Tarea 2

Estructura de datos

Oscar Vargas Pabón

Punto 1:

```
void algoritmo1(int n){
   int i, j = 1;
   for(i = n * n; i > 0; i = i / 2){
       int suma = i + j;
       printf("Suma %d\n", suma);
       ++j;
   }
}
```

Análisis de complejidad

```
   int i, j = 1;
```

Se dice que ejecuta dos veces, pues son dos variables las iniciadas. Nos es irrelevante para la complejidad, pues, sin importar el valor de n, esta línea es ejecutada solo las dos veces.

```
for(i = n * n; i > 0; i = i / 2){
```

$i = n^2, \frac{n^2}{2}, \frac{n^2}{2^2}, \ldots, \frac{n^2}{2^r}, \frac{n^2}{2^{r+1}}$ donde $\frac{n^2}{2^r} = 1$, por lo que $\frac{n^2}{2^{r+1}}$ Incumple la condición de i > 0

Tenemos entonces que

$$\frac{n^2}{2^r} = 1$$

$$n^2 = 2^r$$

$$\log_2 n^2 = \log_2 2^r$$

$$2 * \log_2 n = r$$

Pero aquí aún falta algo:

$$i = \frac{n^2}{2^0}, \frac{n^2}{2^1}, \frac{n^2}{2^2}, \dots, \frac{n^2}{2^r}, \frac{n^2}{2^{r+1}}$$

Nótese como los valores de i pueden ser escritos como una secuencia de n^2 dividido una potencia de 2 que va desde 0 hasta r+1, por lo que i toma r + 2 valores. Esto se ve en:

$$2 * \log_2 n + 2 = r + 2$$

```
int suma = i + j;
printf("Suma %d\n", suma);
++j;
```

Estas tres líneas se ejecutan tantas veces como se ingresa al ciclo, por lo que las tres se ejecutan la misma cantidad de veces. Con la línea del for, se analizó que i toma r+2 valores. También vimos que el valor de i $\frac{n^2}{2^{r+1}}$ incumple la condición del for, por lo que es el único valor de i con el que no se entra al ciclo. Es por esto que estas tres líneas solo se ejecutan r+1 veces. Esto se representa con lo que sigue:

$$2 * \log_2 n + 1 = r + 1$$

Finalmente, tenemos la suma de todas las ejecuciones de todas las líneas:

$$T(n) = 3 * (2 * \log_2 n + 1) + (2 * \log_2 n + 2) + 2$$

$$T(n) = 8\log_2 n + 7$$

$$T(n) = O(\log_2 n)$$

Se concluye que la complejidad del algoritmo1 es de tipo logarítmica.

Al insertar algoritmo1(8) obtenemos:

Suma 65

Suma 34

Suma 19

Suma 12

Suma 9

Suma 8

Suma 8

Esto porque i toma valores 64, 32, 16, 8, 4, 2, 1, 0. j toma valores 1, 2, 3, 4, 5, 6, 7, 8. La siguiente tabla muestra los distintos valores con los que se entra al ciclo y como afecta en la variable suma

(que es la que se imprime). El valor menor o igual a 0 de i y el valor 8 de j no se consideran en la tabla porque no se ingresa al ciclo con estos.

Valores de i	Valores de j	Valores de suma
64	1	65
32	2	34
16	3	19
8	4	12
4	5	9
2	6	8
1	7	8

Punto 2:

```
int algoritmo2(int n){
   int res = 1, i, j;

   for(i = 1; i <= 2 * n; i += 4)
       for(j = 1; j * j <= n; j++)
           res += 2;

   return res;
}
```

