

---

ESCOM - IPN

# Complejidad de Algoritmos

ANÁLISIS DE ALGORITMOS 3CM3



Oscar Andrés Rosas Hernandez

Marzo 2018

# Índice

<b>1. For Genérico</b>	<b>3</b>
1.1. Calculo de Veces que Entra . . . . .	3
1.2. Variante Multiplicativa . . . . .	4
1.3. Calculo de Veces que Entra . . . . .	4
1.4. Costo Temporal de un For Generico . . . . .	5
<b>2. Función Complejidad Temporal - Espacial</b>	<b>6</b>
2.1. Algoritmo 1 . . . . .	7
2.2. Algoritmo 2 . . . . .	8
2.3. Algoritmo 3 . . . . .	9
2.4. Algoritmo 4 . . . . .	11
2.5. Algoritmo 5 . . . . .	12
<b>3. Aproximación de Número de Prints</b>	<b>13</b>
3.1. Algoritmo 6 . . . . .	14
3.2. Algoritmo 7 . . . . .	16
3.3. Algoritmo 8 . . . . .	19
<b>4. Análisis de Casos de Prints</b>	<b>20</b>
4.1. Algoritmo 9 . . . . .	21
4.1.1. Mejor Caso . . . . .	21
4.1.2. Peor Caso . . . . .	22
4.1.3. Caso Medio . . . . .	22
4.2. Algoritmo 10 . . . . .	23
4.2.1. Mejor Caso . . . . .	23
4.2.2. Peor Caso . . . . .	23
4.2.3. Caso Medio . . . . .	23
4.3. Algoritmo 11 . . . . .	24
4.3.1. Mejor Caso . . . . .	24
4.3.2. Peor Caso . . . . .	24

4.3.3. Caso Medio . . . . .	24
4.4. Algoritmo 12 . . . . .	25
4.4.1. Mejor Caso . . . . .	25
4.4.2. Peor Caso . . . . .	25
4.4.3. Caso Medio . . . . .	26
4.5. Algoritmo 13 . . . . .	27
4.5.1. Mejor Caso . . . . .	27
4.5.2. Peor Caso . . . . .	27
4.5.3. Caso Medio . . . . .	28
4.6. Algoritmo 14 . . . . .	29
4.6.1. Mejor Caso . . . . .	30
4.6.2. Peor Caso . . . . .	30
4.6.3. Caso Medio . . . . .	30
4.7. Algoritmo 15 . . . . .	31
4.7.1. Mejor Caso . . . . .	31
4.7.2. Peor Caso . . . . .	31
4.7.3. Caso Medio . . . . .	31

## 1. For Genérico

Vamos a usar la idea de Ontiveros Salazar para simplificar los trabajos a posteriori con su famoso análisis del ciclo for, para esto considera el siguiente ciclo:

Donde  $Start, End, Jump \in \mathbb{Z}$ ,  $Start \leq End$

```
1 for (int i = Start; i <= End, i = i + Jump) {  
2     //Do something  
3 }
```

### 1.1. Calculo de Veces que Entra

Ahora, nuestra  $i$  irá tomando valores, estos son una secuencia que sale inmediato de la definición del for:

$$i = Start + k(Jump - 1) \quad \text{donde } k \in \mathbb{N}$$

Ahora veamos cuantas de estas veces  $i$  toma valores para los cuales se cumpla la condiciones necesarias que  $Start \leq End$

$$Start \leq i \leq End$$

$$Start \leq Start + Jump(k - 1) \leq End$$

$$Start - Start \leq Start + Jump(k - 1) - Start \leq End - Start$$

$$0 \leq Jump(k - 1) \leq End - Start$$

$$0 \leq (k - 1) \leq \frac{End - Start}{Jump}$$

$$1 \leq k \leq \frac{End - Start}{Jump} + 1$$

Y bingo, claro, la cantidad de veces que se ejecuta algo es un número entero, por lo tanto simplemente basta con ver que:

$$\# \text{ Veces que se llega a Do something} = \left\lfloor \frac{End - Start}{Jump} + 1 \right\rfloor$$

## 1.2. Variante Multiplicativa

Ve este for bien bonito:

```

1 for (int i = Start; i <= End, i = i * Jump) {
2     //Do something
3 }
```

## 1.3. Calculo de Veces que Entra

Ahora, nuestra  $i$  irá tomando valores, estos son una secuencia que sale inmediato de la definición del for:

$$i = Start(Jump)^{k-1} \quad \text{donde } k \in \mathbb{N}$$

Ahora veamos cuantas de estas veces  $i$  toma valores para los cuales se cumpla la condiciones necesarias que  $Start \leq End$

$$Start \leq i \leq End$$

$$Start \leq Start(Jump)^{k-1} \leq End$$

$$1 \leq (Jump)^{k-1} \leq \frac{End}{Start}$$

$$1 \leq \log_{Jump}((Jump)^{k-1}) \leq \log_{Jump}\left(\frac{End}{Start}\right)$$

$$1 \leq k - 1 \leq \log_{Jump}\left(\frac{End}{Start}\right)$$

$$1 \leq k \leq \log_{Jump}\left(\frac{End}{Start}\right) + 1$$

Y bingo, claro, la cantidad de veces que se ejecuta algo es un número entero, por lo tanto simplemente basta con ver que:

$$\# \text{ Veces que se llega a Do something} = \left\lfloor \log_{Jump}\left(\frac{End}{Start}\right) + 1 \right\rfloor$$

