Dela Rosa, Pamella Jei S.
2020-06676

PART 1. The Big O

1. The increment function contains a for loop that adds 1 to each of the elements in the list. The incrementation process only takes O(1) while iterating through each element takes O(n). This gives us a time complexity of O(1*n) = O(n).

   Moreover, the while loop does not account for each of the elements in the list. Instead, it uses a counter that skips by 3 and has a runtime of O(logn). The arithmetic only takes up O(1) so the total time complexity is given by O(1*logn) = O(logn).

   Since the increment function is nested in the while loop, the overall time complexity of the code will be O(n*logn) = **O(nlogn)**.

2. The code contains 3 for loops nested together, each running at a time complexity of O(n), linearly dependent on the size of n. Since decrementing only takes up O(1), the overall runtime is O(1*n*n*n) = **$O(n^3)$**.

3. The ten function compares values through the min function which only runs at constant time.

   The first for loop iterates from the range 0 to 10 and is independent from the input, which also runs at constant time. Nested in it is a for loop that iterates from the range of 0 and the result of the ten function, also independent from input and runs at constant time.

   Since all of the operations do not base the runtime on the input, the time complexity is constant **O(1)**.

4. The bar function is mainly comprised of two if-else statement. If we don't meet the conditions of the if statement, the function would only return -1 with a time complexity of O(1). But if the conditions are met, we proceed with the recursive function.

   The function will continue to add to i or subtract to j until foo[x] == y. In its worst case, we can assume that the time complexity will follow the length of y which is given by **O(n)**.

5. There are two nested loops in the function foo. The first loop runs through the range of list n denoted by O(n) while the second loop runs through the range of k which we denote by O(m).

   The following operations all run in constant time so if we will take the overall time complexity of function foo, we will have O(m*n) = **O(mn)**.

6. The while loop is dependent on two variables: i and n. It executes until i is greater than n which gives us a runtime of O(n). However, it is also noticeable that the value

of i increases to n due to the modulo condition. This means that it also runs in O(n). Overall, the code has a time complexity of O(n*n) = **$O(n^2)$**.

7. The function has two main paths that it will follow. If the prod is less than 0, there is only one for loop that iterates over the length of li1. Otherwise, it will follow 2 for loops that iterates over the length of li1 and li2. At its worst case, following the else statement, the time complexity of the function is O(m*n) = **O(mn)**.
8. In this recursive function, we divide n by 2 before calling the function again which gives us a logarithmic time complexity of the form **O(logn)**.
9. In this recursive function, we subtract 3 from n before calling the function again which gives us a linear time complexity of O(n-3) = **O(n)**.
10. The for loop iterates each member in list a so it runs at O(m). On the other hand, the membership operator (in) checks membership for each element in list b so it runs at O(n). Since the membership operator is nested in the for loop, the overall time complexity is O(m*n) = **O(mn)**.

PART 2. Problem Solving

1. The code I created based on the pseudocode is as follows:

```
from collections import deque
A = deque([1,2,…,n]) #queue
B = [] #stack

count = 0
while len(A) !=0:
    if len(B) == 0 or B[-1] <= A[0]:
        x = A.popleft()
        B.append(x)
    else:
        x = B.pop()
        A.append(x)
    count = count + 1
print(count) #the number of iterations of the while loop
```

I used an increment counter that takes the number of iterations done by the while loop. To find the worst and best case, I set up a smaller test case where n=10 such that A = deque([1,2,3,4,5,6,7,8,9,10]).

When queue A was in increasing order, the count was 10 (n) but when it was in decreasing order, the count was 100 (n*n). From then, we can deduce that the **worst case is when A is in decreasing order** and the **best case is when A is in increasing**

order. Following the worst case, the maximum number of iterations is given by n^2. In the case where n=100, **the maximum is 10000**.

2. We must first get the values of a, b, c, d, e, f, and g from the constraints.

$a^2 - 8a - 9a$
$(a-9)(a+1)$
$a = 9$

$b + c = a - 3 = 9 - 3 = 6$
$c < b < a = 9 \; and \; b + c = 6$
$c = 1,2 \qquad b = 5,4$

$b + c < d < a - 1$

$6 < d < 9 - 1$
$6 < d < 8$
$d = 7$
$4(b + c) < f < 3a$
$4(6) < f < 3(9)$
$24 < f < 27$
$f = 25, 26$
$f < h < g$
$25,26 < h < g$

To summarize:

$a = 9$
$b = 1,2$
$c = 5,4$
$d = 7$
$e = 22$
$f = 25,26$
$25, 26 < h < g$

Since list N will be arranged in an increasing order and S must always be in the middle of the partitions, we arrange the N such that N = [b, c, d, a, S, e, f, g, h]

This means that S must be in the middle of a and e, that is $9 < S < 22$ which gives us 12 possible answers. Selecting 6 possible integers, we have **11, 12, 13, 14, 15, 16**.