Análisis de complejidad:

```
int res = 1, i, j;
```

Esta línea se dice ejecuta 3 veces, pues son 3 las variables inicializadas. Igual no afecta mucho a la complejidad del algoritmo, pues se ejecuta las mismas 3 veces sin importar de n (constante).

```
for(i = 1; i <= 2 * n; i += 4)
```

i inicia en 1, y luego en cada iteración aumenta en 4. La formula $4f + 1$ da el valor de i para una iteración f cualquiera (f se cuenta desde 0). Tenemos que buscar entonces el menor r posible tal que

$$4r + 1 > 2 * n$$

$$4r > 2 * n - 1$$

$$r > \frac{n}{2} - \frac{1}{4}$$

Como r y n son enteros, tenemos que n solo tiene dos opciones: ser par (divisible entre 2) o no

n%2 == 0	**n%2 == 1**
Como n es par, se puede representar con un k entero de la manera: $$2k = n$$ Por lo que $\frac{n}{2} = k$ La desigualdad nos queda entonces $$r > k - \frac{1}{4}$$ $$r > (k-1) + \frac{3}{4}$$ No resulta muy difícil ver que el menor numero entero que puede tomar r, cumpliendo con la desigualdad, es el mismo k. No olvidemos que 4f + 1 funciona contando f desde 0 (pues la primera vez que se ejecuta, i comienza con su valor inicial: 1). Por todo esto se concluye por este lado que con un n par, esta línea se ejecuta $\frac{n}{2} + 1$veces	Como n es impar, se puede representar con un h entero de la manera: $$2k + 1 = n$$ Por lo que $\frac{n}{2} = k + \frac{1}{2}$ La desigualdad nos queda entonces $$r > (k + \frac{1}{2}) - \frac{1}{4}$$ $$r > k + \frac{1}{4}$$ Se concluye entonces que el menor número entero que puede tomar r, cumpliendo con la desigualdad, es k+1 $$2(k+1) = 2k + 2 = (2k+1) + 1 = n + 1$$ $$2(k+1) = n + 1$$ $$k + 1 = \frac{n+1}{2}$$ No olvidemos que 4f + 1 funciona contando f desde 0 (pues la primera vez que se ejecuta, i comienza con su valor inicial: 1). Se concluye por este lado que con un n impar, esta línea se ejecuta $\frac{n+1}{2}$ +1 veces.

Ante esto se puede concluir que esta línea maneja un mejor y peor caso, pues con un r impar tenemos que las ejecuciones de esta línea $\frac{r+1}{2}$ + 1 es igual a las ejecuciones con n+1 (que seria par).

```
for(j = 1; j * j <= n; j++)
```

i toma valores desde 1, hasta un r entero tal que

$$r^2 > n$$

$$r > \sqrt{n}$$

Hay un numero entero k, tal que $k^2 = n$	**No se cumple lo de la izquierda**
En este caso, j iría hasta k+1, lo que es en ultimas $\sqrt{n} + 1$	En este caso, $\sqrt{n}$ es un decimal que cumple $f < \sqrt{n} < r$ para un f y r enteros. f es la función floor de $\sqrt{n}$, mientras que r seria la función ceil. j iría entonces hasta r

En el caso en el que $\sqrt{n}$ no sea entero (la derecha), r se puede definir como floor($\sqrt{n}$) + 1. De manera curiosa, cuando $\sqrt{n}$ si es entero, también puede definirse como floor($\sqrt{n}$) + 1 porque floor de un entero es el mismo entero.

J va desde 1 hasta floor($\sqrt{n}$) + 1, j toma floor($\sqrt{n}$) + 1 valores.

Esta línea se ejecuta entonces (floor($\sqrt{n}$) + 1) multiplicado por las veces que se entra al ciclo for anterior ($\frac{n}{2}$ o $\frac{n+1}{2}$)

```
res += 2;
```

Como esta línea se ejecuta cuantas veces se entra en el anterior for, tenemos floor($\sqrt{n}$) multiplicado por las veces que se ingresa al primer for ($\frac{n}{2}$ o $\frac{n+1}{2}$).

```
return res;
```

Se ejecuta una única vez, independiente de n.

