COMPSCI 2CO3

Assignment 1

Feb 16, 2021

Omar Zahir - zahirm1 - 400255963
Talha Amjad - amjadt1 - 400203592

Q1.

a)

```
procedure reverse(L):
    D = new Doubly_Linked_List
    y = new Node
    x = new Node
    x = D.head
    y = L.tail
    x.val = y.val
    y = y.prev
    x.prev = null
    while y != null:
        z = new Node
        x.next.val = y.val
        z = x
        x = x.next
        x.prev = z
        y = y.prev
    x.next = null
    D.tail = x
    return D
```

b)

Explanation

The algorithm above takes a doubly linked list as input, as well as creating a new doubly linked list, *D*, to store the new list and two new nodes, *y* and *x,* for storing the values of the two lists. The y node is primarily for storing the nodes of the original list, and the x node is for storing the values of the new lists. x is initialized as the head of the new list and y is initialized as the tail of the old, with x taking the value of y. y is then set to one previous of itself while the previous of x is set to null since it is the head. The previous instructions each had a constant running time of O(1).

y is then checked to see whether it is the head of the list, i.e, checking if it is null, and a new node *z*, is initialized as a temporary node. The next value of x is then set to the current value of y, and x is then set to its next after storing the current x in z so it can be set as the previous of x after x is set to its next. y is then set to its previous and the latter few procedures are repeated until the previous value of y is null, implying it is the head. The previous procedures, although constant, are repeated in a while loop that is roughly going the length of the linked list, thus implying that the complexity is linear or O(n).

This means that the current x is now the tail so we must set the next of the tail to be null. The list D is then returned at the end. The last instruction is also constant runtime, which would thus imply that the runtime of the entire algorithm is O(n) due to the while loop going the length of the list. **Runtime: O(n)**

Q2.

Written in Pseudocode

```
procedure NSV(A)
    stack = new Stack()
    array = []

    for i in [0..A.length]
        array += [A[i]]
    array += [0]
    smaller = []

    for i in [0..A.length]
        smaller += [0]

    for i in [0..array.length]

        while (!stack.IsEmpty()) & (stack.peek())[0] > array [i]
            ci = (stack.peek())[1]
            difference = i - ci
            smaller [ci] = difference
            stack = stack.pop()

        array2 = []
        array2 += [array[i]]
        array2 += [i]
        stack.push(array2)

    return smaller
```

In this algorithm, although there are more than a few for loops, the first two loops are separate and are unnested, meaning they just add to the complexity of n, insead of making the complexity greater than n. For the second for loop, although it is nested, due to the nature of the algorithm's implementation, the condition on the for loop will decrease over time, making it the overall complexity within **O(n).**

