CS 3340 Assignment 1
Student Number: 250757782
Name: Linsheng Ding

Question 1:

- Given that we know i(i + 1) = [(i + 1)^3 − i^3 − 1]/3 and that Σ i^3 = [(n*(n+1))/2]^2 as well as Σ c = cn, we can then use these to figure out the summation formula for Σ i(i+1).

  We first convert Σ i(i+1) to Σ [(i + 1)^3 − i^3 − 1]/3.

  Then we can split up each of these into 3 parts

  [Σ(i + 1)^3 − Σ i^3 − Σ 1]/3

  And get:

  ([((n+1)*(n+2))/2]^2 - [((n)*(n+1)/2)]^2 - n) /3

  Which is our summation formula.

Question 2:

Number of leaves in complete binary tree proof:
We can use proof by induction to prove the number of leaves of a complete binary tree is 2^h.

Inductive base: n = 0 for when tree only has root node

2^(0) = 1 which is true as there is only one root node that is also the leaf

Inductive hypothesis: For n>=0, a complete binary tree has 2^h leaves where the symbol h represents height.

Inductive step: Suppose a complete binary tree T has height k+1 and that complete binary trees have 2^h leaves which means that that it should 2^(k+1) leaves.
This can be shown because a complete binary tree with k+1 height is the same as saying that you add two nodes to each node of layer k, and thus adding the number of leaves would be equivalent to 2 * 2^k which is 2^(k+1) and thus proving our hypothesis.

Size of complete binary tree proof:

We can use proof by induction to prove size of a complete binary tree is 2^(h+1) − 1.

Inductive base: n = 0 for when the tree only has a root

Plugging in 0 for height, we get:

2^(0+1) - 1 = 2^1 - 1 = 2 -1 = 1 which satisfies our base case where there is only one root node.

Inductive Hypothesis: Suppose for n>=0, a complete binary tree has size 2^(h+1) − 1 where the symbol h represents height.

Inductive step:
Lets use a complete binary tree T with height k+1 so that we get 2^((k+1)+1) - 1 which is 2^(k+2) - 1. We can also re-use what we had from above which is that there is 2^k leaf nodes and using our inductive assumptions we would have for before which is that a complete binary tree would have 2^(k+1) - 1 nodes for the kth layer and then adding our 2^k leaves would give us 2^(k+1) + 2^(k) - 1 which is the same as saying 2^(k+2) - 1 which proves our hypothesis.

Question 3:

Constant time:
2^100

O(log log(n))
log(log(n))

O(logn)
√log n
log^2(n)

O(n)
5n

O(nlogn)
2n log2(n)
6n log n
n log4 n

O(n^x)
n^0.01
1/n

√n
3n^0.5
4n^(3/2)

O((n^2)logn)
n^2 log(n)

O(n^3)
n^3

O(x^(logn))
2^(log n)
4^(log n)

O(x^n)
2^n
4^n

O(x^(x^n))
2^(2^n)

Question 4:

Pick first index for two variables min and max. Iterate through the n numbers, to check if current index is greater than current index + 1, if it is compare max with greater, and store it in max if greater and if smaller, then compare min with lesser and store it in min if it is less. Essentially would would have done at most 3 comparisons per number before finishing the array and satisfying our requirement of 3n/2 comparisons. This would terminate correctly because it would stop once array iteration is done

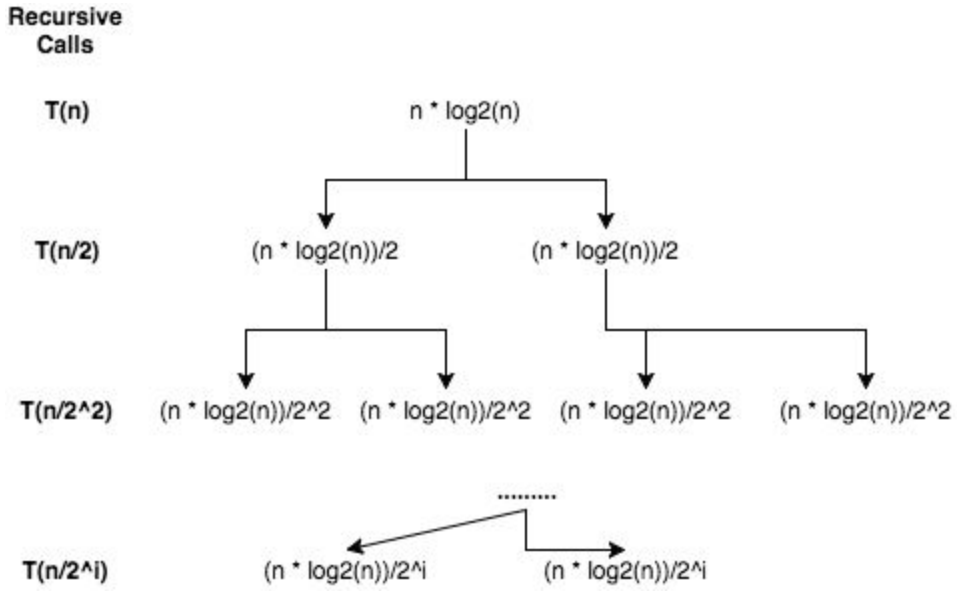
Question 5:

The method would take in an array of n-2, as well an result array of size two would be created to store missing integers, and a result array counter starting at 0 is also created to make sure we're storing the missing integers in the right place. A counter starting at 1 would also be created, you would iterate through the array and check if the counter matches array element, if not, you put the counter number into the result array element referenced by the result array counter and then increment result array counter. If result array counter equals 3, you break out of the loop iterating through the n-2 array and return your result array. This is in constant space and O(n) time because in the worst case you will iterate through the whole array but only uses a constant (2 integers array) space to store your result.

Question 6:

T(2) = 1
T(n) ≤ 2T(n/2) + n log2 (n)

Using the recurrence tree method below:



We can see that everytime we are basically getting n*log2(n) for each row of the tree. Furthermore this keeps going until we hit our base case where we get T(2) = 1 and thus 2 = n/2^i, which using below working out:

- 2 = n/2^i
- n = 4^i
- log4(n) = i

Thus the method terminates at T(2) or log4(n) and we know that we're essentially summing up all the row sums like below:

$$\sum_{i=0}^{\log_4(n)} n*\log_2(n)$$

Which is the same as :

$$n*\log_2(n) * \sum_{i=0}^{\log_4(n)} 1$$

Which is equal to n*log2(n) * log4(n), and then to clean this up, we should see that

$$T(n) <= n*\log_2(n) * \sum_{i=0}^{\infty} 1$$

Then T(n) <= n*log2(n) which means that time complexity is O(nlogn).

To prove this we can substitute a value:

Lets say T(n) = 2T(n/2) + nlog2(n)
Which is less than or equal to an equation called:
<= 2c * nlog(n/2) + nlog2(n)
= nlogn(2c + 1)
Which is less than nlogn(2c + 2) and thus showing that we can always find an equation of O(nlogn) that is either greater or equal to T(n), proving it is O(nlogn).

Question 7:

c)

For Question 7a, the algorithm took 9.042s while the second algorithm for question 7b took 0.134s thus showing that there was less time being used for second algorithm as it had time complexity of O(n).

d)

- I can't use my program in 7a) to compute F(50) because the call stack can't handle that many recursive calls
- My answer to 7b can compute F(300) precisely and fast because Python3 automatically switches to long or other data types able to hold long integers once it goes past the max size of an integer.

```python
#   Recursive Fibonacci function
def recursiveFib(num):
    return fibHelper(0,1,num)

# Fibonacci helper method
def fibHelper(firstNum, secondNum, num):
    if (num == 0):
        return firstNum
    return fibHelper(secondNum, firstNum+secondNum, num - 1)

if __name__ == "__main__":
    # Handles 7b)
#   for i in range(0,31):
#       print(recursiveFib(i*10))
    print(recursiveFib(300))
```

latex, not handwritten and then scanned

Desktop — -bash — 80×24

```
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
  File "asn1_a.py", line 8, in recursiveFib
    return recursiveFib(num - 1) + recursiveFib(num - 2)
KeyboardInterrupt
Euns-MacBook-Air:Desktop mding$ python3 asn1_b.py
222232244629420445529739893461909967206666939096499764990979600
Euns-MacBook-Air:Desktop mding$ who am i
mding    ttys000  Feb  1 16:48
Euns-MacBook-Air:Desktop mding$ open asn1_b.py
Euns-MacBook-Air:Desktop mding$ python3 asn1_b.py
222232244629420445529739893461909967206666939096499764990979600
Euns-MacBook-Air:Desktop mding$
```