Suma de las ejecuciones de todas las líneas:

n par (mejor caso)	**n impar (peor caso)**
$T(n) = 3 + \frac{n}{2} + 1 + \frac{n}{2} * \left(floor\left(\sqrt{n}\right) + 1\right) + \frac{n}{2} * floor(\sqrt{n})$ $T(n) = n * floor\left(\sqrt{n}\right) + n + 4$ $T(n) = O(n * floor\left(\sqrt{n}\right))$	$T(n) = 3 + \frac{n+1}{2} + 1 + \frac{n+1}{2} * \left(floor\left(\sqrt{n}\right) + 1\right) + \frac{n+1}{2} * floor(\sqrt{n})$ $T(n) = (n+1) * floor\left(\sqrt{n}\right) + n + 5$ $T(n) = n * floor\left(\sqrt{n}\right) + floor\left(\sqrt{n}\right) + n + 5$ $T(n) = O(n * floor\left(\sqrt{n}\right))$

La complejidad del algoritmo, tanto en su mejor como en su peor caso, es de

$$T(n) = O\left(n * floor\left(\sqrt{n}\right)\right)$$

Se podría dejar como $T(n) = O\left(n * \sqrt{n}\right) = O(n^{\frac{3}{2}})$ Por simplicidad.

Al ejecutar algoritmo2(8) obtenemos: 17

Esto porque res obtiene el valor de 1 + (veces que se repite res += 2) * 2

$$1 + 2 * (\frac{n}{2} * floor\left(\sqrt{n}\right))$$

$$1 + 2 * (\frac{8}{2} * floor\left(\sqrt{8}\right))$$

$$1 + 2 * (4 * 2)$$

$$1 + 2 * (8)$$

1 + 16

17

Hay también una tabla que muestra los valores que toman las variables en algoritmo2(8). Los espacios en blanco representan que, con el valor de la otra variable no entra al ciclo. Ejemplo: con j = 3 tenemos que j*j = 9, por lo que 9 <= n que es igual a 9 <= 8 es falso, por lo que la variable res queda en blanco.

i	j	Res (inicia en 1)
1	1	3
	2	5
	3	
5	1	7
	2	9
	3	
9	1	11
	2	13
	3	
13	1	15
	2	17
	3	
17		

Punto 3:

```
void algoritmo3(int n){
   int i, j, k;
   for(i = n; i > 1; i--)
       for(j = 1; j <= n; j++)
           for(k = 1; k <= i; k++)
              printf("Vida cruel!!\n");
}
```

Análisis de complejidad:

```
int i, j, k;
```

Esta línea se dice ejecuta 3 veces, pues son 3 las variables inicializadas. Igual no afecta mucho a la complejidad del algoritmo, pues se ejecuta las mismas 3 veces sin importar de n (constante).

```
for(i = n; i > 1; i--)
```

I toma los valores $i = n, n-1, \dots, 2, 1$.

Esta línea se ejecuta n veces.

```
for(j = 1; j <= n; j++)
```

J toma los valores $1, 2, \dots, n, n+1$

Esta línea se ejecuta n+1 veces por las veces que se entra al anterior for (pues este es anidado)

$$(n-1) * (n+1)$$

$$n^2 - 1$$

Esta línea se ejecuta entonces $n^2 - 1$ veces.

```
for(k = 1; k <= i; k++)
```

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

Cuando i = 2, entonces k toma valores 1,2,3 (3 veces). Cuando i = 5, k toma valores 1,2,3,4,6 (6 veces). Por esto se hace el i+1

$$n * \sum_{i=2}^{n} i + 1$$

$$n * \sum_{i=2}^{n} i + n * \sum_{i=2}^{n} 1$$

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

$$n * \frac{n(n+1)}{2} - n + n^2 - n$$

$$n * \frac{n^2 + n}{2} + n^2 - 2n$$

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

Se ejecuta $\frac{n^3}{2} + \frac{3}{2}n^2 - 2n$ veces. Aquí ya se han considerado los anteriores ciclos.

```
printf("Vida cruel!!\n");
```

$$\sum_{i=2}^{n} n * i$$

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

$$n * \frac{n^2 + n}{2} - n$$

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

Esta línea se ejecuta entonces $\frac{n^3}{2} + \frac{1}{2}n^2 - n$ veces

Suma líneas ejecutadas:

$$T(n) = 3 + n + n^2 - 1 + \frac{n^3}{2} + \frac{3}{2}n^2 - 2n + \frac{n^3}{2} + \frac{1}{2}n^2 - n$$

$$T(n) = n^3 + 3n^2 - 2n + 2$$

$$T(n) = O(n^3)$$

El algoritmo 3 tiene entonces complejidad $O(n^3)$.