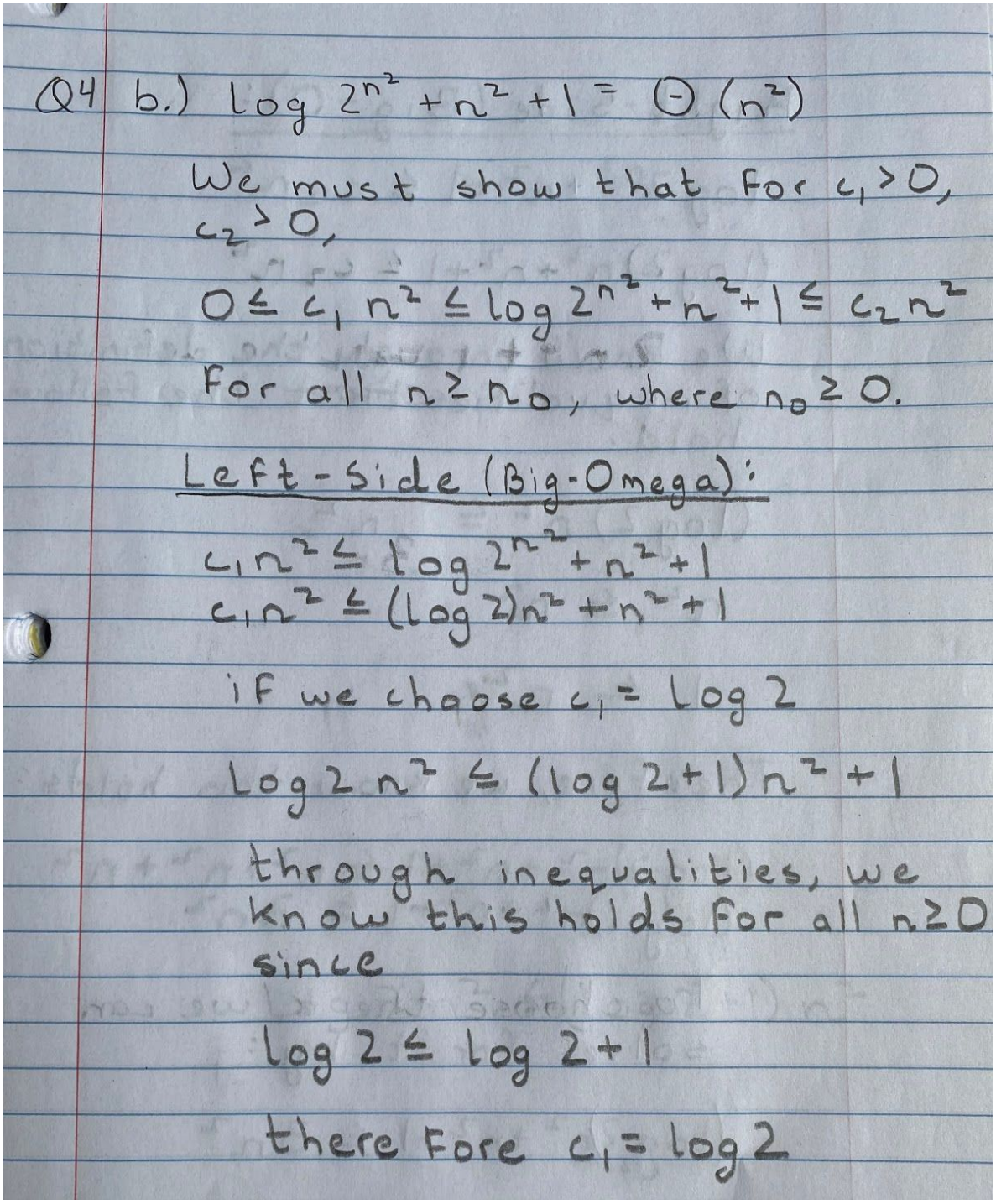
Q3.

Written in Pseudocode

```
Class Queue()

    s = new Stack

    procedure enq(s,k)

        z = new Stack

        while !(s.isEmpty())
            y = s.pop()
            z.push(y)

        s.push(k)

        while !(z.isEmpty())
            y = z.pop()
            s.push(y)

        return s

    procedure deq(s,k)

        if !(s.isEmpty())
            return s.pop()

    procedure isEmpty(s)

        return s.isEmpty()

    procedure size(s)

        return s.size
```

Q4.

b)



Although the value of c1 is log 2, it can also be written as 1.

Right-Side (Big-O):

$$\log 2^{n^2} + n^2 + 1 \leq c_2 n^2$$

$$(\log 2) n^2 + n^2 + 1 \leq c_2 n^2$$

We know through the definition of inequalities that the following hold:

$$(\log 2) n^2 \leq n^2$$

$$n^2 \leq n^2$$

$$1 \leq n^2$$

Therefore, we know this holds:

$$(\log 2) n^2 + n^2 + 1 \leq n^2 + n^2 + n^2$$
$$(\log 2) n^2 + n^2 + 1 \leq 3n^2$$