## 1.4. Costo Temporal de un For Generico

Sea  $CostFor$  el costo que nos cuesta un for generico, vamos a ir añadiendo pieza a pieza los componentes.

$$\text{Sea } NumInnerFor = \left\lfloor \frac{\text{End} - \text{Start}}{\text{Jump}} + 1 \right\rfloor$$

Ok, ahora que ya tenemos las piezas más importantes listas basta con ver que:

- Hay un asignación inicial a  $Start$ , por lo tanto  $CostFor = 1 + Cost(Start)$
- El número de comparaciones de  $i$  con  $End$  es igual al número de ejecuciones de  $DoSomething$  más una extra, es decir,  $NumInnerFor + 1$  Si el costo de calcular cada vez  $End$  es  $Cost(End)$  entonces

$$Cost = 1 + Cost(Start) + Cost(End)[NumInnerFor + 1]$$

- El número de incrementos en  $i$  es igual al número de veces que si entra dentro del for, por lo tanto

$$Cost = 1 + Cost(Start) + Cost(End)[NumInnerFor + 1] + 2CostInnerFor$$

- El número de saltos implícitos es el número de veces que se ejecuta lo de adentro del for más uno, por lo tanto

$$Cost = 1 + Cost(Start) + Cost(End)[NumInnerFor + 1] + 3CostInnerFor + 1$$

- Va a entrar  $NumInnerFor$  veces dentro así que:

$$Cost = 1 + Cost(Start) + Cost(End)[NumInnerFor + 1] + 3CostInnerFor + 1 + NumInnerFor(Cost(DoSomething))$$

- Recueda que ya calculamos cuantas veces vamos a calcular a  $End$ , pero nos faltan las comparaciones que será  $NumInnerFor + 1$  veces.

Por lo tanto:

$$Cost = 1 + Cost(Start) + Cost(End)[NumInnerFor + 1] + 4CostInnerFor + 2 + NumInnerFor(Cost(DoSomething))$$

Es decir:

$$Cost = 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething))$$

## 2. Función Complejidad Temporal - Espacial

Para los siguientes 5 algoritmos determine la función de complejidad temporal y espacial en términos de n.

Considere las operaciones de:

- Asignación
- Aritméticas
- Condicionales
- Saltos Implícitos

## 2.1. Algoritmo 1

```

1  for ( i = 1; i < n; i++ ) {
2      for ( j = 0; j < n - 1; j++ ) {
3          temp = A[ j ];
4          A[ j ] = A[ j + 1 ];
5          A[ j + 1 ] = temp ;
6      }
7  }
```

- Veamos que es lo que pasa primero dentro del ciclo for interior, estamos haciendo 3 lineas, pero la segunda y la tercera no es solo una asignación pero también una operación aritmética al hacer  $j + 1$ , por lo tanto esto nos costara 5 donde  $u$  es una unidad de tiempo.
- Todo lo anterior esta encerrado en un for genérico, Ahora recuerda que el costo de calcular  $Start$  es 0, pero  $Cost(End) = 1$ , así que vamos a calcularlo:

$$\begin{aligned}
Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
&= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + 5) \\
&= 3 + 0 + Cost(End) + NumInnerFor(Cost(End) + 4 + 5) \\
&= 3 + 0 + 1 + NumInnerFor(1 + 4 + 5) \\
&= 3 + 0 + 1 + NumInnerFor(1 + 4 + 5) \\
&= 4 + NumInnerFor(10)
\end{aligned}$$

Ahora sabemos que  $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-2-0}{1} + 1 \right\rfloor = \left\lfloor \frac{n-2}{1} + 1 \right\rfloor = n - 1$

Por lo tanto en general  $Cost = 10(n - 1) + 4 = 10n - 6$

- Ahora podemos calcular el costo del for exterior, lo cual tenemos que:

$$\begin{aligned}
Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + Cost(DoSomething)) \\
&= 3 + 0 + 0 + NumInnerFor(0 + 4 + 10n - 6) \\
&= 3 + NumInnerFor(10n - 2)
\end{aligned}$$

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

Por lo tanto en general  $Cost_{Temporal} = (10n - 2)(n - 1) + 3 = 10n^2 + 12n + 5$

Finalmente tenemos que:

