Andrew DiBella
Homework 1

1)

```
1) int fib(int n){
2)     if (n<0) return 0;
3)     else if (n <4 ) return 1;
4)     else return fib(n-1) + fib(n-2) + fib(n-3);
5)   }
```

This extended version of the fibonacci sequence uses recursion to add the previous 3 values to find the sum of each nth value in the sequence. The code is very simple, if n is less than 4 which satisfies all three base cases then it will just return 1. Otherwise, it will return the sum of the previous three digits and once it reaches the end of the recursive call it adds each value returned by the function and returns the nth term in the sequence. Any negative inputs will result in a special base case of zero since there is no nth term. This program runs very quickly for small input but large inputs for n will take much longer to process.

2) Considering the solution is only 5 lines and works correctly according to the requirements of the fibonacci sequence and also the 4 rules of recursion so I believe it to be a good solution. However, I am not sure if I interpreted the non positive inputs correctly, but since if the number of terms is negative then there is none at all so return 0.

3) 2.1:
2/N
< root(N)
< N
< NloglogN
< NlogN | These two have
< Nlog(N^2) | the same growth rate
< Nlog^2(N)
< N^1.5
< N^2
< N^2logN
< N^3
< 2^(N/2)
< 2^N

4) 2.6:

A) Based on the sequence of terms and how the fine is constructed it seems that the closed form for this is 2 raised to the previous fine day:

2^2^(N-1)

B)

D = 2^2^(N-1)
log2(D) = log(2^2^(N-1))
logD = 2^(N-1)
loglogD = N-1

T(N) = O(loglogN)

5)

#1)

```
int n = 10000000;
long start = System.nanoTime();
                                    // COST    TIMES
int sum = 0;                        // c1      1
for (int i = 0;                     // c2      1
                i < n;              // c3      n+1
                        i++) {      // c4      n
  sum++;                            // c5      n
}
System.out.println(System.nanoTime()- start);
```

T(N) = O(N)

N	Time nano seconds
100	3300
1,000	8300
10,000	65600
100,000	595000
1,000,000	1277200

Since bigOh is O(N) you can clearly see the linear progression as N increases.
T(N) = c1*1 +c2*1 +c3*(n+1) + c4*n+ c5*n

#2)

```
long start = System.nanoTime();
                                  // COST    TIMES
int sum = 0;                      // c1      1
for (int i = 0;                   // c2      1
                i < n;            // c3      n+1
                        i++) {    // c4      n
    for (int j =0;                // c5      n
                 j<n;             // c6      n^2 +1
                     j++){        // c7      n^2
      sum++;                      // c8      n^2
    }
}
System.out.println(System.nanoTime()- start);
```

T(N) = O(N^2)
T(N) = c1*1 + c2*1 + c3*(n+1) + c4*n + c5*n + (c6 * n^2 +1) + c7 *n^2 + c8 *n^2

N	Time nano seconds
100	68000
1,000	2564200
10,000	3579900
100,000	4995700
1,000,000	4493300

You can see how since this is a quadratic equation n^2 is works much faster with smaller inputs of n like n=100 but quickly increases in time with n=1,000.

#3)

```
long start = System.nanoTime();
                                  // COST    TIMES
int sum = 0;                      // c1      1
for (int i = 0;                   // c2      1
                i < n;            // c3      n+1
                        i++) {    // c4      n
    for (int j =0;                // c5      n
                 j<n*n;           // c6      n^3 +1
                     j++){        // c7      n^3
      sum++;                      // c5      n^3
    }
}
System.out.println(System.nanoTime()- start);
```

T(N) = O(N^3)
T(N) = c1*1 + c2*1 + c3*(n+1) + c4*n + c5*n + (c6 * n^3 +1) + c7 *n^3 + c8 *n^3

N	Time nano seconds
100	2703600
1,000	4646600
10,000	4844900
100,000	4959000
1,000,000	2045100

This function increasingly takes longer as input n gets larger. However for n=1,000,000 it gets faster but then for n=2,000,000 it returns to becoming increasingly longer time for T(N).

#4)

```
long start = System.nanoTime();
                                    // COST    TIMES
int sum = 0;                        // c1      1
for (int i = 0;                     // c2      1
                i < n;              // c3      n+1
                        i++) {      // c4      n
    for (int j =0;                  // c5      n
                 j<i;               // c6      n*(n+1)/2 + 1
                     j++){          // c7      n*(n+1)/2
      sum++;                        // c5      n*(n+1)/2
    }
}
System.out.println(System.nanoTime()- start);
```

T(N) = O(N^2)
T(N) = c1*1 + c2*1 + c3*(n+1) + c4*n + c5*n + (c6 * n(n +1)/2+1) + (c7 * n(n +1)/2)+ c8 * n(n +1)/2

N	Time nano seconds
100	34100
1,000	1428200
10,000	2109700
100,000	2139400
1,000,000	3322300

Similar to the other n^2 this algorithm gets slower as input gets greater. This algorithm is faster than the 2 previous exponential functions.

#5)

```
    long start = System.nanoTime();
                                    // COST    TIMES
    int sum = 0;                    // c1      1
    for (int i = 0;                 // c2      1
                  i < n;            // c3      n+1
                          i++) {    // c4      n
      for (int j = 0;               // c5      n
                  j<i*i;            // c6      (not required)
                      j++){         // c7
        for (int k = 0;             // c8
                      k<j;          // c9
                          k++ ){    // c10
          sum++;                    // c11

        }
      }
    }
    System.out.println(System.nanoTime()- start);
  }
```

T(N) = O(N^5)

N	Time nano seconds
100	2506800
1,000	263478100
10,000	266737926800

This program takes incredibly long for large inputs. Actually frustrating waiting for it to complete.

#6)

```
long start = System.nanoTime();
                                  // COST    TIMES
int sum = 0;                      // c1      1
for (int i = 1;                   // c2      1
                i < n;            // c3      n+1
                       i++) {     // c4      n
  for (int j = 1;                 // c5      n
              j<i*i;              // c6      (not required)
                  j++){           // c7
    if(j%i == 0)
    for (int k = 0;               // c8
                  k<j;            // c9
                       k++ ){     // c10
      sum++;                      // c11

    }
  }
}
System.out.println(System.nanoTime()- start);
```

T(N) = O(N^5)

The if statement will run n^3 times but the k-for loop will run N^5 / 2 times since j and i start at 1 it will run half the time depending on the current value of the integers i and j.

5p2)

a) f(n) = log(n), g(n) = log2(n)
   f(n) = o(g(n))

b) f(n) = 3n, g(n) = 2n
   f(n) = \BigTheta(g(n))

c) f(n) = log(n), g(n) = n
   g(n) = o(f(n))

d) f(n) = 5n, g(n) = n
   g(n) = o(f(n)

6)
Since the running time for a permutation of an array is T(N) = O(N^2) and the algorithm runs n times f(n)*g(n) = O(f(n)g(n) so the overall big oh is T(N) = O(N^3).

7) a) Addition can be written as a program that adds each digit and holds a carry if necessary and calculates the overall sum. If this were to be 3 digit addition the number of additions required is 3 or N so T(N) = O(N).

b) Multiplication requires that each digit of each integer is multiplied. If this were 3 digit multiplication xxx * xxx then the number of multiplications performed would be 9. Therefore, T(N) = O(N^2).