Ejercicios de Complejidad Teórica

1.

```
void algoritmo1(int n){
    int i, j = 1; //2 Ejecuciones (Crea dos variables)
    for(i = n * n; i > 0; i = i / 2){ //Este ciclo se evalua Log2(n^2)+2
        int suma = i + j; //Esta linea se ejecuta Log2(n^2)+1
        printf("Suma %d\n", suma); //Esta linea se ejecuta Log2(n^2)+1
        ++j; //Esta linea se ejecuta Log2(n^2)+1
    }
}
```

Es asi ya que se toma el n^2 y se divide por 2 una cantidad k de veces (dependiendo de n) y sabiendo esto planteo que la cantidad de ejecuciones que se hacen son k + 2 (El caso inicial y el 0) es decir el Log2(n^2) es la cantidad de iteraciones para que i llegue a 0, faltando el caso inicial y final

Este algoritmo finalmente tiene:

$$2 + 4 * Log2(n^2) + 2 + 1 + 1 + 1 = 7 + 4Log2(n^2)$$

Tomando su complejidad en notación O():

$$O(Log2(n^2))$$

Al ejecutar "algoritmo(8)" este programa se encarga de tomar el valor de entrada (8) y a su cuadrado sumarle la variable J (Que empieza en 1 e incrementa de 1 en 1) e imprimir este resultado, posteriormente tomar la mitad del cuadrado del valor de entrada y sumarle J, luego la mitad de la mitad y así sucesivamente hasta llegar a $i \approx 0$.

Puntualmente en el caso de n=8 imprime lo siguiente:

Suma 65 (8^2+1)

Suma 34 (8^2/2+2)

Suma 19 (8^2/4+3)

Suma 12 (8^2/8+4)

Suma 9 (8^2/16+5)

Suma 8 (8^2/32+6)

Suma 8 (8^2/64+7)

2.

```
int algoritmo2(int n){
    int res = 1,i,j; //Esta linea cuenta por 3 ejecuciones (Declara 3 vars)
    for(i = 1; i<= 2 * n; i += 4){ //Esta linea se ejecuta 0.5n + 1
        for(j = 1; j * j <= n; j++){ //Esta linea se ejecuta 0.5n * (raiz(n)+1) = 0.5n+0.5n^3/2
            res +=2; //Se ejecuta 0.5n+0.5n^3/2-1
        }
    }
    return res; //Ejecuta 1 vez
}
```

La primera linea se ejecuta 3 veces, ya que crea 3 variables, la segunda al ser un ciclo que va desde 1 hasta 2n en incrementos de 4 se ejecuta 2n/4 veces y la condición donde falla +1 es decir 0.5n+1

La tercer linea es un ciclo que va desde 1 hasta j^2 incrementando j en 1, por lo tanto, el ciclo entrara siempre que j^2 sea menor que n, lo cual implica que j debe ser la raíz de n para que el ciclo deje de entrar, esto sumado al caso en el que falla (ultima comprobación) obtengo $\sqrt{n}+1$ multiplicado por la cantidad de veces que el ciclo anterior entra me queda

$$0.5n + 0.5n^{\frac{3}{2}}$$

La linea res += 2 se ejecuta la cantidad de veces que se entra al segundo ciclo, es decir el termino anterior -1, y pues la ultima linea se ejecuta 1 vez. Quedándome

$$3 + 1 - 1 + 0.5n + 0.5n + 0.5n^{\frac{3}{2}} + 0.5n + 0.5n^{\frac{3}{2}} = 3 + n + n^{\frac{3}{2}}$$

Ahora bien, tomando la derivada de los términos del polinomio verifico cual tiene un crecimiento mayor

$$\frac{d}{dn}(n) = 1$$

$$\frac{d}{dn}\left(n^{\frac{3}{2}}\right) = n^{\frac{1}{2}}$$

Por lo tanto, me queda

$$O(n^{\frac{3}{2}})$$

Al ejecutar algoritmo2(8) retorna 17 el proceso que sigue es que el primer ciclo se evalua 5 veces (incluyendo la vez que se evalua y no entra) y luego dentro del segundo ciclo se entra 4*raíz(4) es decir entra 8 veces y se prueba 9, con esas 8 entradas hace res+= 2 un total de 8 veces y con el valor inicial res = 1 da como resultado 17, por eso retorna 17

