
ESCOM - IPN

Análisis con Big O Notation

ANÁLISIS DE ALGORITMOS 3CM3



Oscar Andrés Rosas Hernandez

Abril 2018

Índice

1. Algoritmo 1	3
1.1. Análisis Visual	3
1.2. Explicación	3
2. Algoritmo 2	4
2.1. Análisis Visual	4
2.2. Explicación	4
3. Algoritmo 3	5
3.1. Análisis Visual	5
3.2. Explicación	5
4. Algoritmo 4	6
4.1. Análisis Visual	6
4.2. Explicación	6
4.3. Algoritmo 2	7
4.4. Algoritmo 4	8
4.5. Algoritmo 5	9
5. Algoritmo 5	10
5.1. Análisis Visual	10
5.2. Explicación	10
6. Algoritmo 6 - 9	11
6.1. Análisis Visual	11
6.2. Explicación	11
7. Algoritmo 7 - 10	12
7.1. Análisis Visual	12
7.2. Explicación	12
8. Algoritmo 8 - 11	13

8.1. Análisis Visual	13
8.2. Explicación	13
9. Algoritmo 9 - 12	14
9.1. Análisis Visual	14
9.2. Explicación	14
10. Algoritmo 10 - 13	15
10.1. Análisis Visual	15
10.2. Explicación	15
11. Algoritmo 11 - 14	16
11.1. Análisis Visual	16
11.2. Explicación	16
12. Algoritmo 12 - 15	17
12.1. Análisis Visual	17
12.2. Explicación	17

1. Algoritmo 1

1.1. Análisis Visual

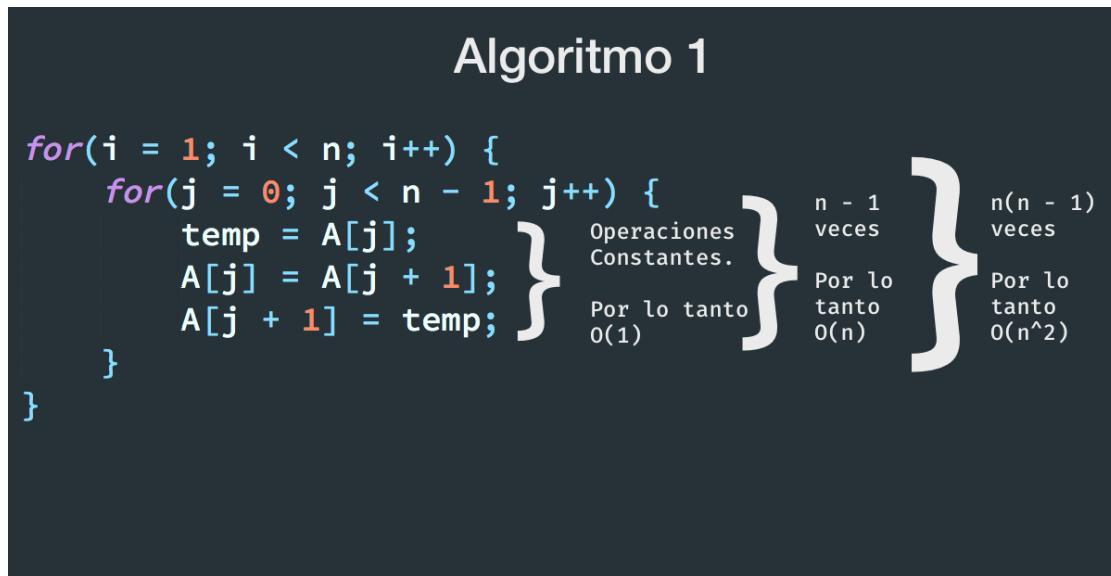


Figura 1: Análisis Visual del Algoritmo

1.2. Explicación

En esta caso el algoritmo es bastante sencillo de analizar, por un lado tenemos 2 bucles for anidados, cada uno de ellos desde cero hasta $n - 1$, por lo tanto se repetirán cada uno n veces.

Ahora lo que está dentro de los ciclos son 3 operaciones primitivas, por lo tanto todas se aproximan a una cota $O(1)$.

Finalmente podemos decir que todo el algoritmo en general tiene un orden de $O(n^2)$