3.

a. print(h(1,"fruits"))

```
def h(n,str):
    while n != 1:
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n + 1
        str = f(str)
    return str
```

In this part of the code, the while statement will run until n is not equal to one. But since we already established that n is 1, we will skip the while loop and proceed with returning str which we also defined as "fruits."

Hence, the output is **"fruits"**

b. print(h(2,"fruits"))

```
def h(n,str):
    while n != 1:
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n + 1
        str = f(str)
    return str
```

In this part of the code, since n = 2 is not equal to 1, we proceed with the while loop. Since 2%2 =0, n will become 2/2 = 1 and then we proceed with the if statement, that is, assigning str into the function f

```
def f(str):
    if len(str) == 0:
        return ""
    else if len(str) == 1:
        return str
    else
        return f(g(str)) + str[0]
```

In this function, we check for the length of str. Since we haven't done anything to str, its length remains as 5 and we proceed with recursing the function and calling the function g

```
def g(str):
    int i = 0
    string new_str = ""
    while i < len(str) - 1:
        new_str = new_str + str[i+1]
        i = i + 1
    return new_str
```

In this function, we have a while loop that will run until i is equal to the length of str -1, which is 4 in this case. The loop concatenates the letter on the i+1th position to a new_str and returns it. Essentially, the new_str will be str without the 1st letter which is "ruits"

To sum up the recursive process, we have

f(g(fruits)) + fruits[0]

f(g(ruits)) + ruits[0] + f

f(g(uits)) + uits[0] + r +f

f(g(its)) + its[0] + u+ r + f

f(g(ts)) + ts[0] + i + u + r + f

f(g(s)) + t + i + u + r + f

s + t + i + u + r + f = **stiurf**

c. print(h(5,"fruits"))

It is important to note that when n!=1, str goes through the process of functions f and h which yields the reverse of str (in this case, "stiurf"). Another iteration would turn return str back to its original string (in this case, "fruits").

When n = 5, n is not equal to 1 so we proceed with the while loop. Since 5%2 is not 0, we follow the else statement and turn n = 3*5 + 1 = 16. For the first iteration, we will have str = "stiurf." Since n is now 16 and 16%2 = 0, we can proceed to the if statement and turn n = 16/2 = 8. We will have another iteration which will yield str as "fruits." Since n is now 8 and 8%2 = 0, we still proceed to the if statement and turn n = 8/2 = 4. This is the third iteration

and thus yields str as “stiurf.” Now that n is 4 and 4%2 = 0, we still proceed to the if statement and turn n = 4/2 = 2. This fourth iteration assigns str into “fruits.” Finally, n is 2 and 2%2 = 0, following the if statement, we turn n = 2/2 = 1. This will be the fifth iteration with the resulting str as “stiurf.” We have reached the end of the while statement (with n ==1), so we return the final form of str which is **“stiurf.”**

d. print(h(pow(2, 1000000000000000), "fruits"))

Following the logic of part c, if n can be divided by 2 in an odd number of times, there will be an odd number of iterations which results to the reverse of str (“stiurf”). Otherwise, if n can be divided by 2 in an even number of times, there will be an even number of iterations which reverts str to the original (“fruits”).

Since we know that these iterations are dependent on n%2 and n/2, when n has a base of 2, the iterations will depend on the exponent of 2 (the number of times 2 can be divided to 2). If the exponent is an odd number, the result will be the reverse of str (“stiurf”). If the exponent is even, str reverts to the original (“fruits”).

Since the exponent of 2 here is 1000000000000000, an even number, str will be **“fruits.”**

e. print(h(pow(2, 9831050005000007), "fruits"))

In this cae, the exponent of 2 is 9831050005000007, an odd number, so str will be **“stiurf.”**

