{x}{Test2(n)} = \frac{xseconds}{n^2lines} = 10^{-4}seconds\frac{x}{100^2} = 10^{-4}x = 10^4 * 10^{-4} =$ $1second$.

Problem 2. A random array of size *n* is given to you and the elements in the array are integers. Your goal is finding two numbers that their product gives you the *second* largest value.

Example 1: *Input*: a = [4, 5, -8, 10, -11, 0, 7] ➔ *Output*: 10 and 7 ⟵

max = -11*(-8) = 88
2nd max = 10*7 = 70

Example 2: *Input*: a = [-4, 6, -3, -1, 5, 0, 7] ➔ *Output*: 5 and 7 ⟵

max = 7*6 = 42
2nd max = 7*5 = 35

Example 3: *Input*: a = [4, 7, -5, -8, -9, -1, 2] ➔ *Output*: -9 and -5 ⟵

max = -9*(-8) = 72
2nd max = -9*(-5) = 45

(**Hint**: First think **how many** and **what** numbers you should check to find the second largest product.)

A. How would you find the two numbers? (**Note**: If you have multiple answers in mind, break them apart and explain each one separately.) Explain each solution/algorithm in a few lines.

The first solution essentially has the same performance as the best solution, however it is cheap and lacks reusability like the second solution. It is a linear search which saves the four largest elements based on their absolute values. Four values are saved because four is the base case, in case there are four positive numbers, four negatives, three positive numbers and one negative, three negative numbers, or two and two. A single traversal gives this algorithm a time complexity of O(n). With each current max, the largest number is saved and each value trickles down and is saved until the fourth largest which discards the fifth current largest. At the end of the linear search, the four elements' products are compared to find the second largest product.

The second solution gathers the four largest numbers according to their absolute value using quick select. After this, it compares the largest product of same-signed numbers to find the second largest product. This code is very reusable, because it cuts down the input size to find the largest k numbers. It is a simple rendition of quick select with a time complexity of O(n) plus an ending algorithm which makes 16 comparisons to find the second largest product. The best case running time for this function is O(n), the average is O(n), and the worst case is a very rare O(n^2). Once the base case is reached with four numbers. Their absolute values will cease to be checked since they are already sorted enough. The second largest product can be determined by looking at just the signs of the elements. It can also easily be implemented with a max of sixteen comparisons in a nested for loop. This is because the input size has been reduced to four, the base case.

B. Write the pseudocode for the best algorithm you came up with.

```
MedianOfThreeABS(a, left, right)
     Mid = (left + right) // 2
     If abs(a[left] > abs(a[mid])
          swap(a[left], a[mid])
     If abs(a[left] > a[right])
          swap(a[left], a[right])
     If abs(a[mid] > abs(a[right])
          swap(a[mid], a[right])
     Return mid

PartitionABS(a, left, right)
     pi = MedianOfThree(a, left, right)
     pv = a[pi]
     swap(a[pi], a[end])
     right--
     While left < right
          While abs(a[left]) < abs(pv)
               Left++
          While abs(a[right]) > abs(pv)
               Right--
          If left => right
               break
          If abs(a[left] > abs(pv) and abs(a[right]) < abs(pv)
               swap( a[left], a[right]
               Left++, right--
     Return left

Max4Nums(a, left, right)
     K = a.len - 4
     While left < right
          Pivotindex = partitionABS(a, left, right)
          If k - 1 = pivotindex
               Return a[k to end]
          Elif k - 1 < pivotindex
               Right = pivotindex - 1
          Else
               Left  = pivotindex + 1
     Return a[k:end]
```

```
SecondLargestProduct(a):
Four_array = max4nums(a, left, right)
For i = 0 to 4
     For j = i to 4
          Temp = four_array[j] * four_array[i]
          If max < temp
               Max2 = max
               Max = temp
          Elif max2 < temp
               Max2 = temp
Return max2
```

C. Implement your answer using any programming language you want to.

```
def median_of_three_abs(a, left, right):
   mid = (left + right) // 2
   if abs(a[left]) > abs(a[mid]):
       a[left], a[mid] = a[mid], a[left]
   if abs(a[mid]) > abs(a[right]):
       a[mid], a[right] = a[right], a[mid]
   if abs(a[mid]) < abs(a[left]):
       a[mid], a[left] = a[left], a[mid]
   return mid
```

```
def partition_abs(a, left, right):
   pi = median_of_three_abs(a, left, right)
   pv = a[pi]
   end = right
   a[pi], a[right] = a[right], a[pi]
   right -= 1
   while left < right:
       while abs(a[left]) < abs(pv):
           if left >= right: break
           left += 1
       while abs(a[right]) > abs(a[pi]):
           if left <= right: break
           right -= 1
```