2. Algoritmo 2

2.1. Análisis Visual

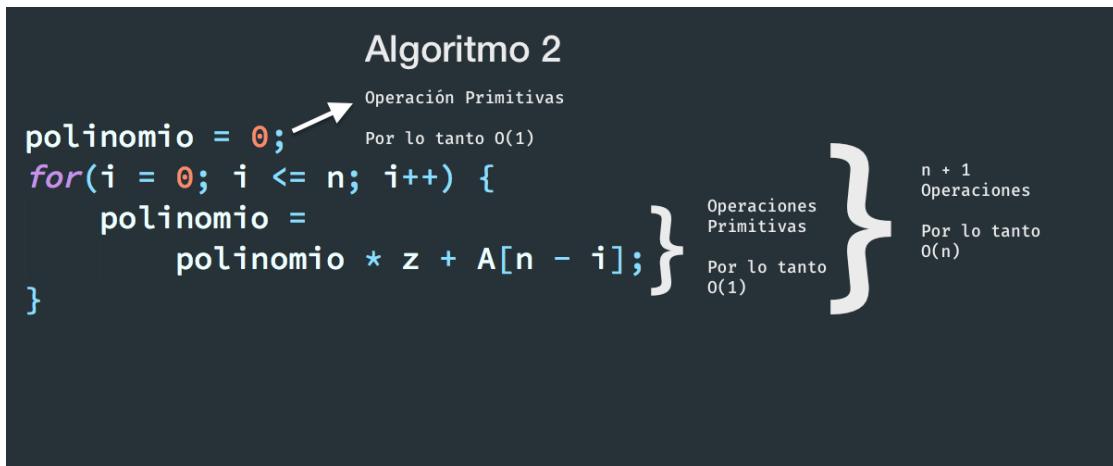


Figura 2: Análisis Visual del Algoritmo

2.2. Explicación

La primera instrucción es primitiva, por lo tanto es constante, después tenemos un polinomio que se ejecuta n veces, dentro tenemos una operación constante, por lo tanto es bastante sencillo ver que el algoritmo es en general $O(n)$

3. Algoritmo 3

3.1. Análisis Visual

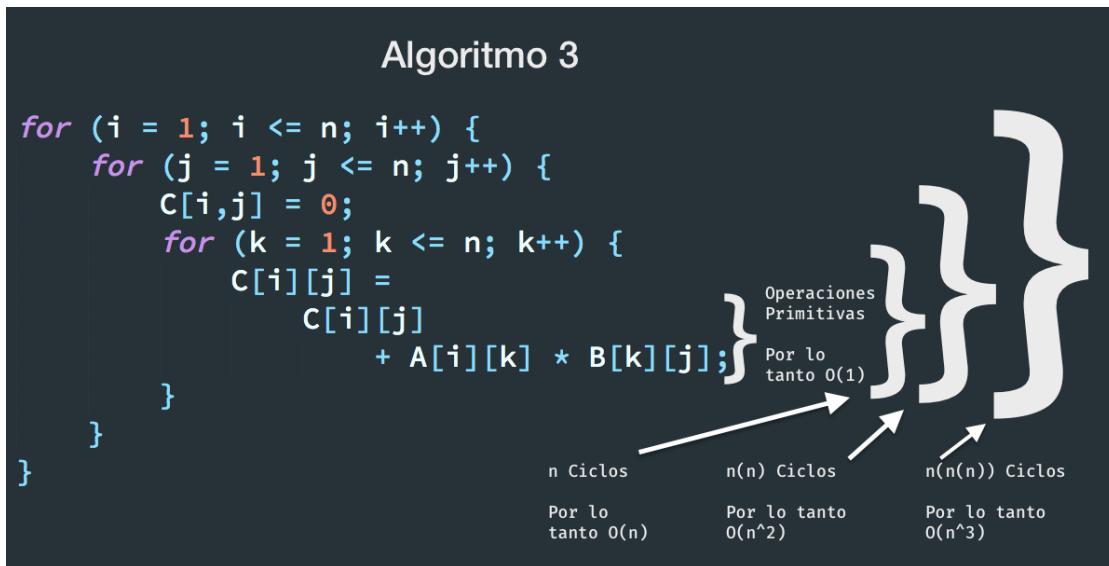


Figura 3: Análisis Visual del Algoritmo

3.2. Explicación

Tenemos 3 bucles for anidados, cada uno de ellos desde 1 hasta n , por lo tanto tenemos n operaciones dentro de cada for.

Y dentro de los 3 fors anidados tenemos algo muy loco, una asignación y multiplicación por lo tanto tenemos un algoritmo en total de $O(n^3)$ cada exponente por el bucle for

4. Algoritmo 4

4.1. Análisis Visual

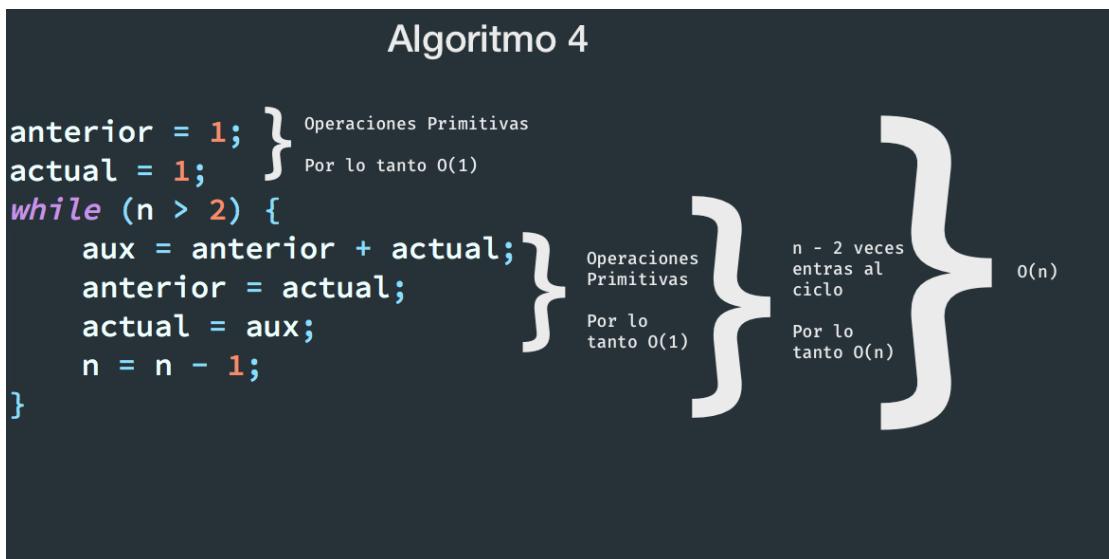


Figura 4: Análisis Visual del Algoritmo

4.2. Explicación

Ok, este esta mucho mas bueno que los otros, primero, dentro del ciclo while tenemos operaciones primitivas, por lo tanto estas serán constantes, ahora...