4. The code I created based on the pseudocode is as follows:

```
stack = [[0],[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12],
         [13],[14],[15],[16],[17],[18],[19],[20],[21],[22],[23],[24]]
entry = ""
commands = []
while entry != ["0",0,"0",0]:
    entry = input().split(" ")
    entry[1] = int(entry[1])
    entry[3] = int(entry[3])
    commands.append(entry)
commands.remove(["0",0,"0",0])
for x in commands: #x[0]=move or pile; x[1]=a; x[2]=onto or over; x[3]=b
    for a in range(24): #find stack of a such that stack[pos_a]
        if x[1] in stack[a]:
            pos_a = a
            break
```

```
    index_a = stack[pos_a].index(x[1])

    for b in range(24): #find stack of b such that stack[pos_b]
        if x[3] in stack[b]:
            pos_b = b
            break
    index_b = stack[pos_b].index(x[3])

    if stack[pos_b] == stack[pos_a] or x[1] == x[3]:
        continue

    if x[0] == "move" and x[2] == "onto":
        for i in range(index_a +1, len(stack[pos_a])): #balik sa orig mga nasa to
p ni a
            stack[stack[pos_a][i]].append(stack[pos_a][i]) #ang ginawa is dinagda
g lang sa orig index which might be wrong na dapat palitan yung buong stack

        for j in range(index_b +1, len(stack[pos_b])): #balik sa orig mga nasa to
p ni b
            stack[stack[pos_b][j]].append(stack[pos_b][j])

        stack[pos_b] = stack[pos_b][:index_b+1] #remove sa stack b yung mga nasa
top ni b
        stack[pos_b].append(x[1]) #put a on stack of b
        stack[pos_a] = stack[pos_a][:index_a] #remove sa stack a si a and mga nas
a top

    elif x[0] == "move" and x[2] == "over":
        for i in range(index_a +1, len(stack[pos_a])): #balik sa orig mga nasa to
p ni a
            stack[stack[pos_a][i]].append(stack[pos_a][i])

        stack[pos_b].append(x[1]) #put a on stack of b
        stack[pos_a] = stack[pos_a][:index_a] #remove sa stack a si a and mga nas
a top

    elif x[0] == "pile" and x[2] == "onto":
        pile = [stack[pos_a][k] for k in range(index_a, len(stack[pos_a]))]

        for j in range(index_b +1, len(stack[pos_b])): #balik sa orig mga nasa to
p ni b
            stack[stack[pos_b][j]].append(stack[pos_b][j])

        stack[pos_b] = stack[pos_b][:index_b+1] #remove sa stack b yung mga nasa
top ni b
```

```
        for y in pile: #dump the pile on stack b
            stack[pos_b].append(y)
        stack[pos_a] = stack[pos_a][:index_a] #remove sa stack a si a and mga nas
a top

    elif x[0] == "pile" and x[2] == "over":
        pile = [stack[pos_a][k] for k in range(index_a, len(stack[pos_a]))]
        for y in pile: #dump the pile on stack b
            stack[pos_b].append(y)

        stack[pos_a] = stack[pos_a][:index_a] #remove sa stack a si a and mga nas
a top

for index,element in enumerate(stack):
    print(f'{index}: {" ".join(str(x) for x in element)}')
```

In turn, the output I got from the input is as follows:

```
0:
1: 1 16 3 19
2: 2
3:
4: 4 17
5: 5
6:
7: 7 8
8:
9: 9
10: 10 15
11: 11
12: 12
13:
14: 14 13 6
15:
16:
17:
18: 18
19:
20: 20
21: 21
22: 22 0
23: 23
24: 24
```

a. The topmost block in position 1 is **19**.
b. The topmost block in position 4 is **17**.
c. The topmost block in position 10 is **15**.
d. The topmost block in position 14 is **6**.
e. The topmost block in position 22 is **0**.
f. The topmost block in position 9 is **9**.

5. The code that I used to minimize the overlapping time intervals is as follows:

```
from operator import itemgetter
N = int(input())
intervals = []
merged = []
for j in range(N):
    time = (input().split())
    time = [int(i) for i in time]
    intervals.append(time)
intervals = sorted(intervals, key = itemgetter(0,1))
for time in intervals:
    if not merged:
        merged.append(time)
    else:
        comparison = merged[-1]
        if time[0] <= comparison[1]:
            end = max(comparison[1], time[1])
            merged[-1] = (comparison[0], end)
        else:
            merged.append(time)
minimized = [item for sublist in merged for item in sublist]
print(minimized) #the minimized time interval (no overlapping)
```

a. largest time interval when the machine is idle (is not processing any job).

   When we input the test cases, the minimized time interval of work is only [10, 88). This means that the machine is working from start time to end time and that there is no idle time. We can denote the **idle time as [10, 10)**.

b. largest time interval when the machine is NOT idle.

   Following the result of the code, the minimized time interval of work is also the largest time interval when the machine is **not idle denoted as [10, 88)**.

c. time interval with the most number of jobs concurrently being processed

   In order to get the time interval, I modified my code as follows:

```
N = int(input())
times = []
counter = {}
for j in range(N):
    start, end = (input().split())
    start = int(start)
    end = int(end)
    for i in range(start, end):
        times.append(i)
for time in times:
    if time in counter:
        counter[time] += 1
    else:
        counter[time] = 1
counter = {k: v for k, v in sorted(counter.items(), reverse=True, key=lambda item
: item[1])}
print(counter)
```

To get the time interval with the most occurrences, I created this code that will count the number of occurrences of each time within the intervals. Clipping the output, the times with the most number of occurrences are as follows: {31: 4, 32: 4, 33: 4, 34: 4, 25: 4, 26: 4}

The smallest start time is 25 while the smallest end time is 26 so the answer is **[25, 26)**.