

2.5 Análisis de Algoritmos

2.5 Análisis de Algoritmos.

2.5.1 Complejidad en el tiempo.

2.5.2 Complejidad en el Espacio.

2.5.3 Eficiencia de Algoritmos.

2.5 Análisis de Algoritmos

- Para cada problema, es frecuente disponer de más de un algoritmo que lo resuelva en forma correcta.
- El análisis de algoritmos permite decidir, desde el punto de vista de los recursos empleados, cual se debe usar.

2.5 Análisis de Algoritmos

Planteamiento

- Una máquina se encarga de detectar continuamente los sismos causados por las erupciones del volcán Popocatepetl.
 - Durante cierto período de tiempo, la máquina recibirá N datos, y encontrará los valores mínimo y máximo detectados en ese período.
- Para simular lo anterior escribiremos dos programas diferentes que generarán al azar un dato real entre .001 y 10.000 correspondiente a valores de la escala de Richter y obtendrán los valores mínimo y máximo.



2.5 Análisis de Algoritmos

- La complejidad computacional es una medida de los recursos que se invertirán para implementar un algoritmo en una computadora.

2.5 Análisis de Algoritmos

- Los recursos que puede ser de interés controlar son variados, sin embargo, el enfoque general se centra en:
 - Tiempo de Ejecución
 - **Complejidad en el Tiempo.**
 - Espacio requerido para resolver el problema.
 - **Complejidad en el Espacio.**
- El objetivo del análisis de algoritmos es crear programas con los mejores comportamientos en ambos aspectos.

2.5.1 Complejidad en el Tiempo

- El recurso más crítico es el **tiempo de ejecución** de un programa.
- El problema planteado antes, puede resolverse con dos algoritmos distintos por lo menos.

```
// ----- Programa 1 (Dos ciclos anidados) -----
```

```
#include <iostream>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define LIMINF 1
#define LIMSUP 10000
using namespace std;
int main(){
    int const n=15;
    float datos[n];
    srand(time(NULL));
    for(int i=0;i<n;i++){
        datos[i]=(rand()%LIMSUP+LIMINF)/1000.0;
    }
    if (n<=20)
        for(int i=0;i<n;i++){
            cout << datos[i] << endl;
        }
    system("pause");
    // ----- Inicia Ordenamiento -----
    float temp;
    for(int i=0;i<n-1;i++){
        for(int j=i+1;j<n;j++){
            if (datos[i]>datos[j]) {
                temp=datos[i];
                datos[i]=datos[j];
                datos[j]=temp;
            }
        }
    }
    // ----- Fin Ordenamiento -----
    cout << "--datos ordenados--" << endl;
    for(int i=0;i<n;i++){
        cout << datos[i] << endl;
    }
    printf("Menor: %7.3f\n",datos[0]);
    printf("Mayor: %7.3f\n",datos[n-1]);
    return 0;
}
```

```
// ----- Programa 2 (Se usa solo un ciclo para comparar) -----
```

```
#include <iostream>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define LIMINF 1
#define LIMSUP 10000
using namespace std;
int main() {
    int const n=20;
    float datos[n];
    srand(time(NULL));
    for(int i=0;i<n;i++)
        datos[i]=(rand()%LIMSUP+LIMINF)/1000.0;
    if (n<=20)
        for(int i=0;i<n;i++)
            cout << datos[i] << endl;
    system("pause");    // --- Inician Comparaciones ---
    float Menor=10.0;
    float Mayor=0.0;
    for(int i=0;i<n;i++){
        if (datos[i]<Menor)
            Menor=datos[i];
        if (datos[i]>Mayor)
            Mayor=datos[i];
    }
    // --- Imprime Resultados ----
    printf("Menor: %7.3f\n",Menor);
    printf("Mayor: %7.3f\n",Mayor);
    return 0;
}
```

1.5.1 Complejidad en el Tiempo

LA COMPLEJIDAD SE PUEDE OBTENER DE 2 MANERAS:

1) A PRIORI (Forma Teórica).

2) La **teoría** de complejidad computacional se aplica **al algoritmo** para determinar el grado de eficiencia que tendrá el programa.

3) Ventajas:

- No se escribe el programa, hasta en tanto no se hayan analizado diferentes algoritmos y elegido el mejor.
- El análisis es independiente de las condiciones de ejecución, por ejemplo, **no importa si el equipo donde correrá el programa es rápido o lento; el algoritmo es eficiente, o no lo es.**

1.5.1 Complejidad en el Tiempo

2) **A POSTERIORI** (Forma Empírica).