¿Cuántas veces se ejecuta el ciclo While? Este se ejecuta $n - 2$ veces porque en cada ciclo del while, n va decreciendo de uno a uno, hasta llegar a 2.

Ahora, en la parte superior tenemos 2 instrucciones primitivas, que nos toman siempre la misma cantidad de tiempo asintótico.

Por lo tanto en general podemos decir que este algoritmo en general es linea con respecto a n , es decir $O(n)$ pues $O(n) > O(1)$.

4.3. Algoritmo 2

```

1 polinomio = 0;
2     for (i = 0; i <= n; i++) {
3         polinomio = polinomio * z + A[n - i];
4     }

```

- Veamos primero la linea: $\text{polinomio} = \text{polinomio} * z + A[n - i];$

Esta esta algo en conglomerada, pues primero es:

- 1 por $n - i$
- 1 por $\text{polinomio} * z$
- 1 por suma de ambas
- 1 por asignación

Por lo tanto cuesta 4 unidades

- Todo lo anterior esta encerrado en un for generico, asi que vamos a calcularlo:

$$\begin{aligned}
Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + 4) \\
&= 3 + NumInnerFor(8)
\end{aligned}$$

Ahora sabemos que $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-0}{1} + 1 \right\rfloor = \left\lfloor \frac{n}{1} + 1 \right\rfloor = n + 1$

Por lo tanto en general $Cost = 8(n + 1) + 3 = 8n - 12$

Finalmente tenemos que:

- $Cost_{Temporal} = 8n + 12$
- $Cost_{Espacial} = n + 3$

Porque pues usamos un arreglo de $n+1$ elementos (para hacer $A[n]$), espacio para temp y espacio para 1 interador

4.4. Algoritmo 4

```

1 anterior = 1;
2 actual = 1;
3 while (n > 2) {
4     aux = anterior + actual;
5     anterior = actual;
6     actual = aux;
7     n = n - 1;
8 }
```

Antes que nada, vamos a ponerlo en un modo de for, para que todo sea mas sencillo:

```

1 anterior = 1;
2 actual = 1;
3
4 for (i = 3; i <= n; i++) {
5     aux = anterior + actual;
6     anterior = actual;
7     actual = aux;
8 }
```

1. Veamos primero lo que esta dentro del for

Esta esta algo enconglomerada, pues primero es:

- 1 por suma y otro por asignación
- 1 por asignación
- 1 por asignación

Por lo tanto cuesta 4 unidades

2. Todo lo anterior esta encerrado en un for generico, asi que vamos a calcularlo:

$$\begin{aligned}
 Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
 &= 3 + 0 + 0 + NumInnerFor(0 + 4 + 4) \\
 &= 3 + NumInnerFor(8)
 \end{aligned}$$

Ahora sabemos que $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-3}{1} + 1 \right\rfloor = \left\lfloor \frac{n-3}{1} + 1 \right\rfloor = n - 2$

Por lo tanto en general $Cost = 8(n - 2) + 3 = 8n - 13$

Finalmente tenemos que:

- $Cost_{Temporal} = 8n - 13 + 4 = 8n - 9$
 - $Cost_{Espacial} = 3$
- Porque pues usamos 3 variables nada mas, 2 normales y 1 interador

4.5. Algoritmo 5

```

1 for (i = n - 1, j = 0; i >= 0; i--, j++)
2     s2[j] = s[i];
3
4 for (i = 0, i < n; i++)
5     s[i] = s2[i];

```

Ok, esta medio raro el codigo, vamos a ponerlo bonito:

```

1 j = 0;
2 for (i = 0; i <= n - 1; i++) {
3     s2[j] = s[n - 1 - i];
4     j++;
5 }
6
7 for (i = 0, i <= n - 1; i++)
8     s[i] = s2[i];

```

Ok, esta esta muy sencilla, porque son sencillamente 2 fors

1. Internamente lo que pasa dentro del primer for es que la primera linea son 2 operaciones aritmética y 1 asignación en la primera linea y una operación y otra asignación, por lo tanto calculamos 5 operaciones
2. Todo lo anterior esta encapsulado en un for generico, asi que vamos a calcularlo:

$$\begin{aligned}
Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + 3) \\
&= 3 + NumInnerFor(3)
\end{aligned}$$

Ahora sabemos que $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-1-0}{1} + 1 \right\rfloor = \left\lfloor \frac{n-1}{1} + 1 \right\rfloor = n$

Por lo tanto en general $Cost = n(7) + 3 = 7n + 3$

3. La siguiente parte del algoritmo es otro pequeño y simple for generico:

$$\begin{aligned}
Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + 1) \\
&= 3 + NumInnerFor(5)
\end{aligned}$$

