Dylan Sapienza

2.1.1

The run time of the program us nLog(n) because there is an outer while loop that increments by i = i * 2. (log n). The inner loop continues j<n and j stats at 0 and increments by 1, so if we combine these two we arrive at n log n. O(n log n)

2.1.2

A for loop from 0 - n iterating by i encapsulates the rest of the program, so we know at least n. Then inside we have an if statement that only runs when i is a multiple of three. If it is, it goes for a while loop that erases count back to zero. This means every three loops, the while loop loops back to zero. This would mean it has a linear growth n. O(n)

2.2.1

Using the worst case scenario analysis every element (n) in an array would be compared to n - 1 elements in the array. Therefore if n elements are each compared n - 1 times, it would be n * n. O(n^2)

2.2.2

Binary Search works by taking a middle guess and then if incorrect splitting up the data in order to remove all the incorrect numbers whether the guess was a higher or lower. This occurs over and over until either the number is guessed or the number of numbers still available is 1. This sort of halving operations models the function O(log n).

2.2.3

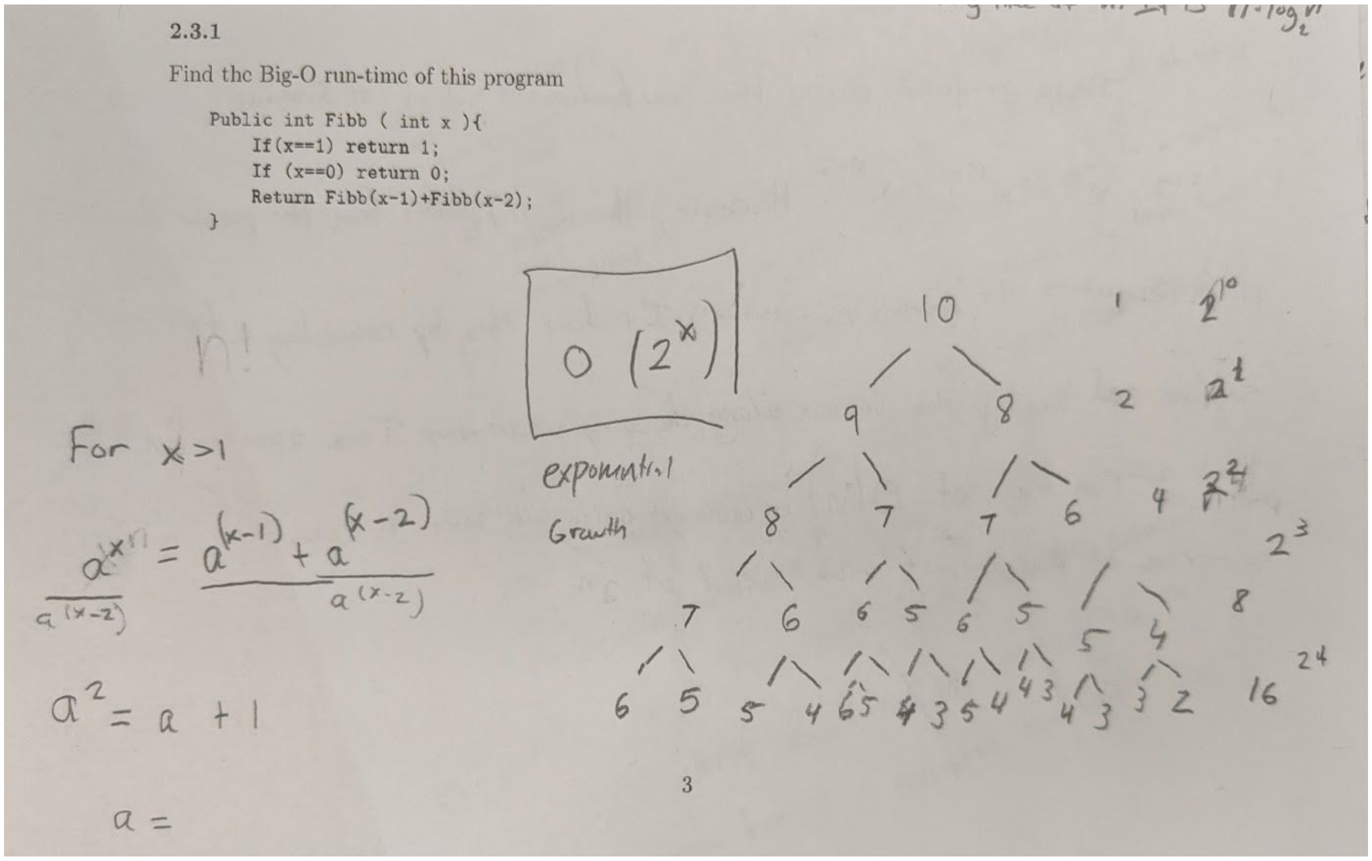
Worst Case: O(n^2)

If the quick sort pivot is consistently very poor it will create two new arrays one of size 1 and one of size n -1. Therefore it would need to perform this splitting and sorting technique of n, n times giving us n*n.