- $Cost_{Temporal} = (10n - 2)(n - 1) + 3 = 10n^2 + 12n + 5$
- $Cost_{Espacial} = n + 3$

Porque pues usamos un arreglo de  $n$  elementos, espacio para temp y espacio para 2 interadores

## 2.2. 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:

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

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

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

Finalmente tenemos que:

- $\text{Cost}_{Temporal} = 8n + 12$
- $\text{Cost}_{Espacial} = n + 3$

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

### 2.3. Algoritmo 3

```

1  for (i = 1; i <= n; i++) {
2      for (j = 1; j <= n; j++) {
3          C[i, j] = 0;
4          for (k = 1; k <= n; k++) {
5              C[i][j] = C[i][j] + A[i][k] * B[k][j];
6          }
7      }
8  }
```

Tenemos 3 for anidados, por lo tanto vamos desde adentro hasta afuera:

- Veamos primero la linea:  $C[i][j] = C[i][j] + A[i][k] * B[k][j]$ ;

Esta esta algo en conglomerada, pues primero es:

- 2 operaciones
- 1 por asignación

Por lo tanto cuesta 3 unidades

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

$$\begin{aligned}
 Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
 &= 3 + 0 + 0 + NumInnerFor(0 + 4 + 3) \\
 &= 3 + NumInnerFor(7)
 \end{aligned}$$

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

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

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

$$\begin{aligned}
 Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
 &= 3 + 0 + 0 + NumInnerFor(0 + 4 + 7n + 3 + 1) \\
 &= 3 + NumInnerFor(7n + 8)
 \end{aligned}$$

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

Por lo tanto en general  $Cost = 3 + n(7n + 8) = 7n^2 + 8n + 3$

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

$$\begin{aligned}
 Cost &= 3 + Cost(Start) + Cost(End) + NumInnerFor(Cost(End) + 4 + Cost(DoSomething)) \\
 &= 3 + 0 + 0 + NumInnerFor(0 + 4 + 7n^2 + 8n + 3) \\
 &= 3 + NumInnerFor(7n^2 + 8n + 7)
 \end{aligned}$$

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

Por lo tanto en general  $Cost = 3 + n(7n^2 + 8n + 7) = 7n^3 + 8n^2 + 7n + 3$

Finalmente tenemos que:

- $Cost_{Temporal} = 7n^3 + 8n^2 + 7n + 3$

- $Cost_{Espacial} = 3n^2 + 3$

Porque pues usamos un 3 arreglos de  $(n)(n)$  elementos (para hacer  $A[n][n]$ ), espacio para 3 interadores

## 2.4. 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

## 2.5. 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

### 3. Aproximación de Número de Prints

Para los siguientes 3 algoritmos determine el número de veces que se imprime la palabra “Algoritmos”. Determine una función lo más cercana a su comportamiento para cualquier  $n$  y compruébelo codificando los tres algoritmos (Adjuntar códigos). De una tabla de comparación de sus pruebas para  $n$ 's igual a 10, 100, 1000, 5000 y 100000 y demostrar lo que la función encontrada determina será el número de impresiones.