Ahora sabemos que $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-1-0}{1} + 1 \right\rfloor = \left\lfloor \frac{n-1}{1} + 1 \right\rfloor = n$

Finalmente tenemos que:

- $Cost_{Temporal} = 5n + 3 + 7n + 3 = 12n + 6$
 - $Cost_{Espacial} = 2n + 2$
- Porque pues usamos 2 iterados, y 2 arreglos mínimo n elementos

5. Algoritmo 5

5.1. Análisis Visual

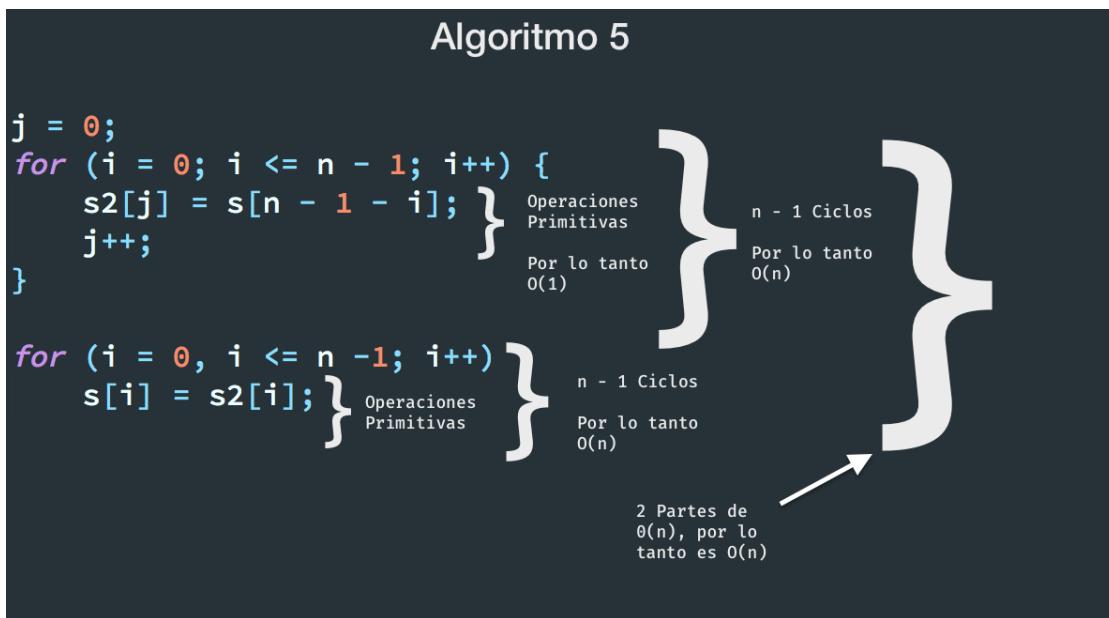


Figura 5: Análisis Visual del Algoritmo

5.2. Explicación

Antes que nada, no te preocupes, no es que haya modificado mucho el algoritmo, lo único que hice fue hacer que los bucles crezcan, y separar un poco a la variable j .

Ahora si, tenemos 2 ciclos for, por lo tanto la cota $O()$ estará dada por la suma de ambos costos de los ciclos:

- En el primer ciclo tenemos solo operaciones primitivas dentro del for, y este irá de 0 hasta $n - 1$ por lo tanto podemos concluir que el primer ciclo tiene un costo de $O(n)$
- En el segundo ciclo tenemos solo operaciones primitivas dentro del for, y este irá también de 0 hasta $n - 1$ por lo tanto podemos concluir que el segundo ciclo tiene un costo de $O(n)$