To choose the k we can solve for it by:

$$(\log_2 2) n^2 + n^2 + 1 \leq 3n^2$$

$$n^2 + n^2 + 1 \leq 3n^2$$
$$1 \leq 3n^2 - 2n^2$$
$$1 \leq n^2$$
$$n = \pm 1, \text{ and since } n_0 \geq 0$$

$$\therefore c_1 = \log 2, c_2 = 3, k = 1$$

c)

Q4. $1+2+3+\ldots+n = O(n^2)$

We have to prove that

$0 \leq 1+2+3+\ldots+n \leq c \cdot n^2$

for all $n \geq n_0$, with $c > 0$

The sum can be approximated:

$$1+2+3+\ldots+n = \frac{n^2+n}{2}$$

$$\frac{n^2+n}{2} \leq c \cdot n^2$$

$$\frac{1}{2}n^2 + \frac{1}{2}n \leq c \cdot n^2$$

We know by the nature of $\leq$, we can say

$\frac{1}{2}n^2 \leq \frac{1}{2}n^2$ AND $\frac{1}{2}n \leq \frac{1}{2}n^2$

which leads us to:

$$\frac{1}{2}n^2 + \frac{1}{2}n \leq \frac{1}{2}n^2 + \frac{1}{2}n^2$$

$$\frac{1}{2}n^2 + \frac{1}{2}n \leq n^2$$

$\therefore c = 1, k = 1$

Q5.

a)

Q5a) $(1000n^4 + n^2 + 4n)/(\log n) = \Theta(n^4)$

By the definition of Big Theta, we need to show that either Big O or Big Omega is false, as disproving one of them would disprove Theta to be false.

Big Omega: $(\Omega(n^4))$

$c \cdot n^4 \leq (1000n^4 + n^2 + 4n) / \log n$

$c \cdot n^4 \log n \leq 1000n^4 + n^2 + 4n$

$c \cdot n^3 \log n \leq 1000n^3 + n + 4$

$c \cdot n^3 \log n - 1000n^3 \leq n + 4$

$n^3 \cdot (c \cdot \log n - 1000) \leq n + 4$

Since we need to show that this statement does not hold, we will take c as the lowest possible constant of 1. With that substitution, we can clearly see that when $n > 2^{1000}$, the left-side will be much greater than the right, thus proving the inequality False.

$\therefore (1000n^4 + n^2 + 4n)/(\log n) = \Theta(n^4)$ is False

b)

Q5b) Prove $\sqrt{n} = O(\log n)$ is False

This can be proven through a proof by contradiction.

We will assume that $\sqrt{n} = O(\log n)$ which implies that for all $c > 0$.

$\sqrt{n} \leq c \log n$ holds

$\frac{\sqrt{n}}{c} \leq \log n$

If we choose c^4 we get:

$c^{4/2-1} \leq \log c^4$

$c \leq 4 \log c$

Since we know that $n > \log n$, which is currently represented with c, we know that for all $n \geq c^4$ would make the inequality false.

$\therefore \sqrt{n} = O(\log n)$ is proven False

Q6.

Written in pseudo-code.

a)

```
procedure summer_On3(l):
    max, mi, mj = 0, 0, 0
    for i in [0..l.length]
        for j in [i..l.length]
            sum = 0
            for k in [i..j + 1]
                sum += l[k]
            if (sum > max)
                max = sum
                mi = i + 1
                mj = j + 1
    if max < 0
        return (0,0,0)
    return (max, mi, mj)
```

The algorithm written above can be seen to exceed O($\mathbf{n^2}$) as we have three nested for loops, where the first ranges from 0 to n, making it O(**n**), and the nested for loop within it, making the complexity of such loops O($\mathbf{n^2}$). For a loop within that, because it can potentially go from 0 to n as well in the worst case, it is safe to say that the complexity can be determined as O($\mathbf{n^3}$).

b)

```
procedure summer_On2(l)
    max, mi, mj = 0, 0, 0
    for i in [0..l.length]
        for j in [i..l.length]
            if sum(l[i:j+1]) > max
                max = sum(l[i:j+1])
                mi = i + 1
                mj = j + 1
    if max < 0
        return (0,0,0)
    return (max, mi, mj)
```