### 3.1. Algoritmo 6

Considera el siguiente algoritmos

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5     scanf( "%d", &n);
6
7     int NumberOfPrints = 0;
8
9     for( int i = 10; i < n * 5; i *= 2) {
10        //printf("Algoritmos\n");
11        NumberOfPrints++;
12    }
13
14    printf("Number of Prints: %d\n", NumberOfPrints);
15    return 0;
16 }
```

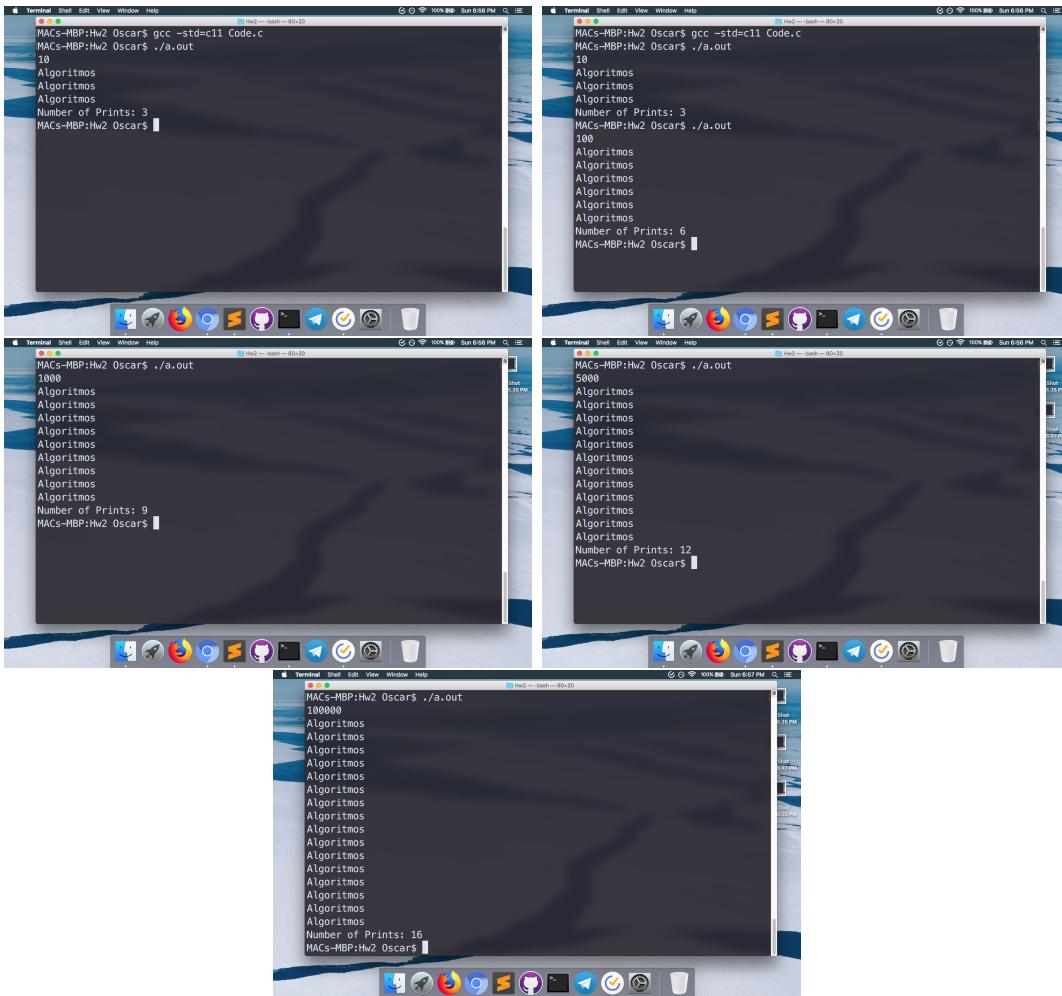
Este algoritmo es simplemente un for generico, vamos a calcularlo:

$$\begin{aligned} \text{Numero de Veces} &= \left\lfloor \log_{Jump} \left( \frac{End}{Start} \right) + 1 \right\rfloor \\ &= \left\lfloor \log_2 \left( \frac{5n - 1}{10} \right) + 1 \right\rfloor \end{aligned}$$

Ahora vamos a resolver para algunas  $n$  y veamos como se compara con la realidad:

	Numero de N	Resultado Real	Resultado Teorico
10	3		$\left\lfloor \log_2 \left( \frac{5(10) - 1}{10} \right) + 1 \right\rfloor = \lfloor \log_2(4.9) + 1 \rfloor = \lfloor 2.29 + 1 \rfloor = 3$
100	6		$\left\lfloor \log_2 \left( \frac{5(100) - 1}{10} \right) + 1 \right\rfloor = \lfloor \log_2(49.9) + 1 \rfloor = \lfloor 5.54 + 1 \rfloor = 6$
1000	9		$\left\lfloor \log_2 \left( \frac{5(1000) - 1}{10} \right) + 1 \right\rfloor = \lfloor \log_2(499.9) + 1 \rfloor = \lfloor 8.96 + 1 \rfloor = 9$
5000	12		$\left\lfloor \log_2 \left( \frac{5(5000) - 1}{10} \right) + 1 \right\rfloor = \lfloor \log_2(2499.9) + 1 \rfloor = \lfloor 11.28 + 1 \rfloor = 12$
100000	16		$\left\lfloor \log_2 \left( \frac{5(100000) - 1}{10} \right) + 1 \right\rfloor = \lfloor \log_2(49999.9) + 1 \rfloor = \lfloor 15.6 + 1 \rfloor = 16$

Vamos a mostar ahora las evidencias:



### 3.2. Algoritmo 7

Considera el siguiente algoritmos

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5     scanf( "%d", &n);
6
7     int NumberOfPrints = 0;
8
9     for( int j = n / 2; j >= 1; j /= 2) {
10        for( int i = 0; i < n; i += 2) {
11            // printf("\Algoritmos\n");
12            count++;
13        }
14    }
15
16    printf("Number of Prints: %d\n", NumberOfPrints);
17    return 0;
18 }
```

1. Vamos primero por el for generico interno:  $NumInnerFor = \left\lfloor \frac{End-Start}{Jump} + 1 \right\rfloor = \left\lfloor \frac{n-1-0}{2} + 1 \right\rfloor = \left\lfloor \frac{n}{2} \right\rfloor$