Punto 4:

```
int algoritmo4(int* valores, int n){
   int suma = 0, contador = 0;
   int i, j, h, flag;

   for(i = 0; i < n; i++){
     j = i + 1;
     flag = 0;
     while(j < n && flag == 0){
        if(valores[i] < valores[j]){
           for(h = j; h < n; h++){
             suma += valores[i];
           }
        }
        else{
           contador++;
           flag = 1;
        }
        ++j;
     }
   }
   return contador;
}
```

En este caso, tomare un camino ligeramente distinto.

Mejor caso:

En el mejor caso, el condicional dentro del while nunca se cumple.

```
if(valores[i] < valores[j])
```

Esto permite no entrar nunca al ciclo for que está dentro, además de que el else vuelve 1 la bandera 'flag', lo que hace que el ciclo while acabe de manera prematura.

En este caso contaríamos 2 ejecuciones para la primera línea y 4 ejecuciones para la segunda

```
int suma = 0, contador = 0;
int i, j, h, flag;
```

La línea del for se ejecuta n veces, mientras que el 'j=i+1' y 'flag=0' se ejecuta n-1 veces (no se entra a este for con i =n).

```
   for(i = 0; i < n; i++){
     j = i + 1;
     flag = 0;
```

Las siguientes líneas dentro del while se ejecutarían 1 vez cada que se entra al while(1 * (n-1)):

```
if(valores[i] < valores[j]){
```

```
else{
    contador++;
    flag = 1;
}
++j;
```

El while en cuestión se ejecutaría dos veces (la vez que entra y la vez que para), por lo que se ejecutaría 2(n-1) veces.

Las demás líneas del while (las que están dentro del if) nunca se ejecutarían.

```
    return contador;
```

Por último, el return se ejecuta 1 vez

Tenemos entonces:

$$T(n) = 6 + n + n - 1 + n - 1 + 2(n-1) + n - 1 + n - 1 + n - 1 + n - 1 + n - 1 + 1$$

$$T(n) = 10n - 2$$

$$T(n) = O(n)$$

En el mejor caso, el algoritmo4 es de orden lineal.

Peor caso:

En el peor caso, el condicional siempre se cumple. Por este motivo podemos omitir las líneas del else, pues nunca se llegan a ejecutar.

```
else{
    contador++;
    flag = 1;
}
```

Las siguientes líneas se ejecutan igual cantidad de veces en el peor como en el mejor caso.

```
    int suma = 0, contador = 0;
    int i, j, h, flag;

    for(i = 0; i < n; i++){
      j = i + 1;
      flag = 0;

    return contador;
```

Las 2 líneas que comienzan por 'int' y el return se ejecutan en total 7 veces (porque se inician varias variables).

La línea del for se ejecuta n+1 veces, mientras que las otras dos que dependen de las veces que se entra a este for: n veces. Tenemos por ahora:

$3n + 8$

```
while(j < n && flag == 0){
```

Como la variable 'flag' solo cambia su valor en el else y, por este ser el peor caso, el else nunca se ejecuta, podemos omitir la condición de 'flag == 0'. Quedamos con la condición de j < n. j se suma de 1 en 1 como se ve en la línea '++j;' que esta al final del while.

Con i = 0 tenemos $j = 1, 2, .., n, n + 1$. Con i = 2 tenemos $j = 3, 4, .., n, n + 1$. Con i = (n-3) $i = n - 2, n - 1, n, n + 1$. Con i = (n – 1) ya no entra al ciclo, pues j = i +1 y i+1 = (n-1)+1 = n que incumple n < n. Esto indica que para i = (n-1), n solo se ejecuta 1 vez.

