Tarea 2 Juan Diego Rojas

Punto 1

```
#include <stdio.h>
void algoritmo1(int n){
    int i, j = 1;                       //1 vez
    for(i = n * n; i > 0; i = i / 2){   //lg(n*n)+2 veces
        int suma = i + j;               //lg(n*n)+1 veces
        printf("Suma %d\n", suma);      //lg(n*n)+1 veces
        ++j;                            //lg(n*n)+1 veces
    }
    /*Número de veces que se ejecuta: 1+lg(n*n)+2+lg(n*n)+1+lg(n*n)+1+lg(n*n)+1 =
6+4lg(n*n)
    O(lg(n^2))
    Si ejecutamos algoritmo1(8) se obtiene el siguiente retorno:
    Suma 65 (porque i=64 y j=1 y 64+1=65)
    Suma 34 (porque i=32 y j=2 y 32+2=34)
    Suma 19 (porque i=16 y j=3 y 16+3=19)
    Suma 12 (porque i=8 y j=4 y 8+4=12)
    Suma 9 (porque i=4 y j=5 y 4+5=9)
    Suma 8 (porque i=2 y j=6 y 2+6=8)
    Suma 8 (porque i=1 y j=7 y 1+7=8)
    Y el número de veces que se repite cada línea es 30, de acuerdo con lo
establecido en el cálculo de la complejidad:
    1 + lg(64)+2 + lg(64)+1 + lg(64)+1+ lg(64)+1
    1 + 6 + 2 + 6 + 1 + 6 + 1 + 6 + 1 = 30
    */
}
```

Punto 2

```
int algoritmo2(int n){
    int res = 1, i, j;                  //1 vez
    for(i = 1; i <= 2 * n; i += 4){     //((2*n)/4)+1 veces si n es par, si n es
impar, ((2*n)/4)+2 (tomándose solamente la parte entera, como si se usara la
función floor() de Python)
        for(j = 1; j * j <= n; j++){    //((2*n)/4)(sqrt(n))+1 veces si n es par,
si n es impar (((2*n)/4)+1)(sqrt(n)) (tomándose solamente la parte entera, tanto
para la división como para la raíz si son inexactas, como si se usara la función
floor() de Python)
            res += 2;                   //((2*n)/4)(sqrt(n)) veces si n es par,
si n es impar (((2*n)/4)+1)(sqrt(n)) (tomándose solamente la parte entera, tanto
para la división como para la raíz si son inexactas, como si se usara la función
floor() de Python)
            printf("res %d\n",res);     //((2*n)/4)(sqrt(n)) veces si n es par,
si n es impar (((2*n)/4)+1)(sqrt(n)) (tomándose solamente la parte entera, tanto
```

```
para la división como para la raíz si son inexactas, como si se usara la función
floor() de Python)
        }
    }
    return res;                         //1 vez
    /*Número de veces que se ejecuta:
1+((2*n)/4)+1+((2*n)/4)(sqrt(n))+1+((2*n)/4)(sqrt(n))+((2*n)/4)(sqrt(n))+1
    O()
    Si ejecutamos algoritmo2(8) se obtiene el siguiente retorno:
    res 3 (porque se suma 2 a res cuando i=1 y j=1)
    res 5 (porque se suma 2 a res cuando i=1 y j=2)
    res 7 (porque se suma 2 a res cuando i=5 y j=1)
    res 9 (porque se suma 2 a res cuando i=5 y j=2)
    res 11 (porque se suma 2 a res cuando i=9 y j=1)
    res 13 (porque se suma 2 a res cuando i=9 y j=2)
    res 15 (porque se suma 2 a res cuando i=13 y j=1)
    res 17 (porque se suma 2 a res cuando i=13 y j=2)
    Y el número de veces que se repite cada línea es 32, de acuerdo con lo
establecido en el cálculo de la complejidad:
    1 + ((2*8)/4)+1 + ((2*8)/4) floor(sqrt(8))+1 + ((2*8)/4) floor(sqrt(8)) +
((2*8)/4) floor(sqrt(8)) + 1
    1 + (16/4)+1 + (16/4)(2)+1 + (16/4)(2) + (16/4)(2) + 1
    1 + 4 + 1 + 8 + 1 + 8 + 8 + 1 = 32
    */
}
```

Punto 3

```
void algoritmo3(int n){
    int i, j, k;                            //1 vez
    for(i = n; i > 1; i--){                 //n veces
        for(j = 1; j <= n; j++){            //(n-1)(n+1)
            for(k = 1; k <= i; k++) {       //sumatoria i=0 -> n
(i+2)
                printf("Vida cruel!!\n");   //sumatoria i=0 -> n (i+1)
            }
        }
    }
    /*La complejidad sería de O(n^3)*/
}
```