2. Ahora podemos calcular del for exterior, lo cual tenemos que:

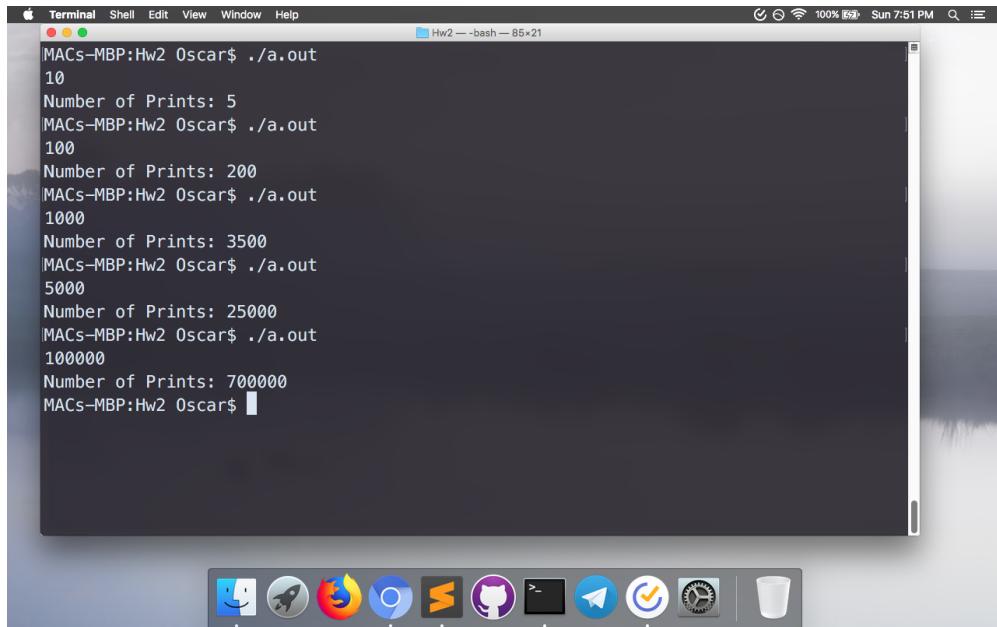
$$NumInnerFor = \left\lfloor \log_{Jump} \left( \frac{End}{Start} \right) + 1 \right\rfloor = \left\lfloor \log_2 \left( \frac{n}{2} \right) \right\rfloor$$

Finalmente tenemos que Numero de Prints =  $\text{Floor} \frac{n}{2} \left\lfloor \log_2 \left( \frac{n}{2} \right) \right\rfloor$

Ahora vamos a resolver para algunas  $n$  y veamos como se compara con la realidad:

Numero de N	Resultado Real	Resultado Teorico
10	5	$\lfloor \frac{10}{2} \rfloor \left\lfloor \log_2 \left( \frac{\frac{10}{2}}{2} \right) \right\rfloor = 5 \lfloor \log_2(2.5) \rfloor = 5 \lfloor 1.32 \rfloor = 5$
100	200	$\lfloor \frac{100}{2} \rfloor \left\lfloor \log_2 \left( \frac{\frac{100}{2}}{2} \right) \right\rfloor = 50 \lfloor \log_2(25) \rfloor = 50 \lfloor 4.64 \rfloor = 200$
1000	3500	$\lfloor \frac{1000}{2} \rfloor \left\lfloor \log_2 \left( \frac{\frac{1000}{2}}{2} \right) \right\rfloor = 500 \lfloor \log_2(250) \rfloor = 500 \lfloor 7.96 \rfloor = 3500$
5000	25000	$\lfloor \frac{5000}{2} \rfloor \left\lfloor \log_2 \left( \frac{\frac{5000}{2}}{2} \right) \right\rfloor = 2500 \lfloor \log_2(1250) \rfloor = 2500 \lfloor 10.28 \rfloor = 25000$
100000	700000	$\lfloor \frac{100000}{2} \rfloor \left\lfloor \log_2 \left( \frac{\frac{100000}{2}}{2} \right) \right\rfloor = 50000 \lfloor \log_2(14.609) \rfloor = 50000 \lfloor 14.609 \rfloor = 700000$

Vamos a mostar ahora las evidencias:



A screenshot of a macOS desktop environment. At the top, there's a dark grey menu bar with the 'Terminal' application selected. Below it is a white system status bar showing battery level (100%), signal strength, and the date and time (Sun 7:51 PM). The main window is a Terminal session titled 'Hw2 — bash — 85x21'. The user has run the command `./a.out` multiple times, each time increasing the number of prints. The output shows the following sequence of numbers: 10, 5, 100, 200, 1000, 3500, 5000, 25000, 100000, and 700000. The terminal window has a dark grey background and white text. The Dock at the bottom of the screen contains icons for various applications, including Finder, Mail, Safari, and others.

```
MACs-MBP:Hw2 Oscar$ ./a.out
10
Number of Prints: 5
MACs-MBP:Hw2 Oscar$ ./a.out
100
Number of Prints: 200
MACs-MBP:Hw2 Oscar$ ./a.out
1000
Number of Prints: 3500
MACs-MBP:Hw2 Oscar$ ./a.out
5000
Number of Prints: 25000
MACs-MBP:Hw2 Oscar$ ./a.out
100000
Number of Prints: 700000
MACs-MBP:Hw2 Oscar$ █
```