Average Case: O(n log n)

If pivots are somewhat efficient in their choosing we will see, like in binary search, a near halving of new subproblems. Which we can say is log n. Therefore since we also need to account for the sorting time of n, it results in n log n.

2.3.1



2.3.1

Find the Big-O run-time of this program

```
Public int Fibb ( int x ){
     If(x==1) return 1;
     If (x==0) return 0;
     Return Fibb(x-1)+Fibb(x-2);
}
```

$n!$

For $x>1$

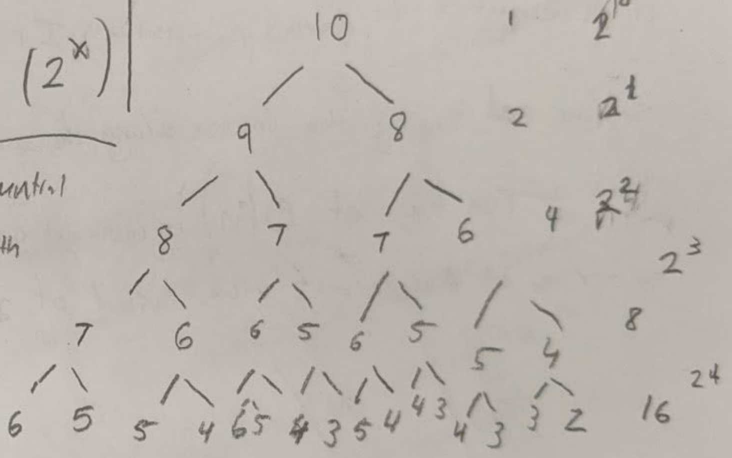
$$\frac{a^{x}}{a^{(x-2)}} = \frac{a^{(x-1)} + a^{(x-2)}}{a^{(x-2)}}$$

$$a^2 = a + 1$$

$a =$

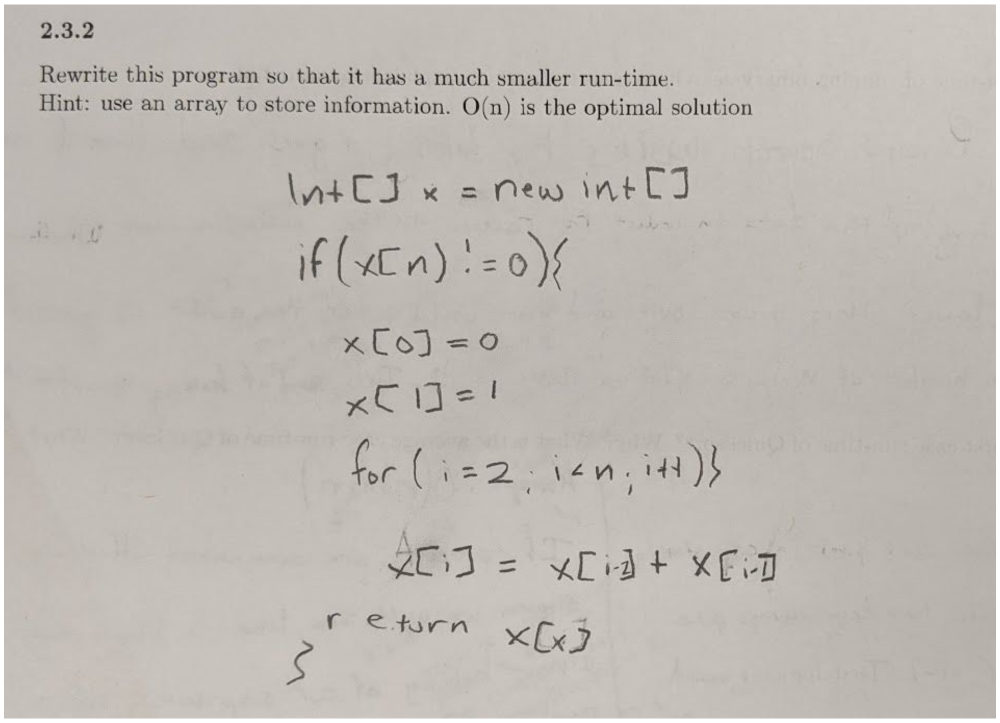
$$\boxed{O(2^x)}$$

exponential Growth



3

2.3.2



2.3.2

Rewrite this program so that it has a much smaller run-time.
Hint: use an array to store information. O(n) is the optimal solution

```
Int[] x = new int[]
if(x[n] != 0){
   x[0] = 0
   x[1] = 1
   for ( i=2; i<n; i++)}
      x[i] = x[i-2] + x[i-1]
  return x[x]
}
```

2.3.3

This program carries the same fundamental ideas of fibonacci using x^n = x^n-1 + x^n-2. However, the way it differs from the previous solution is it remembers its previous computations. It does this by calculating a Fibonacci sequence and saving all the values along the way in an array. This approach utilized has a run time of O(n) because it only will take n calculations in order to arrive at the correct value instead of the previous 2^n.