3.

```
void algoritmo3(int n){
    int i,j,k; //Esta linea se ejecuta 1 vez pero cuenta por 3
    for(i = n; i>1;i--) //Este ciclo entra n veces
        for(j = 1; j <= n; j++) //Este ciclo se analiza n+1 pero entra solamente n veces
            for(k = 1; k<= i;k++) // Este ciclo se analliza como una sumatoria
                printf("Vida cruel!!\n"); //Esta linea se ejecuta 1 vez menos que el ciclo ant
```

Este algoritmo tiene una complejidad $O(n^3)$ Ya que el primer ciclo se analiza n veces, el segundo ciclo solo depende de n y se analiza n+1 veces, pero solo entra n veces. Finalmente, el tercer ciclo depende del primero y del segundo, puntualmente depende del valor de i pero se ejecutara por cada valor de j, entonces se debe plantear como una sumatoria $\sum_{i=2}^{n}(i+1)$ ya que se debe multiplicar por la cantidad de veces que se entra al ciclo 2, dándonos como resultado un valor un $T(n)$ de grado 3. También la ultima linea se ejecuta 1 vez menos que el ciclo 3, es decir su resultado -1.

Analizando un poco los posibles casos

N	3	2	
i	3,2,1	2,1	Para cada repetición del ciclo 1 i toma:
j	1,2,3,4	1,2,3	Para cada I se ejecuta:
k	1,2,3,4	1,2,3	Para cada J se ejecuta eso (y depende de i)

$$3 + n + n(n+1) + n * \sum_{i=2}^{n}(i+1) + \sum_{i=2}^{n}(i+1) - 1$$

$$\sum_{i=2}^{n}(i+1) = ((n\text{^}(2))/(2)) + ((3*n)/(2)) - 2$$

Eso da como resultado:

$$O(n^3)$$

4.

```
int algoritmo4(int* valores, int n){
    int suma = 0, contador = 0; //2 ejecuciones
    int i, j, h, flag; //4 Ejecuciones
    for(i = 0;i<n;i++){ // n+1 Ejecuciones
        j = i + 1; //n ejecuciones
        flag = 0; //n ejecuciones
        while(j < n && flag == 0){ //En este ciclo hay que analizar con cuidado
            if(valores[i]<valores[j]){ //Este condicional tiene mejor y peor caso
                for(h = j; h < n; h++){ //Este ciclo hay que expresarlo como sumatoria
                    suma += valores[i]; //1 ejecucion menos que el for anterior
                }
            }
            else{ //Mejor y peor caso
                contador++; //Depende del caso
                flag = 1; //Depende del caso
            }
            ++j; // Una vez menos que el ciclo while
        }
    }
    return contador; //1 Ejecucion
}
```

Este algoritmo requiere un análisis a detalle, en la primera linea se declaran dos variables, cuenta como 2 ejecuciones, luego en la segunda linea se crean 4 variables, cuenta como 4 ejecuciones, el ciclo for va desde 0 hasta n en incrementos de 1, por lo tanto, entrara al ciclo n veces pero se comprobara n+1 veces.

En caso de entrar al ciclo la linea j= i+1 y flag = 0 se ejecutan n veces.

En el ciclo while va desde j = i +1 hasta n, pero i incrementa con el ciclo for anterior, además de eso está la condición de la bandera que varia dependiendo del caso, entonces.

En el peor caso de que siempre se cumpla el if(valores[i]<valores[j]) el siguiente ciclo for(h=j;h<n;h++) se expresa con la siguiente sumatoria:

$$\sum_{j=i+1}^{n} (j+1)$$

Además a esto hay que multiplicarlo por la cantidad de veces que se ingresa al ciclo while, el cual incrementa de 1 en 1 pero su valor de inicio es i+1 por lo tanto para cada i, j recorrerá hasta n (en el caso de que la flag siempre sea 0) esto se puede expresar como la siguiente sumatoria:

$$\sum_{i=0}^{n} i$$

Como la solución de la sumatoria da por resultado un polinomio de grado 2 y se multiplica, obtengo hasta ese punto un polinomio $T(n)$ de grado 3.