### 3.3. Algoritmo 8

Considera el siguiente algoritmos

```

1 #include <stdio.h>
2
3 int main() {
4     int n;
5     scanf( "%d", &n);
6
7     int NumberOfPrints = 0;
8     int i = n;
9
10    while( i >= 0){
11        for( int j = n; i < j; i -= 2, j /= 2){
12            // printf("Algoritmos\n");
13            NumberOfPrints++;
14        }
15    }
16
17    printf("Number of Prints: %d\n", NumberOfPrints);
18    return 0;
19 }
```

Esta es muy curiosa porque se cicla, ya que al entrar a while  $i = n$ , pero nunca vamos a entrar al for, pues la condición es que  $i < j$ , pero en el momento de la primera comparación  $i = j$ , por lo tanto se va a quedar ahí, dentro del while, para siempre.

## 4. Análisis de Casos de Prints

Para los siguientes algoritmos determine las funciones de complejidad temporal, para el mejor caso, peor caso y caso medio. Indique cual(es) son las condiciones (instancia de entrada) del peor caso y cual(es) la del mejor caso.

## 4.1. Algoritmo 9

Considera el siguiente algoritmos

```

1 func Producto2Mayores(A,n)
2     if (A[1] > A[2])
3         mayor1 = A[1];
4         mayor2 = A[2];
5     else
6         mayor1 = A[2];
7         mayor2 = A[1];
8
9     i = 3;
10
11    while( i <= n )
12        if (A[ i ] > mayor1)
13            mayor2 = mayor1;
14            mayor1 = A[ i ];
15        else if (A[ i ] > mayor2)
16            mayor2 = A[ i ];
17            i = i + 1;
18
19    return mayor1 * mayor2;

```

Usaremos las recomendaciones, es decir usamos como operaciones básicas: comparación entre elementos del arreglo A y Asignaciones a mayor1 y mayor 2.

### 4.1.1. Mejor Caso

Ok, en este caso es sencilla, al ser una modificación de una busqueda lineal es fácil encontrar el mejor caso.

Este se da cuando el elemento mayor de A es A[1] y el siguiente mayor es A[2].

Porque entonces son solo 3 operaciones que se ejecutan antes del while, mientras que el while simplemente se ejecutan  $2(n-2)$ , entonces nota que: el mejor caso es  $f(n) = 2n+1$

#### 4.1.2. Peor Caso

Ok, en este caso es sencilla, al ser una modificación de una búsqueda lineal es fácil encontrar el peor caso.

Este se da cuando el arreglo esté ordenado de menor a mayor, porque en ese caso se hará:

- 3 instrucciones antes del while
- $n-2$  veces:
  - 3 veces al cumplirse el primer if

Por lo tanto tenemos que el peor caso está dado por:  $f(n) = 3n - 3$

#### 4.1.3. Caso Medio

Ok, en este caso es difícil, por un lado tenemos que arriba del while sin importar qué pasa se ejecutan 3 instrucciones, pero dentro del while pasan cosas raras:

- $\frac{1}{3}$  de las veces en promedio se ejecuta el primer if, es decir  $3n - 3$
- $\frac{1}{3}$  de las veces en promedio se ejecuta el segundo if, es decir  $3n - 3$
- $\frac{1}{3}$  de las veces en promedio no se ejecuta nada, es decir  $2n - 1$

Por lo tanto tenemos que el peor caso está dado por:  $f(n) = 3 + \frac{1}{3}(3n - 3) + \frac{1}{3}(3n - 3) + \frac{1}{3}(2n - 1) = \frac{8-7}{3}n$

## 4.2. Algoritmo 10

Considera el siguiente algoritmos

```

1 func OrdenamientoIntercambio(a, n)
2   for (i = 0; i < n - 1; i++)
3     for (int j = i + 1; j < n; j++)
4       if (a[j] < a[i]) {
5         temp = a[i];
6         a[i] = a[j];
7         a[j] = temp;
8       }

```

Usaremos las recomendaciones, es decir usamos como operaciones básicas: comparación entre elementos del arreglo a y Asignaciones a temp y al arreglo a.

### 4.2.1. Mejor Caso

No podemos hacer nada para evitar la ejecución de los dos fors, pero si que podemos evitar la ejecución de todas las instrucciones dentro del if, por lo tanto el mejor caso es simplemente cuando el arreglo esta ordenado de menor a mayor.