Se escribe el programa, y mediante la experiencia de ejecución, se obtienen conclusiones.

Ventajas:

- Podemos hacer el razonamiento inverso: **¿por qué toma tanto tiempo este programa?**
- El razonamiento inverso es una alternativa para escribir programas más eficientes: encontramos una solución probable, la analizamos, y del análisis puede resultar una mejor aproximación al problema.

1.5.1 Complejidad en el Tiempo

Ejercicio

- **Añada a los dos programas** el código necesario para **medir el tiempo** que se emplea para encontrar los resultados esperados.
- **Uno de los factores** que influyen en la complejidad es la cantidad de datos a procesar, es decir **N**.
- **Registre en una tabla** como la que se muestra a continuación, los resultados que se piden para cada uno de los dos programas.
- **Comparta** algunos de los resultados obtenidos.

1.5.1 Complejidad en el Tiempo

Programa 1
2 ciclos anidados

Tamaño del Vector (n)	Tiempo de ejecución (segundos)
1,000	0
2,000	0.015
5,000	0.1
10,000	0.401
20,000	1.526
50,000	7.746
100,000	26.692
200,000	96.081
500,000	
1,000,000	

Programa 2.
Solo un ciclo

N	Tiempo
1,000	0
2,000	0
5,000	0
10,000	0
20,000	0
50,000	0
100,000	0
200,000	0
500,000	0
1,000,000	0.06

1.5.1 Complejidad en el Tiempo

Otro factor muy importante que influye en la complejidad es la **naturaleza de los datos de entrada**.

Un ejemplo claro es como están dispuestos físicamente. Dependiendo de ello, ciertas instrucciones pueden ejecutarse o no.

Supongamos que, bajo ciertas condiciones, **los datos están ordenados** de **menor a mayor** cuando se obtienen la muestra de los sismos.

Por otro lado, después asumamos que, bajo otras condiciones, los datos están ordenados, pero de **mayor a menor**.

1.5.1 Complejidad en el Tiempo

Siguiendo con el **enfoque empírico**, para determinar las diferencias de eficiencia entre ambos programas, añadiremos a cada uno de ellos un **contador de instrucciones**.

El contador se **incrementará en 1** por cada instrucción simple ejecutada (una asignación, la llamada a una función, etc.).

Las instrucciones dentro de los ciclos y dentro de las funciones (en caso de que las hubiera) también cuentan como 1.

```
// ----- Inicia Ordenamiento -----  
system("cls");  
float temp;  
contador += 2;  
for(int i=0;i<n-1;i++){  
    contador++;  
    for(int j=i+1;j<n;j++){  
        contador++;  
        if (datos[i]>datos[j]) {  
            contador += 3;  
            temp=datos[i];  
            datos[i]=datos[j];  
            datos[j]=temp;  
        };  
    }  
}  
} // ----- Fin Ordenamiento -----
```

1.5.1 Complejidad en el Tiempo

Modificaremos los programas para que los datos se generen ya ordenados y ordenados a la inversa tambien.

En las diapositivas siguientes se muestran los resultados del tiempo y contador de instrucciones en tablas identificadas de la siguiente forma:

- Programa 1. Al azar.
- Programa 1. Ya ordenados.
- Programa 1. Ordenados a la inversa.

- Programa 2. Al azar.
- Programa 2. Ya ordenados.
- Programa 2. Ordenados a la inversa.

Compruebe los resultados ejecutando los programas.

1.5.1 Complejidad en el Tiempo

**Programa 1. Dos ciclos anidados
(datos de entrada ordenados)**

Tamaño del Vector (n)	Número de instrucciones ejecutadas $n(n+1)/2+1$	Tiempo de ejecución (segundos)
1,000	500,501	0.003
2,000	2,001,001	0.013
5,000	12,502,501	0.07
10,000	50,005,001	0.245
20,000	200,010,001	0.991
50,000	1,250,025,001	5.832
100,000	5,000,050,001	23.258
200,000	20,000,100,001	93.126
500,000	125,000,250,001	580.579
1,000,000		

Se generaron solo valores diferentes

Mejor Caso

El tiempo de ejecución y la cantidad de instrucciones ejecutadas son los mínimos.