Punto 4

```
int algoritmo4(int* valores, int n){
    int suma = 0, contador = 0;             //1 vez
    int i, j, h, flag;                      //1 vez
    for(i = 0; i < n; i++){                 //n+1 veces
        j = i + 1;                          //n veces
```

```
        flag = 0;                               //n veces
        while(j < n && flag == 0){              //mejor caso: 1 vez ; peor caso: n*n
            if(valores[i] < valores[j]){        //mejor caso: 0 veces ; peor caso:
n*(n-1)
                for(h = j; h < n; h++){         //mejor caso: 0 veces ; mejor caso
del peor caso: 0 veces; peor caso del peor caso: n*(n-1)*(sumatoria i=j -> n-1
(i+2))
                    suma += valores[i];         //mejor caso: 0 veces ; mejor caso
del peor caso: 0 veces; peor caso del peor caso: n*(n-1)*(sumatoria i=j -> n-1
(i+1))
                }
            }
            else{                               //mejor caso: 0 veces ; peor caso:
n*(n-1)
                contador++;                     //mejor caso: 0 veces ; mejor caso
del peor caso: 0 veces; peor caso del peor caso: n*(n-1)
                flag = 1;                       //mejor caso: 0 veces ; mejor caso
del peor caso: 0 veces; peor caso del peor caso: n*(n-1)
            }
            ++j;                                //mejor caso: 0 veces ; peor caso:
n*(n-1)
        }
    }
    return contador;                            //1 vez
    /*El algoritmo regresa "1" si existe algún valor en un array que sea mayor
que el valor inmediatamente siguiente, hasta el elemento n del array, de lo
contrario devuelve "0".
    El parámetros valores de tipo int*, corresponde a una dirección de memoria
(puntero) de un array, y el parámetro n corresponde al índice del valor máximo
que se va a estudiar.
    La variable suma va sumando los valores que no cumplen la condición de ser
mayores que el valor siguiente.
    La variable contador va contando la cantidad de veces que dicha condición se
cumple, aunque solo podrá alcanzar hasta el valor "1", por lo que dice el
algoritmo. Ese es el valor que se retorna.
    La variable i es el índice del elemento que se estudia en el momento, y la
variable j es el índice del elemento siguiente. La variable cumple la función de
tomar el valor actual de j para inicicar un ciclo for, que posteriormente
modifique el valor de la variable suma.
    La variable flag cumple la función de lo que se conoce como un break, cuyo
propósito es terminar el ciclo while cuando la condición se cumpla.*/
}
```

Punto 5

```
void algoritmo5(int n){
```

```
    int i = 0;              //1 vez
    while(i <= n){              //(n/5)+2 veces
        printf("i: %d\n", i);   //(n/5)+1 veces
        i += n / 5;             //(n/5)+1 veces
    }
    //Complejidad: O(n)
}
```

Punto 6

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0.0024776458740234375	35	2235.0758221149445
10	0.018949508666992188	40	NA
15	0.16620540618896484	45	NA
20	2.0176360607147217	50	NA
25	19.415305376052856	60	NA
30	243.52265048027039	100	NA

El valor más alto al que llegué fue n=35. Como se aprecia en la tabla, cuando n es igual a 35, el tiempo de ejecución fue de 2235.08 segundos aproximadamente, lo que son más o menos 37 minutos. Pude haber continuado con los otros datos, pero notando que los tiempos de ejecución aumentan exponencialmente, por el bien de mi máquina, decidí detenerme en el 35 (solo en el 35 y el computador ya sonaba como un helicóptero). Como dije anteriormente, esperaría que la complejidad de este algoritmo sea exponencial, debido a la alta tasa de crecimiento en el tiempo entre un dato de entrada y el siguiente. Se pasó de 2 segundos a más de media hora habiendo aumentado n en 15 unidades solamente.

Punto 7

Tomando el siguiente algoritmo:

```
def Fibonacci(n):
    a=0                 #1
    b=1                 #1
    c=a+b               #1
    cont=0              #1
    while (cont<n-2):   #peor caso: n-1 / mejor caso: 1
        a=b             #peor caso: n-2 / mejor caso: 0
        b=c             #peor caso: n-2 / mejor caso: 0
        c=a+b           #peor caso: n-2 / mejor caso: 0
        cont=cont+1     #peor caso: n-2 / mejor caso: 0
    print(c)            #1
```

La complejidad sería O(n), es decir que tiene complejidad lineal.

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0.0005953311920166016	45	0.0007681846618652344
10	0.0004820823669433594	50	0.0006237030029296875

15	0.000492095947265625	100	0.0005345344543457031
20	0.0005295276641845703	200	0.0005753040313720703
25	0.0007479190826416016	500	0.0006170272827148438
30	0.0009975433349609375	1000	0.0008785724639892578
35	0.0009992122650146484	5000	0.0015954971313476562
40	0.0005557537078857422	10000	0.004967689514160156

Punto 8

Tamaño Entrada	Solución Propia	Solución Profes
100	0.0076143741607666016	0.004994869232177734
1000	0.0715939998626709	0.03273272514343262
5000	1.1760194301605225	0.149705171585083
10000	3.8347535133361816	0.25861454010009766
50000	84.92227005958557	1.159726619720459
100000	336.12253856658936	2.0680909156799316
200000	1365.3037271499634	3.674633264541626

a) Son bastante diferentes los tiempos de ejecución. Al principio son relativamente parecidos, pero a partir de n = 50000, el tiempo de ejecución de mi solución es muchísimo más grande que el de la solución de los profes. Esto significa que mi solución es mucho menos eficiente, y que seguramente tiene una complejidad mucho mayor.