Ahora, vemos que la función se puede sacar facilmente como:

- El for interno se ejecuta cada  $n - i + 1$  veces, mas sencillo es  $n - i$  veces
- Ahora vamos a ver cuantas veces se ejecuta el for de afuera, es sencillo pues son  $(n - 1)$  veces

Por lo tanto su suma es sencillamente  $\frac{n^2-n}{2}$

### 4.2.2. Peor Caso

Ok, ahora que vimos el mejor caso, el peor caso es muy sencillo, es cuando el arreglo esta ordenado de mayor a menor y es porque siempre, siempre se van a ejecutar las instrucciones del if, es decir, 4 veces mas que antes por lo tanto es tan sencillo calcularlo como:  $f(n) = 4 \frac{n^2-n}{2} = 2n^2 - 2n$

### 4.2.3. Caso Medio

Ahora esta esta sencilla, pero engañosa, y es que existe la misma posibilidad de que se ejecute el if o no, por lo tanto y SOLO EN ESTE CASO el caso promedio se puede aproximar usando el promedio del mejor y menor, es decir  $f(n) = \frac{5n^2-5n}{2}$

### 4.3. Algoritmo 11

Considera el siguiente algoritmos

```

1 func MaximoComunDivisor(m, n){
2     a = max(n, m);
3     b = min(n, m);
4     residuo = 1;
5     mientras (b > 0){
6         residuo = a mod b;
7         a = b;
8         b = residuo;
9     }
10    MaximoComunDivisor = a;
11    return MaximoComunDivisor;
12 }
```

Usaremos las recomendaciones, es decir usamos como operaciones básicas: operaciones básicas y la operación de modulo de a y b

#### 4.3.1. Mejor Caso

Ok, voy a ser un poco trámoso, pero el mejor caso sería cuando  $n = 0$  en ese caso nunca entramos al while y por lo tanto es cero.

Es bastante obvio que otro caso bastante bueno es cuando  $m = kn$  es decir que uno es múltiplo de otro, pero nada supera a ese caso.

#### 4.3.2. Peor Caso

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  $\phi^k$  entonces tenemos que el peor caso será  $f(n) = \log \phi(n)$

#### 4.3.3. Caso Medio

Si no son el peor caso entonces el cociente nunca podrá ser 1, es decir en cada paso podremos dividir a la mitad el mayor número por lo tanto, es lo inverso a una exponencial es sencillo deducir que es  $f(n) = \log_2(n)$

## 4.4. Algoritmo 12

Considera el siguiente algoritmos

```

1 func BurbujaOptimizada(A, n)
2     cambios = "Si"
3     i = 0
4     Mientras i < n - 1 && cambios != "No" hacer
5         cambios = "No"
6         Para j = 0 hasta (n - 2) - i hacer
7             Si A[j + 1] < A[j] hacer
8                 aux = A[j]
9                 A[j] = A[j + 1]
10                A[j + 1] = aux
11                cambios = "Si"
12            Fin Si
13        Fin Para
14        i = i + 1
15    Fin Mientras
16 Fin func

```

Usaremos las recomendaciones, las comparaciones entre elementos del arreglo A, asignaciones a aux y al arreglo A.

### 4.4.1. Mejor Caso

Ok, esta también esta sencilla, no puedes hacer nada para cambiar el flow del programa mas que evitando entrar en los ifs, así como al while, por lo tanto si nuestro array esta ordenado, en cuyo cas tenemos que  $n - 1$

### 4.4.2. Peor Caso

Es lo inverso, es cuando siempre entramos en el while y siempre en el if, por lo tanto es lo inverso, cuando esta ordenado pero de mayor a menor.

Por lo tanto:

$$\begin{aligned}
 f(n) &= \sum_{i=0}^{n-2} \sum_{i=0}^{n-2-i} 4 \\
 &= 4(n-1)(n-1) - \frac{4(n-1)(n-1)}{2} \\
 &= 2n^2 - 2n
 \end{aligned}$$

#### 4.4.3. Caso Medio

De forma similar tenemos dos bifurcaciones con el if, suponiendo que tenemos la misma probabilidad de entrar es sencillo ver que:  $f(n) = \frac{5n^2 - 5n}{2}$

## 4.5. Algoritmo 13

Considera el siguiente algoritmos

```

1 func BurbujaSimple(A, n)
2     Para i = 0 hasta n - 2 hacer
3         Para j = 0 hasta (n - 2) - i hacer
4             Si A[j + 1] < A[j] hacer
5                 aux = A[j]
6                 A[j] = A[j + 1]
7                 A[j + 1] = aux
8             Fin Si
9         Fin Para
10    Fin Para
11 Fin func

```

Usaremos las recomendaciones, las comparaciones entre elementos del arreglo A, asignaciones a aux y al arreglo A.

### 4.5.1. Mejor Caso

Ok, esta también esta sencilla, no puedes hacer nada para cambiar el flow del programa mas que evitando entrar en los ifs, así como al while. Por lo tanto lo mejor que nos puede pasar es que el arreglo ya este ordenado ascendentemente

Ahora, vemos que la función se puede sacar facilmente como:

- El for interno se ejecuta cada  $n - 1 - i + 1$  veces, mas sencillo es  $n - i$  veces
- Ahora vamos a ver cuantas veces se ejecuta el for de afuera, es sencillo pues son  $(n - 1)$  veces

Por lo tanto su suma es sencillamente  $\frac{n^2-n}{2}$

### 4.5.2. Peor Caso

Es lo inverso, es cuando siempre entramos en el while y siempre en el if, por lo tanto es lo inverso, cuando esta ordenado pero de mayor a menor.