Por lo tanto tenemos que el costo total es $O(n) + O(n) = O(n)$

6. Algoritmo 6 - 9

6.1. Análisis Visual

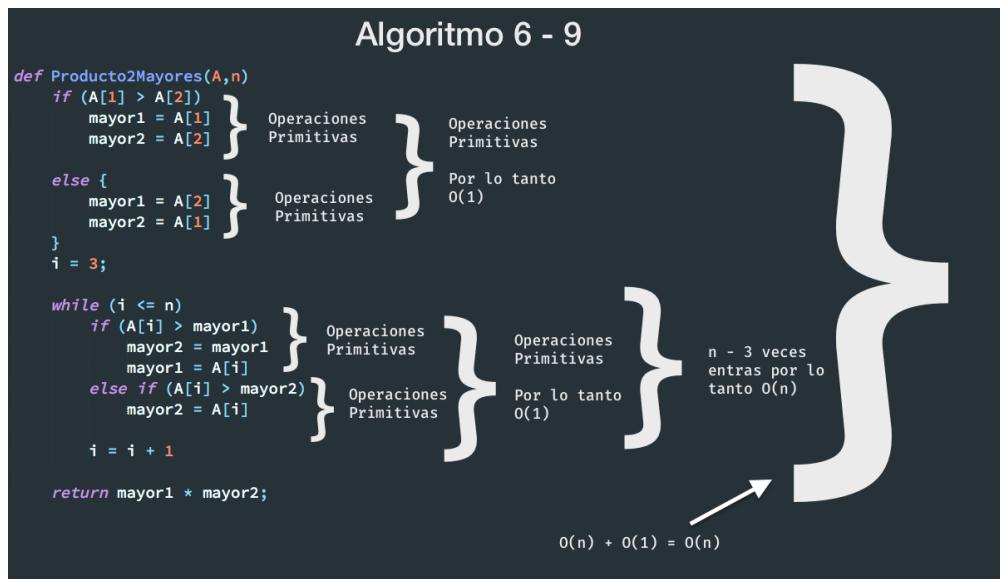


Figura 6: Análisis Visual del Algoritmo

6.2. Explicación

Ahora si estamos empezando a ver algoritmos mucho más interesantes,

Ahora si, tenemos 2 bloques de código dentro de la función, por lo tanto la cota $O()$ de la función completa estará dada por la suma de ambos costos:

- Bloque de código 1, en el peor de los casos (después de todo de eso también se trata $O()$) tenemos operaciones primitivas, de hecho, sin importar como sea nuestro vector en si, por lo tanto el primer bloque tiene un costo de $O(1)$
- En el segundo bloque de código esta un poco más interesante, tenemos una variable i que ira creciendo desde 3 hasta n , por lo tanto se ejecutará $n - 3$ veces el código interno, pero lo curioso es que dentro del bucle while solo tenemos operaciones primitivas, sin importar como sea nuestra información el costo del interior del bucle será $O(1)$.

Por lo tanto el costo de este bloque será de $O(n)$

Por lo tanto tenemos que el costo total es $O(n) + O(1) = O(n)$

7. Algoritmo 7 - 10

7.1. Análisis Visual

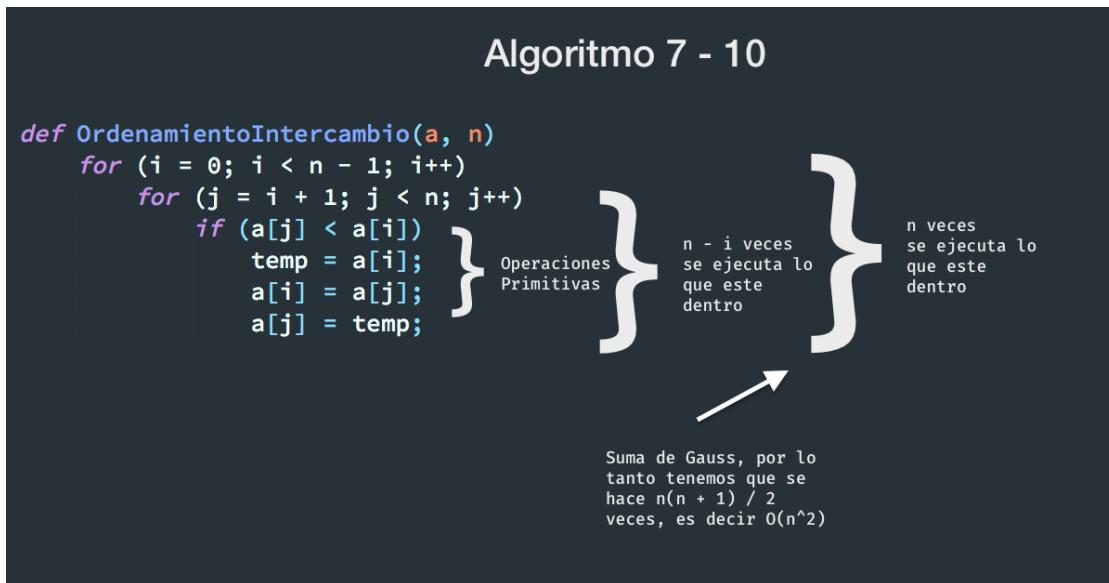


Figura 7: Análisis Visual del Algoritmo

7.2. Explicación

Nuestro primero algoritmo de ordenamiento, ¡Qué emoción!

Vamos de adentro hacia afuera, por un lado, primero, en el peor de los casos nuestro algoritmo siempre entrará dentro del if, dentro solo tenemos operaciones básicas, por lo tanto incluso si estamos lo que esta dentro de segundo for tiene un costo constante, es decir $O(1)$.

Ahora este código constante se hará primero desde 0 hasta n , luego de 1 hasta n , y así, por lo tanto podemos ver que si lo ordenamos al revés tenemos que se ejecutará $\sum_{i=1}^n i$ veces pero lo lindo es que ese resultado es muy conocido, entonces entre los dos fors el código constante se hará $\frac{n(n+1)}{2}$, que si aplicamos el análisis sintótico tenemos que tiene un costo de $O(n^2)$ al final y al cabo.

