

Ana Gabriela Argüello Cedeño

Punto 1.

```
void algoritmo1(int n){
    int i, j = 1; 2
    for(i = n * n; i > 0; i = i / 2){  $\text{piso}[\log_2 n^2 + 2]$ 
        int suma = i + j;  $\text{piso}[\log_2 n^2 + 1]$ 
        printf("Suma %d\n", suma);  $\text{piso}[\log_2 n^2 + 1]$ 
        ++j;  $\text{piso}[\log_2 n^2 + 1]$ 
    }
}
```

Al ejecutar el algoritmo con $n = 8$ el resultado es:

| |
|---------|
| Suma 65 |
| Suma 34 |
| Suma 19 |
| Suma 12 |
| Suma 9 |
| Suma 8 |
| Suma 8 |

Esto ya que los valores actuales de lo que es i y j se suman y producen el resultado dado en el print. Para más claridad se muestra los valores de i y j y su sumatoria:

| i | j | Suma |
|-----|-----|-------|
| 64 | 1 | 65 |
| 32 | 2 | 34 |
| 16 | 3 | 19 |
| 8 | 4 | 12 |
| 4 | 5 | 9 |
| 2 | 6 | 8 |
| 1 | 7 | 8 |
| 0 | 8 | ----- |

La última fila no se imprime y no se suma puesto que no se cumple la condición de que i sea mayor a 0 (el resultado verdadero es 0.5 pero en C se manejan enteros entonces queda el 0).

Este algoritmo se ejecuta $(1 + (\log_2 n^2 + 2) + (\log_2 n^2 + 1) + (\log_2 n^2 + 1) + (\log_2 n^2 + 1))$, $4 \log_2 n^2 + 6$, por tal motivo la complejidad del algoritmo es $O(\log_2 n^2)$

|

Punto 2

```

int algoritmo2(int n){
    int res = 1, i, j; 3
    for(i = 1; i <= 2 * n; i += 4) piso  $\left\lfloor \frac{(2 * n) + 6}{4} \right\rfloor$ 
        for(j = 1; j * j <= n; j++) (techo( $\sqrt{n+1}$ )) * (piso  $\left\lfloor \frac{(2 * n) + 6}{4} \right\rfloor - 1$ )
            res += 2; (techo( $\sqrt{n+1}$ ) - 1) * (piso  $\left\lfloor \frac{(2 * n) + 6}{4} \right\rfloor - 1$ )
    return res; 1
}

```

El resultado que retorna al remplazar n por en numero 8 es: 17

Este algoritmo se ejecuta $(3 + \left(\frac{(2*n)+6}{4}\right) + ((\sqrt{n+1})) * (\left\lfloor \frac{(2*n)+6}{4} \right\rfloor - 1) + (\sqrt{n+1}) - 1) * (\left\lfloor \frac{(2*n)+6}{4} \right\rfloor - 1) + 1) n\sqrt{n+1} + \sqrt{n+1} + 5$, por tal motivo la complejidad del algoritmo es $O(\sqrt{n+1})$

El resultado anterior se da por lo siguiente:

| i | j | res |
|----|-------|-------|
| 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 | ----- | ----- |

En la ultima fila no se ejecuta el j ni el res por el motivo de que el numero que entro a i, no cumple la condición de que $i \leq 2*n$

Punto 3

```
void algoritmo3(int n){  
    int i, j, k; 3  
    for(i = n; i > 1; i--) n+1  
        for(j = 1; j <= n; j++) n  
            for(k = 1; k <= i; k++)  $\left(\frac{n^2 + 3n + 2}{2}\right) - 1$   
                printf("Vida cruel!!\n");  $\frac{n(n+1)}{2}$   
}
```

Este algoritmo se ejecuta $(3 + n + 1 + n + \left(\frac{n^2 + 3n + 2}{2}\right) - 1) + \frac{n(n+1)}{2}$, $n^2 + 4n + 4n$ veces, por tal motivo su complejidad es de $O(n^2)$