Tenemos entonces que esta línea se ejecuta:

$$\sum_{i=0}^{n-2}(n+1) - i$$

$$\sum_{i=0}^{n-2}(n+1) \; - \sum_{i=0}^{n-2} i$$

$$\sum_{i=0}^{n-2} n + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-2} i$$

$$n * \sum_{i=0}^{n-2} 1 + (n-1) - \frac{(n-2) * (n-1)}{2}$$

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

$$n(n-1) + n - 1 - \frac{n^2}{2} + \frac{3}{2}n - 1$$

$$n^2 - n - \frac{n^2}{2} + \frac{5}{2}n - 2$$

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

No olvidemos los dos valores de i con los que esta line se ejecuta solo 1 vez, por lo que nos queda.

$$\frac{1}{2}n^2 + \frac{3}{2}n$$

Esta línea se ejecuta entonces $\frac{1}{2}n^2 + \frac{3}{2}n$ en el peor caso.

```
if(valores[i] < valores[j]){
++j;
```

Las dos líneas de arriba se ejecutan la misma cantidad de veces, por lo que serán analizadas a la par. Se ejecutan cada vez que se entra al while, el cual entra en todos los valores de j menos n+1.

$$\sum_{i=0}^{n-2}(n) - i$$

$$n * \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-2} i$$

$$n * (n-1) - \frac{n^2 - 3n + 2}{2}$$

$$\frac{1}{2}n^2 + \frac{1}{2}n - 1$$

Ambas líneas se ejecutan cada una un total de $\frac{1}{2}n^2 + \frac{1}{2}n - 1$ veces

```
for(h = j; h < n; h++){
```

Cuando j= 1 $h = 1,2, \dots, n, n+1$. Cuando j= 3 $h = 3,4, \dots, n, n+1$. Se ejecuta la misma cantidad de veces que el while, pero este esta anidado en el while.

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

$$\sum_{i=o}^{n-2} n^2 + n - 2in - i + i^2$$

$$\sum_{i=1}^{n-1} n^2 + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-2} 2in - \sum_{i=1}^{n-2} i + \sum_{i=1}^{n-2} i^2$$

$$(n^3 - n^2) + (n^2 - n) \; - \; 2n \sum_{i=1}^{n-2} i - \frac{(n-2)(n-1)}{2} \; + (\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6})$$

$$\frac{7}{3}n^3 - n \; - \; 2n\frac{n^2 - 3n + 2}{2} - \frac{n^2 - 3n + 2}{2} + \frac{n^2}{2} + \frac{n}{6}$$

$$\frac{7}{3}n^3 - n \; - \; n^3 + 3n^2 + 2n + -\frac{n^2}{2} + \frac{3n}{2} + 1 \; + \frac{n^2}{2} + \frac{n}{6}$$

$$\frac{4}{3}n^3 + 3n^2 + \frac{16n}{6} + 1$$

Esta línea se ejecuta $\frac{4}{3}n^3 + 3n^2 + \frac{16n}{6} + 1$ veces.

```
suma += valores[i];
```

Esta línea se ejecuta tantas veces como se entra en el anterior for (el que ya esta anidado 2 veces). Entra con todos los valores de h excepto h +´1. Por esto:

$$\sum_{i=o}^{n-2}(n-i)*(n-i)$$

$$\sum_{i=o}^{n-2} n^2 - 2ni + i^2$$

$$\sum_{i=o}^{n-2} n^2 - \sum_{i=o}^{n-2} 2ni + \sum_{i=o}^{n-2} i^2$$

$$\sum_{i=1}^{n-1} n^2 - 2n\sum_{i=1}^{n-2} i + \sum_{i=1}^{n-2} i^2$$

$$(n^3 - n^2) - 2n\frac{n^2-3n+2}{2} + (\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6})$$

$$\frac{4}{3}n^3 - \frac{1}{2}n^2 - n^3 + 3n^2 + 1 + \frac{1}{6}n$$

$$\frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n + 1$$

Esta linea se ejecuta $\frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n + 1$ veces.