Finalmente hay que tener en cuenta que el tercer ciclo depende de j, quien depende de i, por lo que en realidad se debe expresar como una doble sumatoria (Si no me equivoco es la siguiente)

$$\sum_{i=0}^{n}\left(\sum_{j=i+1}^{n}(j+1)\right)$$

Cuyo resultado es un polinomio de grado 3.

Por ese motivo es que la complejidad total del algoritmo en el peor de los casos incrementaría en orden de $O(n^3)$ sin embargo en un análisis del mejor caso el tercer ciclo nunca se ingresa pues la condición siempre seria falsa, por lo tanto, la complejidad en ese caso sería diferente, ya que solo se ejecutaría una vez el ciclo while, luego la flag cambia y pasa directamente a la siguiente iteración del ciclo for más externo, en caso tal la complejidad del ciclo pasaría a ser $O(n^2)$.

Nota: En este caso aunque se solucionar la sumatoria doble como dijeron que lo dejáramos hasta ahí, lo dejo, sin embargo podrían considerar hacer uso de algun sistema de algebra computarizada para poder pedir el polinomio T(n).

5.

```
void algoritmo5(int n){
    int i = 0; //1 Ejecucion
    while(i<=n){ // Este ciclo se ejecuta siempre 7 veces
        printf("%d\n", i); //una ejecucion menos que el ciclo 6
        i += n/5; //una ejecucion menos que el ciclo 6
    }
}
```

Este algoritmo tiene complejidad O(1) ya que siempre que se ejecute con un n cualquiera el numero de veces que se ejecuta el ciclo es siempre la misma, esto se puede demostrar de la siguiente manera:

$$k = numero\ de\ repeticiones$$

$$i = k * \left(\frac{n}{5}\right)$$

$$i \leq n$$

$$k * \left(\frac{n}{5}\right) \leq n$$

$$k \geq 5$$

Por ende, el ciclo siempre será constante para cualquier k mayor que 5 y esto sumando el caso 0 y el caso de verificación, obtenemos un total de 7, por lo que podemos decir que la complejidad es O(1).

6.

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0.111 seg	35	2.069 seg
10	0.111 seg	40	21.994 seg
15	0.095 seg	45	3 min 55.767 seg
20	0.111 seg	50	45 min 8.890 seg
25	0.127 seg	60	Seguramente más de 3 segundos
30	0.285 seg	100	Seguramente más de 3 segundos

El valor mas alto que mi calculadora con un I7 9750H y una RTX 2070, 16 de Ram a 2666mhz fue hasta 50, ya que mi paciencia no dio para más.

Analizando los datos y viendo el comportamiento ese crecimiento con variaciones tan pequeñas de datos de entrada me hacen pensar dos posibilidades, o una complejidad exponencial o incluso factorial.

7.

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0.127 seg	45	0.111 seg
10	0.109 seg	50	0.128 seg
15	0.095 seg	100	0.095 seg
20	0.109 seg	200	0.095 seg
25	0.095 seg	500	0.110 seg
30	0.109 seg	1000	0.110 seg
35	0.127 seg	5000	0.111 seg
40	0.110 seg	10000	0.110 seg

8.

Tamaño Entrada	Tiempo Sol Propia	Tiempo Sol Profesores
100	0.126 seg	0.109 seg
1000	0.095 seg	0.112 seg
5000	0.173 seg	0.095 seg
10000	0.379 seg	0.111 seg
50000	5.947 seg	0.190 seg
100000	22.246 seg	0.331 seg
200000	1 min 22.307 seg	0.726 seg

Aunque su solución sea más rápida la mía les enseña a tener paciencia.

A. Los tiempos de ejecución se van alejando a medida que el tamaño de entrada crece, esto probablemente se deba a la implantación en cada caso, en mi solución utilizo métodos cuestionables para probar la suma de dígitos mientras que en su solución utiliza una operación aritmética que considero es mas eficiente. Sin embargo, el mío le dará al usuario una valiosa cualidad en su vida la cual es la paciencia.

B. En mi solución la complejidad del bloque de código que define si un numero es primo o no, es de O(n^2) mientras que en el caso de ustedes es de O(n), ahí radica la diferencia el mío presenta un crecimiento cuadrático y el de ustedes lineal. Admito que ganaron con su código, pero yo gane al enseñar paciencia.