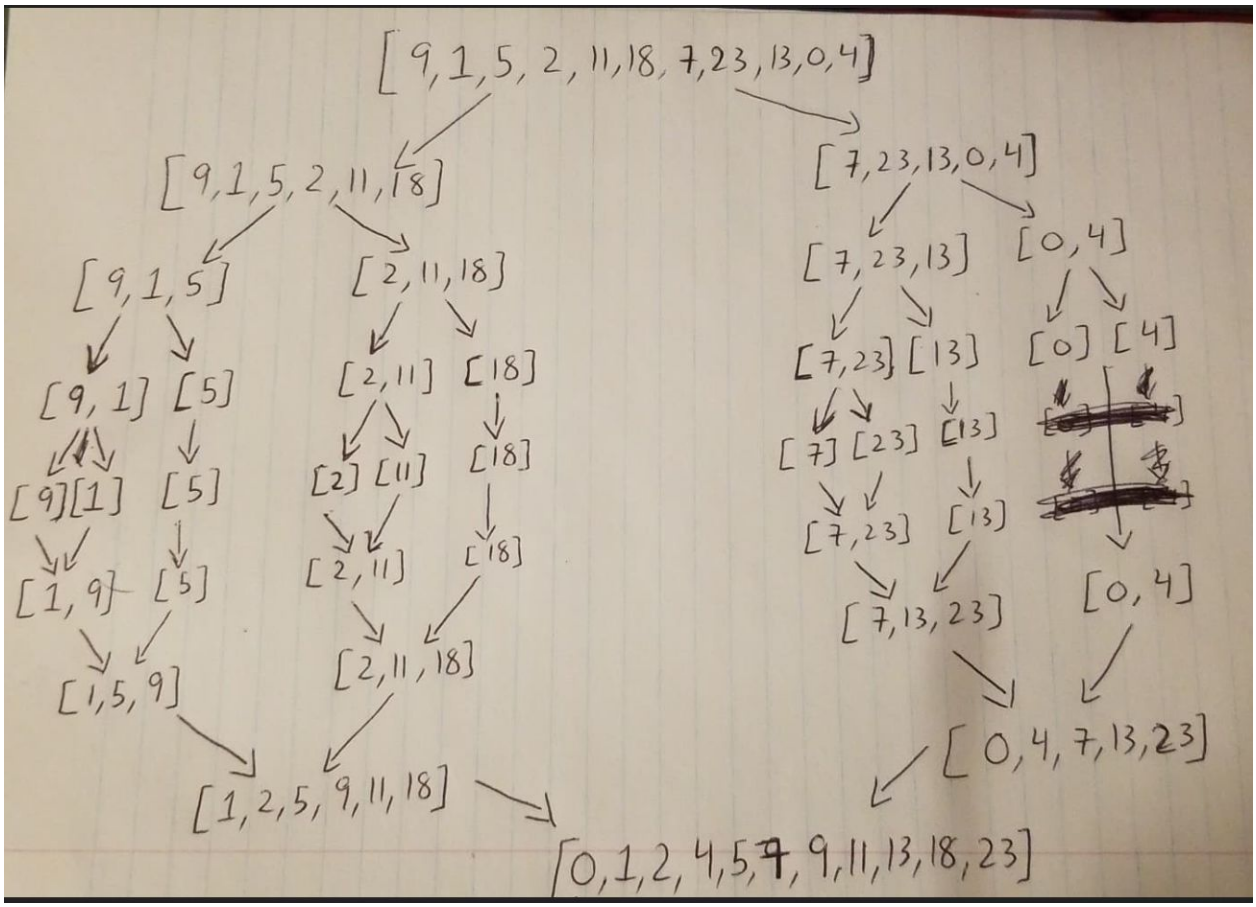
The algorithm above is implemented to be O($\mathbf{n^2}$). The implementation includes two nested for loops and because from the scenario above we know that an algorithm going from 0 to n and an algorithm going from i to n would be put in the complexity of O($\mathbf{n^2}$).