8. Algoritmo 8 - 11

8.1. Análisis Visual

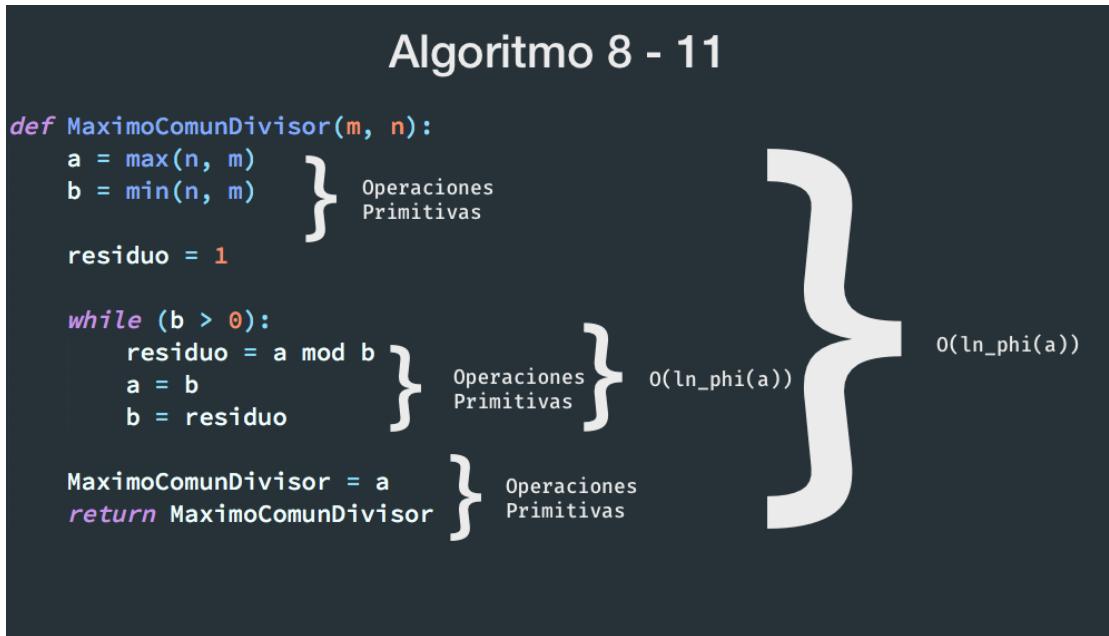


Figura 8: Análisis Visual del Algoritmo

8.2. Explicación

Para obtener el $O()$ tenemos que ver como es que se comporta nuestro algoritmo en el peor de los casos mientras nuestro tamaño del problema crece, así que vamos primero a recordar lo que dije en el trabajo pasado:

Ok, tengo que admitir que este es un problema conocido, es el clásico algoritmo de Euclides y hay un caso muy feo para ese algoritmo, y es que sean dos número de Fibonacci consecutivos.

Vamos a dar una explicación de porque: Por definición $F_k = F_{k-1} + F_{k-2}$, por lo tanto cuanto lo divides tienes que que el cociente siempre sera 1 en cada iteración, por lo tanto tardaremos k divisiones en llegar a k , y como los números de Fibonacci se aproximan a ϕ^k entonces tenemos que el peor caso será $f(n) = \log \phi(n)$

Así que ahora, conociendo la función complejidad del peor caso es trivial ver que tiene un costo de $O(\log_\phi(n))$

9. Algoritmo 9 - 12

9.1. Análisis Visual

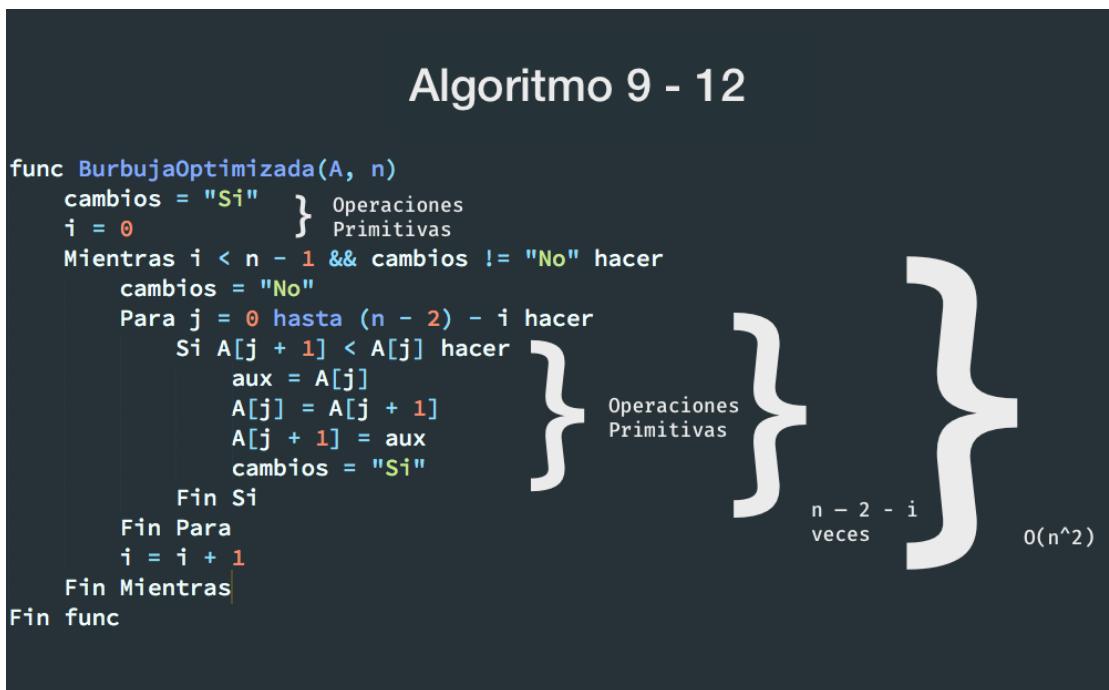


Figura 9: Análisis Visual del Algoritmo

9.2. Explicación

Ok, voy creo que ya vamos teniendo experiencia para ver que las primeras 2 instrucciones son primitivas por lo tanto no importa mucho que es lo que pase con ellas, sino lo que pasa dentro del ciclo.