Por lo tanto:

$$\begin{aligned}f(n) &= \sum_{i=0}^{n-2} \sum_{i=0}^{n-2-i} 4 \\&= 4(n-1)(n-1) - \frac{4(n-1)(n-1)}{2} \\&= 2n^2 - 2n\end{aligned}$$

#### 4.5.3. Caso Medio

De forma similar tenemos dos bifurcaciones con el if, suponiendo que tenemos la misma probabilidad de entrar es sencillo ver que:  $f(n) = \frac{5n^2-5n}{2}$

## 4.6. Algoritmo 14

Considera el siguiente algoritmos

```

1 func Ordena(a, b, c)
2     if (a > b)
3         if (a > c)
4             if (b > c)
5                 salida(a, b, c);
6             else
7                 salida(a, c, b);
8         else
9             salida(c, a, b);
10    else
11        if (b > c)
12            if (a > c)
13                salida(b, a, c);
14            else
15                salida(b, c, a);
16        else
17            salida(c, b, a);
18 Fin func

```

Usaremos las recomendaciones, las comparaciones entre a, b, c.

Ok, voy a analizar este problema un poco diferente, porque voy a ir a cada camino posible como no son muchos contando las comparaciones:

```

1 func Ordena(a, b, c)
2     if (a > b)
3         if (a > c)
4             if (b > c)
5                 salida(a, b, c);           // 3 Operaciones
6             else
7                 salida(a, c, b);         // 3 Operaciones
8         else
9             salida(c, a, b);           // 2 Operaciones
10    else
11        if (b > c)
12            if (a > c)
13                salida(b, a, c);       // 3 Operaciones
14            else
15                salida(b, c, a);       // 3 Operaciones
16        else
17            salida(c, b, a);           // 2 Operaciones
18 Fin func

```

#### 4.6.1. Mejor Caso

Ok, este problema esta sencillo de analizar por la falta de ciclos raros y feos, esto ayuda mucho.

Entonces, lo mejor que nos prodria pasar es entrar al primer if, y al segundo ... y al tercero, es decir cuando  $a > b$  y  $a \leq c$ .

O bien, igualmente cuando  $a \leq b$  y  $b \leq c$

Con  $f(n) = 2$

#### 4.6.2. Peor Caso

Es lo inverso, y pasa que como este algoritmo solo puede tardar 2 tiempos diferentes entonces el peor caso es cualquier entrada que no sea el mejor caso.

#### 4.6.3. Caso Medio

Por primera vez creo que hare una función que si sea realmente la correcta, es decir es cada uno de los tiempos por la probabilidad de que ocurra, es decir:  $f(n) = \frac{4}{6}(2) + \frac{2}{6}(3) = \frac{8}{3}$

## 4.7. Algoritmo 15

Considera el siguiente algoritmos

```

1 func Seleccion(A, n)
2     Para k = 0 hasta n - 2 hacer
3         p = k
4         Para i = k + 1 hasta n - 1 hacer
5             Si A[i] < A[p] entonces
6                 p = i
7             Fin Si
8         Fin Para
9         temp = A[p]
10        A[p] = A[k]
11        A[k] = temp
12    Fin Para
13 Fin func

```

Usaremos como operaciones basicas las comparaciones entre elementos del arreglo A, asignaciones a temp y al arreglo A.

### 4.7.1. Mejor Caso

Como ya va siendo costumbre en el mejor caso en los algoritmos de ordenamiento es que este ordenado ascendentemente, porque en ese caso nunca entramos a los if, por lo tanto podemos encontrar la función facilmente gracias a las operaciones que estamos contando:

$$\begin{aligned}
 f(n) &= \sum_{k=0}^{n-2} \left( 3 + \sum_{i=k+1}^{n-1} 1 \right) \\
 &= \frac{2n^2 + 2n - 4 - n^2 + 3n - 2}{2} \\
 &= \sum_{k=0}^{n-2} 3 + n - 1 + k \\
 &= \frac{n^2 + 5n - 6}{2}
 \end{aligned}$$

### 4.7.2. Peor Caso

Es lo inverso, y pasa que como este algoritmo incluso si entra dentro del if y en la vida real si que se tardaría mas como no estamos contando eso como una operación resulta que el valor es igual:  $f(n) = \frac{n^2+5n-6}{2}$

### 4.7.3. Caso Medio

Jajaja, ahora esto esta sencillo, porque si su caso mejor y peor son el mismo adivina cual es su caso medio  $f(n) = \frac{n^2+5n-6}{2}$