1)

```
public int longestCommonSubsequence(String firstText, String secondText){  1 usage  jonathanp2004

    int a = firstText.length();  O(1)
    int b = secondText.length(); O(1)

    int [][] table = new int[a +1][b + 1]; O(1)

    for(int i = 1; i <= a; i++){  O(n) ← depends on the  a = firstText
        for(int j = 1; j <= b; j++ )  ← runs b times x a
        {
            if(firstText.charAt(i-1) == secondText.charAt(j-1))  O(1)
            {
                table [i][j] = table[i-1][j-1]+1; O(1)
            } else{
                table [i][j] = Math.max(table[i-1][j], table[i][j-1]);  O(1)
            }
        }
    }

    return table[a][b]; O(1)
}
```

The time complexity of this algorithm would be $O(n^2)$ with n being the length of the input array. There both have a time complexity of O(n) and since the inner loop is iterated b times x a times you multiply both getting you $O(n^2)$

The space complexity is $\Omega(n^2)$ because the int j is initialized a times, so the amount of initialization depends on the n = a.length

2)

```
public String longestCommonSubstring(String fText, String sText){  1 usage  jonathanp2004

    int a = fText.length();  O(1)
    int b = sText.length();  O(1)

    int[][] ta = new int[a+1][b+1];  O(1)

    int maxLength = 0;  O(1)
    int endIndex= 0;  O(1)

    for(int i =1; i <= a; i++){  O(n)
        for(int j = 1; j <= b; j++){  O(n)

            if(fText.charAt(i - 1) == sText.charAt(j - 1))
            {
                ta[i][j] = ta[i-1][j-1]+1;

                if(ta[i][j] > maxLength)
                {
                    maxLength = ta[i][j];
                    endIndex = i;
                } else {
                    ta[i][j] = 0;
                }
            }
        }
        if(maxLength == 0) {  O(1)
            return "";  O(1)
        }
    }  return fText.substring(endIndex - maxLength, endIndex);  O(1)
}
```

O(1)

$O(n^2)$

The time complexity of this algorithm is $O(n^2)$
Because the inner loop is itterated through b times x a again similar to the subsequence algorithm.

The space complexity is $\Omega(n^2)$ because the int j is initialized a times.

3)

```java
Scanner scan = new Scanner(System.in);

System.out.print("Sequence Amount: ");
int n = scan.nextInt();

if(n<= 0)
{
    System.out.println("Enter Positive");
    return;
}
long[] sq = new long[n];
//Initial value of 0,1
sq[0] = 0;
if(n>1)
{

    sq[1] = 2;
}
for(int i = 2; i < n; i++)
{
    //same calc as show in the assignment
    double calc = (1.5* sq[i-1]) + (2* sq[i - 2]);

    sq[i] = (long) Math.floor(calc);
}

System.out.println("Input: " + n);

System.out.print("Output: ");
for(int i = 0; i < n; i++)
{
    System.out.print(sq[i]);
    if(i < n -1)
    {
        System.out.print(", ");
    }
}

System.out.println();
```

O(n-2) = O(n)

O(1)

O(1)

The time complexity is O(n) where the n is the length of the sequence, because the loop run n times.

The space complexity is constant because the total memory created stays the same regardless of the n.

```java
Scanner scan = new Scanner(System.in);

System.out.print("Enter Number: ");

long x = scan.nextLong();

if (x < 0){
    System.out.println("Output: -1");
    return;
}
if(x == 0){
    System.out.println("Output: 0");
}
if(x == 2){
    System.out.println("Output: 1");
}

long nm1 = 2;
long nm2 = 0;
int index = 1;

while(true){

    double calc = (1.5 * nm1) + (2*nm2);
    long c = (long)Math.floor(calc);
    index++;

    if(c == x){
        System.out.println("Input: "+ x);
        System.out.println("Output: " + index);
        return;
    }
    if(c > x){
        System.out.println("Input: "+ x);
        System.out.println("Output: " + (index - 1));
        return;
    }

    nm2 = nm1;
    nm1 = c;
```

goes on depending on x

O(1)

O(1)

O(1)

O(1)

O(1)

The time complexity for this algorithm is O(n) where n depends on the value user inputs

The space complexity is Ω(1) because again the created variables don't change.

(5)

```java
public class RemoveElements {  jonathanp2004

    public static int removeElements(double[] nums, double val){  1 usage  jonathanp2004
        int k =0;

        for(int i =0; i < nums.length; i++){  O(n)
            if(nums[i] >= val){    O(1)
                nums[k] = nums[i];  O(1)
                k++;          O(1)
            }
        }
        return k;     O(1)
    }

    public static void main(String[] args){  jonathanp2004

        double[] nums = {3.5, 1.2, 2.1, 6.7,9.0,4.4};
        double val = 3.0;

        System.out.print("Input: nums = {" );
        for(int i = 0; i < nums.length; i++){
            System.out.print(nums[i]);
            System.out.print(",");
        }
        System.out.println("},  val =  " + val);

        int s = removeElements(nums,val);

        System.out.println("Output = " + s);
        System.out.print("Modified array: {");
        for(int i = 0; i < s; i++){
            System.out.print(nums[i]);
            System.out.print(",");
        }
        System.out.println("}");

    }
}
```

The time complexity of this would be O(n) because the algorithm run a total of n time where n is the length of nums.

The space complexity is O(1) because there are no extra variable being created.