Ahora, claro, en el mejor de los casos la variable de cambio podría hacer que salieramos repentinamente del ciclo, pero en el peor de los casos, que es lo que estamos haciendo esta variable nunca pinta de nada.

Y si eliminamos esa variable / bandera tenemos que volvemos al mismo algoritmo de burbuja, con sus sumas de Gauss que ya lo resolví en otros ejercicios.

Así que creo que con estos argumentos tiene mucho sentido que este algoritmo siga siendo $O(n^2)$

10. Algoritmo 10 - 13

10.1. Análisis Visual

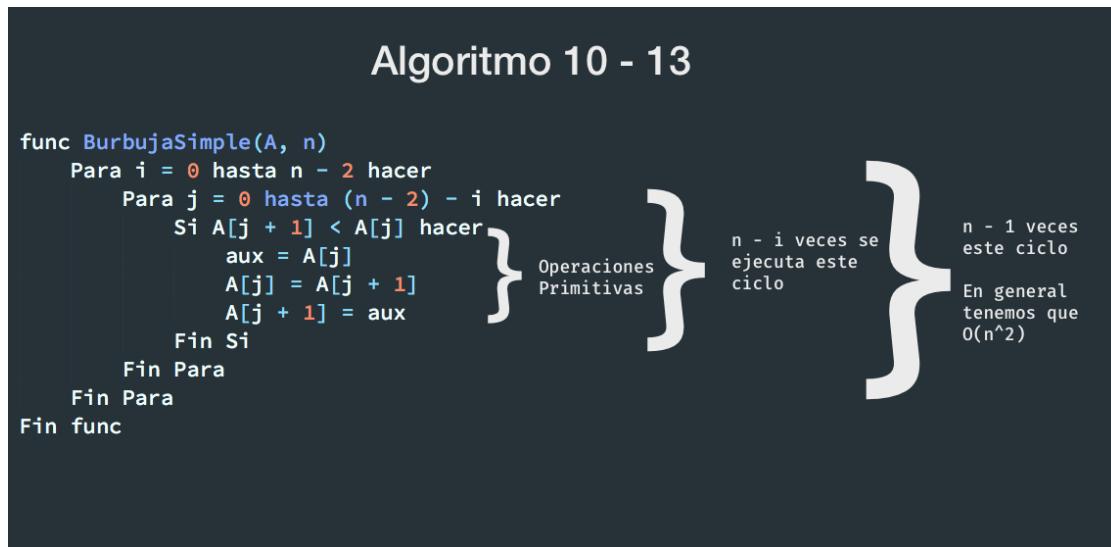


Figura 10: Análisis Visual del Algoritmo

10.2. Explicación

Nuestro segundo algoritmo de ordenamiento, ¡Qué emoción!

Vamos de adentro hacia afuera, por un lado, primero, en el peor de los casos nuestro algoritmo siempre entrará dentro del if, dentro solo tenemos operaciones básicas, por lo tanto incluso si estamos lo que esta dentro de segundo for tiene un costo constante, es decir $O(1)$.

Ahora este código constante se hará primero desde 0 hasta n , luego de 1 hasta n , y así, por lo tanto podemos ver que si lo ordenamos al revés tenemos que se ejecutará $\sum_{i=1}^n i$ veces pero lo lindo es que ese resultado es muy conocido, entonces entre los dos fors el código constante se hará $\frac{n(n+1)}{2}$, que si aplicamos el análisis sintótico tenemos que tiene un costo de $O(n^2)$ al final y al cabo.