Suma líneas ejecutadas peor caso:

$$T(n) = 3n + 8 + \frac{1}{2}n^2 + \frac{3}{2}n + \frac{1}{2}n^2 + \frac{1}{2}n - 1 + \frac{4}{3}n^3 + 3n^2 + \frac{16n}{6} + 1 + \frac{1}{3}n^3 + \frac{5}{2}n^2 + \frac{1}{6}n + 1 + \frac{1}{2}n^2 + \frac{1}{2}n - 1$$

$$T(n) = \frac{5}{3}n^3 + 8n^2 + \frac{50}{6}n + 8$$

$$T(n) = O(n^3)$$

Tenemos que, en el peor caso, el algoritmo4 funciona en tiempo cuadrático.

Mejor caso	Peor caso
La condición dentro del while nunca se cumple, por lo que se ejecuta el else que le cambia el valor a la variable 'flag' que hace que el while termine de manera prematura.	La condición dentro del while siempre se cumple, por lo que el while se rige por el valor de j y ejecuta un for anidado.
$T(n) = O(n)$	$T\ (n) = O(n^3)$

Punto 5:

```
void algoritmo5(int n){
  int i = 0;
  while(i <= n){
    printf("%d\n", i);
    i += n / 5;
  }
}
```

Análisis complejidad:

```
int i = 0;
```

Se ejecuta 1 vez independientemente de n.

```
while(i <= n){
  i += n / 5;
```

Si / es división entera, entonces tenemos con n= 0,1,2,3,4 que los valores por los que se aumenta 0/5 =1/5 = 2/5 = 3/5 = 4/5 = 0, por lo que con un n con alguno de esos valores obtendríamos un ciclo infinito.

Si / es división con decimal, esto solo ocurriría con n = 0.

No sé cómo se representa en notación o grande un ciclo infinito. Asumo que sería algo como $O(\infty)$.

En los demás casos, i toma los valores respectivos de $i = 0, \frac{1}{5}n, \frac{2}{5}n, \frac{3}{5}n, \frac{4}{5}n, n, \frac{6}{5}n$ son 7 valores, aunque con $\frac{6}{5}n$ no entra al while.

las líneas del printf y i += se ejecutarían un total de 6 veces independientemente de n.

```
printf("%d\n", i);
i += n / 5;
```

El while se ejecutaría 7 veces independientemente de n.

```
while(i <= n){
```

$$T(n) = 1 + 7 + 6 + 6$$

$$T(n) = 20$$

$$T(n) = O(1)$$

Se concluye que el algoritmo5, a excepción de los ciclos infinitos a los que puede entrar, funciona en tiempo lineal.

Punto 6:

Tamaño Entrada	Tiempo en segundos	Tamaño Entrada	Tiempo en segundos
5	0.060	35	3.592
10	0.065	40	38.878
15	0.063	45	7 minutos 14.952
20	0.065	50	
25	0.091	60	
30	0.380	100	

Implementación en Python:

```python
def fiboRecur(n):
    if n==0:
        ans = 0
    elif n==1:
        ans = 1
    else:
        ans = fiboRecur(n-1) + fiboRecur(n-2)
    return ans

fiboRecur(40)
```

Los espacios en blanco no los pienso probar.

¿Cuál es el valor más alto para el cuál pudo obtener su tiempo de ejecución? 7 minutos 14.952

¿Qué puede decir de los tiempos obtenidos? Al comienzo se mantienen más o menos constante, pero después de la entrada con valor 25 comienza a escalar mucho.

¿Cuál cree que es la complejidad del algoritmo? Creo que es complejidad $O(2^n)$, pues por cada invocación de fiboRecur() se hacen dos invocaciones recursiva, que a su vez hacen otras 2 (y así sucesivamente).