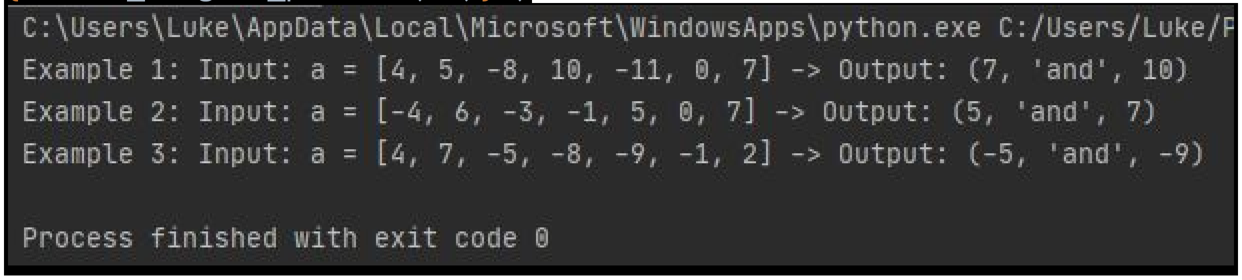
```
        if left >= right: break
        a[left], a[right] = a[right], a[left]
        left += 1
        right -= 1
    if left == right:
        left -= 1
    a[left], a[end] = a[end], a[left]
    return left


def max_4_nums(a, left, right):
    k = len(a) - 4
    end = right
    while left < right:
        pi = partition_abs(a, left, right)
        if k - 1 == pi:
            return a[k:end + 1]
        elif k - 1 < pi:
            right = pi - 1
        else:
            left = pi + 1
    return a[k:end + 1]


def second_largest_product(a):
    four_hi = max_4_nums(a, 0, len(a) - 1)
    prod = [0, 0]
    prod2 = [0, 0]
    for i in range(4):
        for j in range(i):
            temp = four_hi[j] * four_hi[i]
            if i == j:
                temp = 0
            if prod[0] * prod[1] < temp:
                prod2 = prod
                prod = [four_hi[j], four_hi[i]]
            elif prod2[0] * prod2[1] < temp:
                prod2 = [four_hi[j], four_hi[i]]
    s = f"{prod2[0]} and {prod2[1]}"
```

```
return s
```

```
a1 = [4, 5, -8, 10, -11, 0, 7]
a2 = [-4, 6, -3, -1, 5, 0, 7]
a3 = [4, 7, -5, -8, -9, -1, 2]
print(f"Example 1: Input: a = {a1} -> Output:
{second_largest_product(a1)}")
print(f"Example 2: Input: a = {a2} -> Output:
{second_largest_product(a2)}")
print(f"Example 3: Input: a = {a3} -> Output:
{second_largest_product(a3)}")
```



```
C:\Users\Luke\AppData\Local\Microsoft\WindowsApps\python.exe C:/Users/Luke/P
Example 1: Input: a = [4, 5, -8, 10, -11, 0, 7] -> Output: (7, 'and', 10)
Example 2: Input: a = [-4, 6, -3, -1, 5, 0, 7] -> Output: (5, 'and', 7)
Example 3: Input: a = [4, 7, -5, -8, -9, -1, 2] -> Output: (-5, 'and', -9)

Process finished with exit code 0
```

D. What is the time complexity of your answer? **Explain in detail and show all the work.** (**Note:** If possible, break your code/pseudocode to different parts, calculate the runtime for each step and then try to calculate the total running time based on that.)

```
def second_largest_product(a):
    four_hi = max_4_nums(a, 0, len(a) - 1)    ← Stores max four
elements in an array. O(n) time, O(1) space. This function is a
modified quickselect which heavily narrows the bounds of the
observable array. With the first partition, n comparisons are made.
Based on the pivot index, this step can take as little as O(n)
instructions or as great as ($n^2$)instructions.
    prod = [0, 0] ← max product O(1)
    prod2 = [0, 0] ← second largest product O(1)
    for i in range(4): ← loop through mini array O(16)
        for j in range(i): ← nested to find all possible combos, ^
            temp = four_hi[j] * four_hi[i] ← product of each O(1)
combination
            if i == j:
                temp = 0 ← Prevent false case O(1)

            if prod[0] * prod[1] < temp: ← Trickle down max product
O(1)

                prod2 = prod ← ^ O(1)

                prod = [four_hi[j], four_hi[i]] ← ^ O(1)

             elif prod2[0] * prod2[1] < temp: ← ^ O(1)

                prod2 = [four_hi[j], four_hi[i]] ← ^ O(1)
```

```
14    s = f"{prod2[0]} and {prod2[1]}" ← O(1)

15    return s ← return elements of product as formatted string. O(1)
```