11. Algoritmo 11 - 14

11.1. Análisis Visual

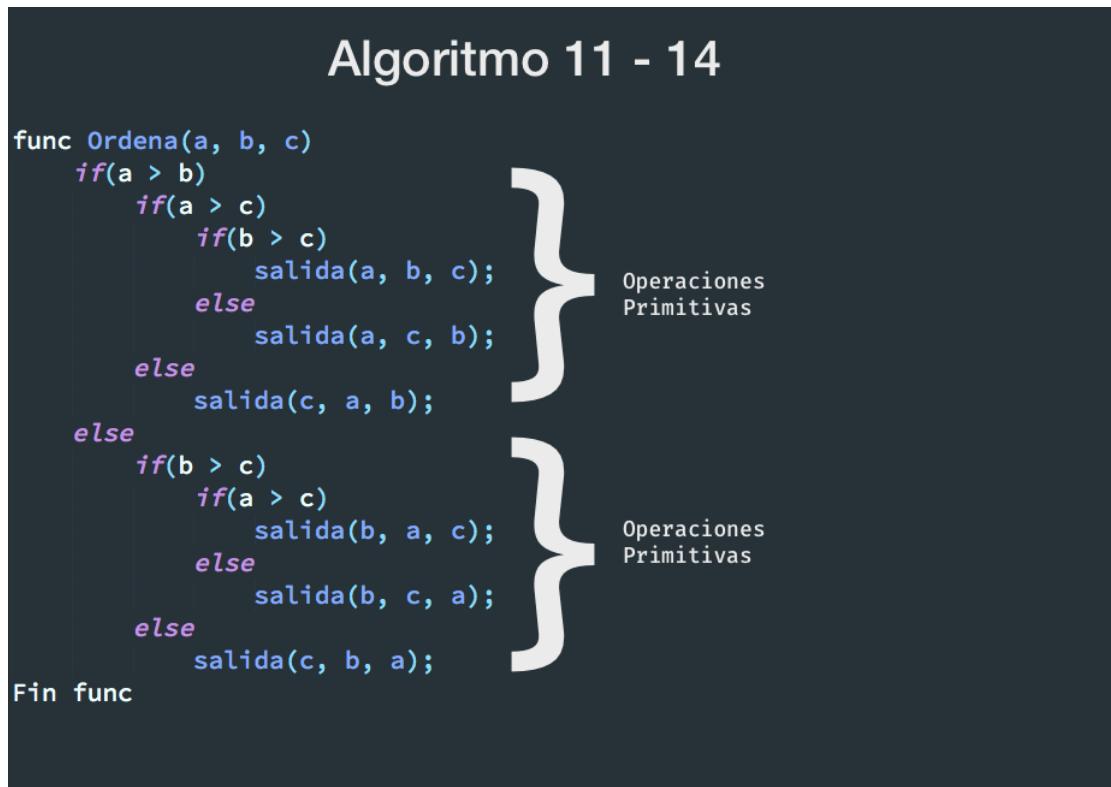


Figura 11: Análisis Visual del Algoritmo

11.2. Explicación

Ok, parece que vayamos hecho una soberana estupidez, de análisis, pero dejame explicar por un lado ¿Que es lo peor que puede pasar?

Pues nada, osea, que sin importar que pase tienes el mismo costo, operaciones elementales, sin importar que tan grandes sean a, b, c su comparación no cuesta diferente.

Por lo tanto tenemos que este es un algoritmo de ordenamiento sencillo de $O(1)$, el único algoritmo de ordenamiento que he visto así.

12. Algoritmo 12 - 15

12.1. Análisis Visual

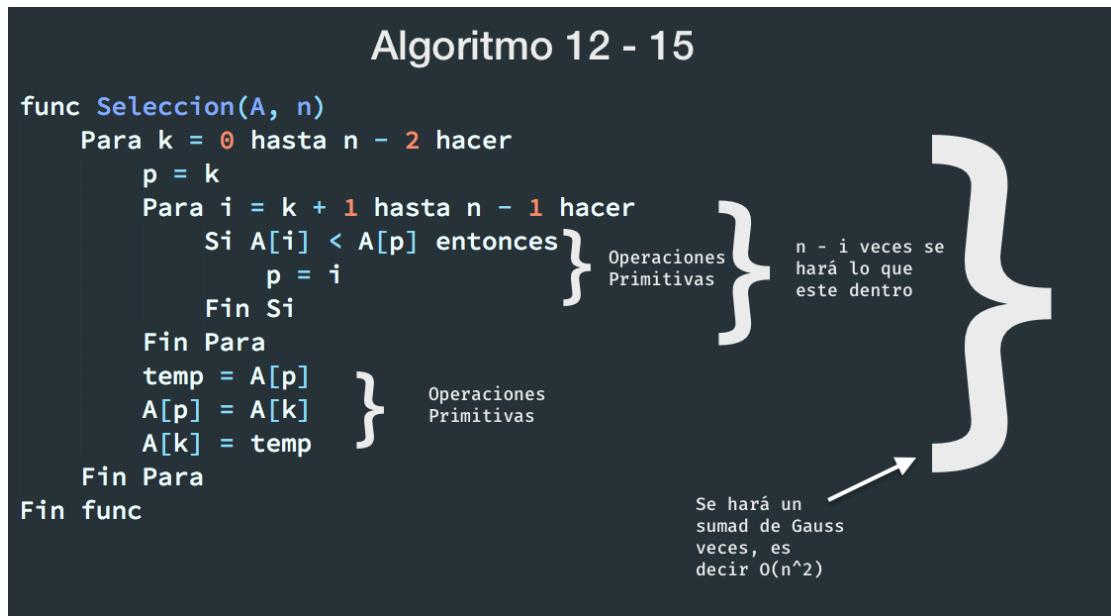


Figura 12: Análisis Visual del Algoritmo

12.2. Explicación

Vamos de adentro hacia afuera, por un lado, primero, en el peor de los casos nuestro algoritmo siempre entrará dentro del if, dentro solo tenemos operaciones básicas, por lo tanto incluso si estamos lo que esta dentro de segundo for tiene un costo constante, es decir $O(1)$.

Ahora este código constante se hará primero desde 0 hasta n , luego de 1 hasta n , y así, por lo tanto podemos ver que si lo ordenamos al revés tenemos que se ejecutará $\sum_{i=1}^n i$ veces pero lo lindo es que ese resultado es muy conocido, entonces entre los dos fors el código constante se hará $\frac{n(n+1)}{2}$, que si aplicamos el análisis sintótico tenemos que tiene un costo de $O(n^2)$ al final y al cabo.