Punto 7:

Tamaño Entrada	Tiempo	Tamaño Entrada	Tiempo
5	0.061	45	0.063
10	0.061	50	0.061
15	0.060	100	0.064

20	0.062	200	0.061
25	0.065	500	0.062
30	0.062	1000	0.062
35	0.063	5000	0.061
40	0.061	10000	0.060

Implementación en Python:

```python
def iterFib(n):
    if n == 0:
        act = 0
    else:
        ant2 = 0
        ant1 = 1
        for i in range(n):
            act = ant1 + ant2
            ant1 = ant2
            ant2 = act
    return act

iterFib(10000)
```

Análisis complejidad:

```python
    if n == 0:
        act = 0
    else:
```

El if esta para manejar el único caso de n = 0. Con n = 0, el if y lo que está dentro del if se ejecuta 1 vez. El resto de las veces hay 2 ejecuciones en este fragmento (el if y el else).

```python
        ant2 = 0
        ant1 = 1
```

Se ejecuta cada una de estas dos líneas 1 vez en el caso de n ¡= 0

```python
        for i in range(n):
```

Se ejecuta n+1 veces

```python
            act = ant1 + ant2
            ant1 = ant2
            ant2 = act
```

Cada una de las 3 líneas se ejecuta n veces

```python
    return act
```

Se ejecuta una vez independiente de n.

$$T(n)5 + n + 1 + n + n + n$$

$$T(n) = 4n + 6$$

$$T(n) = O(n)$$

Como en el caso de T(0), queda nomas el término que no depende de n, entonces el caso donde n = 0 no afecta realmente la complejidad lineal (no hay mejor-peor caso con n=0).

La implementación con ciclos de la secuencia de Fibonacci es de tipo lineal.

Punto 8:

Tamaño Entrada	Tiempo Solución Propia	Tiempo Solución Profesores
100	0.118	0.119
1000	0.123	0.118
5000	0.170	0.151
10000	0.255	0.142
50000	2	0.244
100000	7.102	0.469
200000	25.516	0.957

(a)¿qué tan diferentes son los tiempos de ejecución y a qué cree que se deba dicha diferencia?

Una de las falencias de mi código es la operación que uso para sumar los dígitos, pues los convierte y desconvierte a string, a diferencia del de los profesores que usa el módulo y la división entera únicamente. Otra falencia es que busco en todos los primos anteriores sin tomar en cuenta que una vez el cuadrado de alguno de ellos sobrepasa el número que busco, ese numero ya no va a tener divisores. De resto no se.

(b) ¿cuál es la complejidad de la operación o bloque de código en el que se determina si un número es primo en cada una de las soluciones?

Yo	Profesores

`primalidad = True`
`j = 0`
`while primalidad and j < len(primos):`
`  if i%primos[j] == 0:`
`  primalidad = False`
`  else:`
`  j += 1`
	`def esPrimo(n):`
	`  if n < 2: ans = False`
	`  else:`
	`  i, ans = 2, True`
	`  while i * i <= n and ans:`
	`  if n % i == 0: ans = False`
	`  i += 1`
	`  return ans`

En el mejor caso (i es divisible entre 2), el while se ejecuta dos veces, el if 1 vez y el 'primalidad = False' también una vez. En el peor caso (i es primo), entonces el while se ejecuta len(primos)+1 veces, el if, else y el 'j += 1' otras len(primos) veces cada uno. Funciona en tiempo $len(primos)$lineal en len(primos). No se como se podría poner en términos de n.	El if, else y 'i, ans = 2, True' funcionan en tiempo constante. En el peor caso (es primo), el while funciona $\sqrt{n} + 1$ veces, mientras que solo ingresa al ciclo $\sqrt{n}$ veces. En el mejor caso (divisible entre 2), el while se ejecuta 2 veces, el if y el 'i += 1' una vez y el 'ans = False' también una vez. Funciona en tiempo $O(\sqrt{n})$

FIN