<u>Step:</u>	<u>Input Size:</u>	<u>Tree:</u>	<u>Line:</u>
0	n	n	0 - $\Theta(1)$
1	n / 2	$n/2$	1 - Θ(n) n = 4
2	n / 4	$n/4$	2 - $\Theta(1)$
3	n / 8	$n/8$	3 - $\Theta(1)$
4	n / 16	$n/16$	4 - $\Theta(1)$
|			5 - $\Theta(1)$
|			6 - $\Theta(1)$
|			7 - $\Theta(1)$
k	1 = n / 2^k		8 - $\Theta(1)$
	2^k = n		9 - $\Theta(1)$
	log(2^k) = log(n)		10 - $\Theta(1)$
	k = log(n)		11 - $\Theta(1)$
			12 - $\Theta(1)$

T(n) = $n + n/2 + n/4 + n/2^{k-1} + n/2^k$

13 - $\Theta(1)$

$$= \sum_{i=0}^{n} \frac{n}{2^i} = n \sum_{i=0}^{n} (\tfrac{1}{2})^i = n\frac{(1/2)^{n+1} - 1}{(1/2) - 1}$$

14 - $\Theta(1)$

15 - $\Theta(1)$

= n^2.....

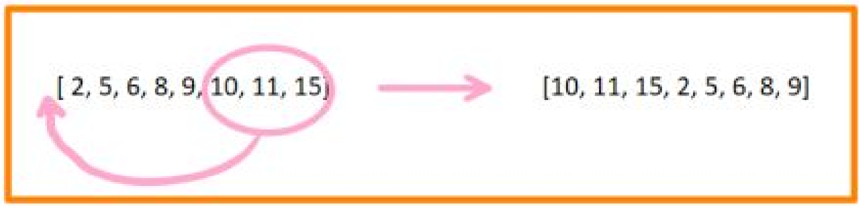
= O(n^2)← In the worst case...

*Recursion → Will execute n^2 instructions, max.

However, the average case is O(n) since the input size is reduced on average by half of the array, with the choice of a good pivot. That is why pivot selection algorithms are very valuable, so a near best case algorithm can be attained.

Problem 3. You were given a sorted array. Pouye shifted the last k numbers to the front. Implement an algorithm to figure out how many numbers are brought to the front.

An example of a sorted array where last k numbers were shifted to front:



A. How would you find the total number of the elements shifted to the front? (**Note**: If you have multiple answers in mind, break them apart and explain each one separately.) Explain each solution/algorithm in a few lines.

I could do a linear search which compares a[i] from i = 0, iterating until a[i] < a[n - 1], (n = length of array), returning the amount of elements greater than or equal to the last element. This would return the shifted quantity of elements. The time complexity for this algorithm would be best case constant, O(1) when there are no shifted elements. Average case, it would traverse half of the array (n / 2 = O(n)). Worst case, the algorithm would run through the entire array to find that the split is between the second to last and last elements (n = O(n)).

I <u>will</u> do a binary search which compares the last element to the first initially to make the best case the same as the linear search's best case (O(1)). Then, I will go about binary search similar to the binary string lab to find where the split from numbers greater than the last element of the array and those less than the last element of the array. This will give me the index of the shift which I add 1 to and return. They key idea behind this fast search algorithm, which will run best case O(1), average and worst case O(log(n)), is that it reduces the size of the array by ½ with each iteration, giving it its effective and fast runtime of O(log(n)).