Punto 4

Esta operación lo que calcula es si una lista esta en orden ascendente o descendente y retorna los valores del contador que serían los números que no se encuentran en el orden de menor a mayor.

```
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;
}
```

Para este algoritmo nos encontramos con el mejor y peor caso:

1. En el mejor caso, los valores están en orden descendente, lo que significa que la línea del for de h, y la línea de lo suma, no se ejecutaran nunca
2. En el peor caso, los valores estarán en orden ascendente lo que ocasionará que la línea del for en h y la de la suma se ejecute,

causando que el algoritmo se ejecute una gran cantidad de veces

Para el mejor caso:

```

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

    for(i = 0; i < n; i++){  $n + 1$ 
        j = i + 1;  $n$ 
        flag = 0;  $n$ 
        while(j < n && flag == 0)  $(2n) - 1$ 
            if(valores[i] < valores[j])  $n - 1$ 
                for(h = j; h < n; h++){  $0$ 
                    suma += valores[i];  $0$ 
                }
            }
        else  $n - 1$ 
            contador++;  $n - 1$ 
            flag = 1;  $n - 1$ 
        }
        ++j;
    }
    return contador;  $1$ 
}

```

Este algoritmo ejecutara $(n+1+n+n+(2n)-1+n-1+n-1+n-1+n-1)$, $9n - 3$ veces. La complejidad de este algoritmo es $O(n)$

Para el peor caso:

```

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

    for(i = 0; i < n; i++){  $n + 1$ 
        j = i + 1;  $n$ 
        flag = 0;  $n$ 
        while(j < n && flag == 0){  $\sum_{j=i+1}^n n-j$ 
            if(valores[i] < valores[j]){  $\sum_{j=i+1}^n (n-j) - 1$ 
                for(h = j; h < n; h++){  $\left(\sum_{h=j}^n h\right) \times \left(\sum_{j=i+1}^n (n-j) - 1\right)$ 
                    suma += valores[i];
                }
            }
            else{  $0$ 
                contador++;  $0$ 
                flag = 1;  $0$ 
            }
            ++j;  $\sum_{j=i+1}^n (n-j) - 1$ 
        }
    }
    return contador;  $1$ 
}

```

La complejidad de este algoritmo en el peor caso es de $O(n^2)$

Punto 5

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

Para este algoritmo se hacen algunas restricciones entre estas están:

1. Cuando $n < 5$ el algoritmo será infinito
2. Cuando $5 \leq n < 10$, el algoritmo se ejecutará $(1+(n+1)+n+n) 3n + 2$, Por ese motivo, su complejidad será $O(n)$

```
void algoritmo5(int n){
    int i = 0; 1
    while(i <= n){ n+1
        printf("%d\n", i); n
        i += n / 5; n
    }
}
```

3. Cuando $10 \leq n < \infty$, el algoritmo se ejecutará $(1+7+6+6) 20$ veces, y en este caso su complejidad será $O(1)$

```
void algoritmo5(int n){
    int i = 0; 1
    while(i <= n){ 7
        printf("%d\n", i); 6
        i += n / 5; 6
    }
}
```

Punto 6

```
#1
def fibo (n):
    listaAns = []
    suma = 0
    for i in range(0,n+1):
        if i == 0:
            listaAns.append(0)
        elif i == 1:
            listaAns.append(1)
        else:
            suma = listaAns[len(listaAns)-1] + listaAns[len(listaAns)-2]
            listaAns.append(suma)
    ans = suma
    return ans
#print(fibo(6))
```

Según lo anterior logro concluir que la complejidad del algoritmo es $O(n)$.

| números | Replit | Bash |
|---------|--------|------|
| 5 | 30s | 85s |
| 10 | 33s | 88s |
| 15 | 37s | 93s |
| 20 | 37s | 95s |
| 25 | 39s | 97s |
| 30 | 36s | 94s |
| 35 | 31s | 95s |
| 40 | 34s | 94s |
| 45 | 32s | 95s |
| 50 | 35s | 95s |
| 60 | 33s | 95s |
| 100 | 38s | 94s |

El valor más alto para el cual obtuve su tiempo de ejecución es cuando $n = 25$, los tiempos obtenidos no fueron lo que imagine ,por el motivo de que la idea que tenia era que serían datos ascendentes en donde el menor tiempo seria cuando $n = 5$, y el mayor, cuando $n = 100$, pero me lleve la sorpresa, cuando me dio el numero mas alto en $n = 25$. De estos tiempo se puede decir que no poseen un patrón exacto, pero se puede llegare a la conclusión de que los números se mantienen en el rango desde 85 hasta 97.