Q7.

Written in Pseudocode

```
procedure 4_Sort(a,b,c,d)
    if a > b
        l1 = b
        h1 = a
    else
        l1 = a
        h1 = b

    if c > d
        l2 = d
        h2 = c
    else
        l2 = c
        h2 = d

    if l1 > l2
        small = l2
        mid1 = l1
    else
        small = l1
        mid1 = l2

    if h1 > h2
        big = h2
        mid2 = h1
    else
        big = h1
        mid2 = h2

    if mid1 > mid2
        return(small, mid2, mid1, big)
    else
        return(small, mid1, mid2, big)
```

Q8.



In the picture above, you can see the implementation for a top-down mergesort. In a top-down mergesort, the algorithm keeps dividing the given list into halves. Once each element on either side is split into its own element, the two elements next to each other are compared and swap places if one of them is greater than the other. Furthermore, when each side is sorted and merged together, we merge the end result of both sides to get our finalized array. As a result, in the picture above, you can see the final array is sorted.

Q9.

```
procedure nonRecursiveQuicksort(L)
    stack = new Stack
    stack.push(0)
    stack.push(L.length)
    while !stack.isEmpty()
        l = stack.pop()
        f = stack.pop()
        if (2 > (l - f))
            continue
        p = ((l - f) // 2) + f
        p = partition(L, l, f, p)
        stack += [p + 1]
        stack += [l]
        stack += [f]
        stack += [p]
    return L

procedure partition(L, l, f, i)
    pivot = L[i]
    t1 = L[i]
    j = l-1
    L[i] = L[j]
    L[j] = t1
    low = f
    hi = l - 2
    while hi > low
        if L[hi] > pivot
            hi = hi - 1
        elif L[low] <= pivot
            low = low + 1
        else
            t2 = L[low]
            L[low] = L[hi]
            L[hi] = t2
    x = hi
    if L[hi] < pivot:
        x = x + 1
    t3 = L[j]
    L[j] = L[x]
    L[x] = t3
    return x
```

Q10.

a)

Yes, an array that is sorted in decreasing order is a max-oriented heap by the definition of a max oriented heap, which says that the parent nodes must be greater than or equal to the child nodes. We can say this property holds for an array sorted in decreasing order through the property of the max oriented heap which is created through the array by assigning the two children node of an index *i* to be the values at the indexes 2(*i*) + 1 and 2(*i*) + 2, where if no such indexes exist, the children nodes do not exist. For that reason, because we know that a descending order has the property that all values at indexes after it are either smaller than or equal to it, that would prove that indeed decreasing ordered arrays are max-oriented heaps.

That being said, it is necessary to make the distinction that while all descending orders are max heaps, not all max heaps are descending orders, as for a heap to be a max heap, the only requirement is that all the parent nodes are greater than or equal to the child nodes, which could mean that the left child node can be bigger than the right child node, meaning that if it were to be converted back into an array, it would not be sorted descendingly.

b)

The following answers are assuming that the heapsort is using a max-heap algorithm.

Least number of compares: [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

An array with all elements of equal values, such as the one shown above, would give you the least amount of compares as the heap would always remain in the “heap” state as the values are equal and there would simply be one comparison each time with the child nodes. Everytime the top and bottom elements are switched, we know that there will only be one comparison with the two children below the node, and if the condition is to check less than equal, the nodes can remain in their place and it can move on to the next switch.

Most number of compares: [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

Any fully sorted array, such as the one shown above, can provide the worst case in terms of compares, given that it is a max heap. This is because when this is converted into a heap, using the k, and 2k+1 and 2k+2 technique, we will see that the heap will essentially be sorted in the order opposite of what it should be. This would mean that everytime a comparison is made, the heap will need to fix the order of the heap as all the parent nodes are the child nodes and vice-versa, which leads to the maximum number of comparisons possible with a heap structure.