B. Write the pseudocode for the best algorithm you came up with.

```
Binarysearch(a, left, right)
     Mid = (left + right) / 2
     If a[0] < a[end]
          Return 0
     Elif a[mid] >= a[end] and a[mid + 1] < a[end]
          Return mid + 1
     Elif a[mid] >= a[end]
          Binarysearch(a, mid + 1, right)
     Elif a[mid] < a[end]
          Binarysearch(a, left, mid - 1)
```

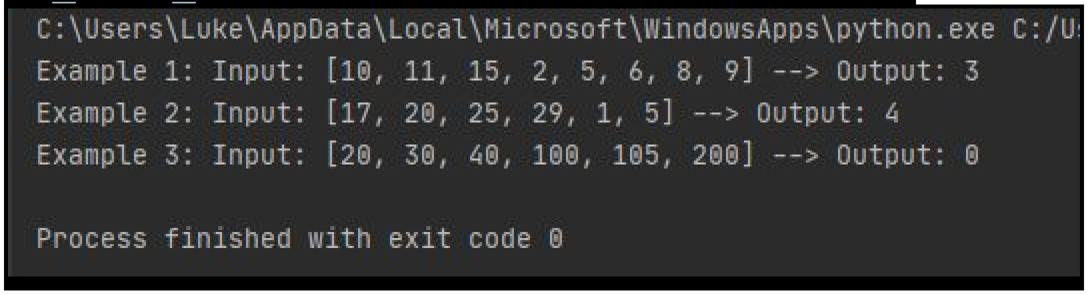
Very similar to implemented code, surprisingly

C. Implement your answer using any programming language you want to.

```
def k_nums_shifted(a, left, right):
   mid = (left + right) // 2
   end = len(a) - 1
   if a[0] < a[end]:
       return 0
   elif a[mid] >= a[end] > a[mid + 1]:
       return mid + 1
   elif a[mid] >= a[end]:
       return k_nums_shifted(a, mid + 1,
right)
   elif a[mid] < a[end]:
       return k_nums_shifted(a, left, mid
- 1)
```

```
# Tester
a1 = [10, 11, 15, 2, 5, 6, 8, 9]
a2 = [17, 20, 25, 29, 1, 5]
a3 = [20, 30, 40, 100, 105, 200]
print(f"Example 1: Input: {a1} --> Output:
{k_nums_shifted(a1, 0, len(a1) - 1)}")
print(f"Example 2: Input: {a2} --> Output:
{k_nums_shifted(a2, 0, len(a2) - 1)}")


print(f"Example 3: Input: {a3} --> Output:
{k_nums_shifted(a3, 0, len(a3) - 1)}")
```



```
C:\Users\Luke\AppData\Local\Microsoft\WindowsApps\python.exe C:/U
Example 1: Input: [10, 11, 15, 2, 5, 6, 8, 9] --> Output: 3
Example 2: Input: [17, 20, 25, 29, 1, 5] --> Output: 4
Example 3: Input: [20, 30, 40, 100, 105, 200] --> Output: 0

Process finished with exit code 0
```

D. What is the time complexity of your answer? **Explain in detail and show all the work.** (**Note**: If possible, break your code/pseudocode to different parts, calculate the runtime for each step and then try to calculate the total running time based on that.)

```
1 def k_nums_shifted(a, left, right):
2   mid = (left + right) // 2
3   end = len(a) - 1
4   if a[0] < a[end]:   ← No shifted numbers
5       return 0
6   elif a[mid] >= a[end] > a[mid + 1]: ← Base case: k number split
7       return mid + 1
8   elif a[mid] >= a[end]: ← If element greater than last element, go right
9       return k_nums_shifted(a, mid + 1, right)
10   elif a[mid] < a[end]: ← If element less than last element, go left
11      return k_nums_shifted(a, left, mid - 1)
```

Step:	Input Size:	Tree:	Line:
0	n	1	0 - $\Theta(1)$
1	n / 2	1	1 - $\Theta(1)$
2	n / 4	1	2 - $\Theta(1)$
3	n / 8	1	3 - $\Theta(1)$
4	n / 16	1	4 - $\Theta(1)$
|			5 - $\Theta(1)$
|			6 - $\Theta(1)$
|			7 - $\Theta(1)$
k	1 = n / 2^k	1	8 - $\Theta(1)$
	2^k = n		9 - $\Theta(1)$: n = n / 2
	log(2^k) = log(n)		10 - $\Theta(1)$
	k = log(n)		11 - $\Theta(1)$: n = n / 2

T(n) = k + 1
      = log(n) + 2
      = O(log(n))

*Recursion → Will execute 5log(n) instructions, max.