Punto 7

```
#2
def fibo2 (n):
    listaAns = []
    suma = 0
    for i in range(0,n+1):
        if i == 0:
            listaAns.append(0)
        elif i == 1:
            listaAns.append(1)
        else:
            suma = listaAns[1]+ listaAns[0]
            listaAns[0]= listaAns[1]
            listaAns[1] = suma
    return suma
#print(fibo2(6))
```

El número total de ejecuciones es: $5n+3$

Según lo anterior este algoritmo sería una función lineal, por lo tanto su complejidad es $O(n)$.

| número | replit | Bash |
|--------|--------|------|
| 5 | 32s | 95s |
| 10 | 33s | 95s |
| 15 | 31s | 94s |
| 20 | 23s | 79s |
| 25 | 31s | 95s |
| 30 | 32s | 93s |
| 35 | 31s | 96s |
| 40 | 31s | 93s |
| 45 | 31s | 95s |
| 50 | 29s | 95s |
| 100 | 33s | 95s |
| 200 | 37s | 91s |
| 500 | 34s | 94s |

| | | |
|-------|-----|-----|
| 1000 | 34s | 97s |
| 5000 | 31s | 94s |
| 10000 | 29s | 95s |

Punto 8

| Números | Tiempo-Solución propia | | Tiempo-Solución profes | |
|---------|------------------------|-----------|------------------------|--------|
| | Replit | Bash | Replit | Bash |
| 100 | 1.623s | 1.595 | 0.35s | 0.131s |
| 1000 | 2.868s | 3.337s | 0.37s | 0.139s |
| 5000 | 4.688s | 3.353s | 0.49s | 0.158s |
| 10000 | 5.067s | 4.829s | 0.63s | 0.168s |
| 20000 | 10.492s | 4.897s | 0.293s | 0.181s |
| 100000 | 2m42.445s | 34.594s | 2.505s | 0.445s |
| 200000 | 9m41.928 | 1m56.852s | 6.254s | 0.929s |

- a) Los tiempos de ejecución poseen una gran diferencia, en donde el tiempo de ejecución de la solución propia tiene un valor más grande a comparación de la solución de los profes; lo anterior seguramente se deba a que la solución propia hace más trabajo del necesario, a diferencia de la solución otorgada por los profes.

b)

Solución propia:

```

def sonPrimos(N):
    ans = True
    flag = True
    i = 2
    while i < N and flag:
        if (N%i) == 0:
            ans = False
            flag = False
        i += 1
    return ans

```

Para este algoritmo está el mejor y el peor caso:

El mejor => Que $n < i$, en este caso el algoritmo se ejecutará $(1+1+1+1+1)$ 5 veces. En tal caso su complejidad será de $O(1)$

El peor => Que n sea primo y se ejecute $(1+1+1+(n)+(n-1)+1)$ $2n+3$ veces. En tal caso su complejidad será de $O(n)$

Solución profes:

```
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
```

Para este algoritmo se saca lo que es el mejor y el peor caso:

Mejor caso => Que $n < 2$, en este caso, solo se ejecutaría $(1+1+1)$ 3 veces y su complejidad sería $O(1)$

Peor caso => Que n sea primo y se ejecute $(1+2+(\lfloor \sqrt{n} \rfloor) + (\lfloor \sqrt{n} \rfloor - 1) + (\lfloor \sqrt{n} \rfloor - 1) + 1)$, $2\sqrt{n} + 2$ veces. En tal caso su complejidad será de $O(\sqrt{n})$