1.5.1 Complejidad en el Tiempo

**Programa 1. Dos ciclos anidados
(datos de entrada ordenados a la inversa)**

Tamaño del Vector (n)	Número de instrucciones ejecutadas $2n^2 - n + 1$	Tiempo de ejecución (segundos)
1,000	1,999,001	0.015
2,000	7,998,001	0.015
5,000	49,995,001	0.125
10,000	199,990,001	0.453
20,000	799,980,001	1.796
50,000	4,999,950,001	11.481
100,000	19,999,900,001	47.158
200,000	79,999,800,001	187.917
500,000		
1,000,000		

Se generaron solo valores diferentes

Peor Caso

El tiempo de ejecución y la cantidad de instrucciones ejecutadas son los máximos.

1.5.1 Complejidad en el Tiempo

Programa 1. Dos ciclos anidados
(datos de entrada generados al azar)

Tamaño del Vector (n)	Número de instrucciones ejecutadas	Tiempo de ejecución (segundos)
1,000	1,244,912	0.007
2,000	4,754,149	0.017
5,000	28,588,486	0.097
10,000	106,339,877	0.396
20,000	374,835,562	1.506
50,000	1,878,190,543	6.876
100,000	6,424,364,120	25.185
200,000	23,016,485,647	94.625
500,000	132,791,428,150	592.158
1,000,000		

Caso Promedio

El tiempo de ejecución y la cantidad de instrucciones ejecutadas caen en cierto valor intermedio.

1.5.1 Complejidad en el Tiempo

Programa 2. Solo un ciclo
(datos de entrada generados al azar)

N	Contador	Tiempo
1,000	2,020	0
2,000	4,019	0
5,000	10,019	0
10,000	20,022	0
20,000	40,020	0
50,000	100,020	0
100,000	200,020	0
200,000	400,024	0
500,000	1,000,024	0
1,000,000	2,000,017	0.06

Mejor Caso

1.5.1 Complejidad en el Tiempo

Programa 2. Solo un ciclo
(datos de entrada ordenados)

N	Contador	Tiempo
1,000	3,002	0
2,000	6,002	0
5,000	15,002	0
10,000	29,432	0
20,000	50,002	0
50,000	110,002	0
100,000	210,002	0
200,000	410,002	0
500,000	1,010,002	0.06
1,000,000	2,010,002	0.11

Peor Caso

No hay caso promedio, solo hay un mejor caso y dos peores casos

1.5.1 Complejidad en el Tiempo

Programa 2. Solo un ciclo
(datos de entrada ordenados a la inversa)

N	Contador	Tiempo
1,000	3,002	0
2,000	6,002	0
5,000	15,002	0
10,000	29,432	0
20,000	50,003	0
50,000	110,003	0
100,000	210,003	0
200,000	410,003	0
500,000	1,010,003	0.05
1,000,000	2,010,003	0.06

Peor Caso

No hay caso promedio, solo hay un mejor caso y dos peores casos

1.5.1 Complejidad en el Tiempo

La tabla con datos al azar está creada con valores generados entre 1.0 y 10.0 como se planteó en el problema.

Para obtener los límites precisos de comportamiento evitando repeticiones de valores, las tablas con datos ordenados al derecho y a la inversa, se generaron con datos únicos (valores enteros entre 1 y n).

En seguida veremos cómo se pueden expresar los límites del comportamiento de los programas mediante un polinomio. Iniciaremos con el **Programa_1**.

Mejor Caso

(menor cantidad de instrucciones ejecutadas)

Antes del primer ciclo hay **2** instrucciones.
El primer ciclo se ejecuta **$n-1$** veces.

El segundo ciclo se ejecutará múltiples veces, dependiendo del primer ciclo ($n-1, n-2, n-3$, etc hasta 1, es decir, la sumatoria de los números entre 1 y $n-1$):
 $(n-1)(n-1+1)/2$.

Como el vector está ordenado, las instrucciones para el intercambio nunca se ejecutan.

$$2 + n - 1 + (n-1)(n-1+1)/2 \rightarrow n(n+1)/2 + 1$$

$0.5n^2 + 0.5n + 1$

```
system("cls");  
float temp;  
contador += 2;  
for(int i=0; i<n-1; i++){  
    contador++;  
    for(int j=i+1; j<n; j++){  
        contador++;  
        if (datos[i]>datos[j]) {  
            contador += 3;  
            temp=datos[i];  
            datos[i]=datos[j];  
            datos[j]=temp;  
        };  
    }  
}
```

Recordemos que, siendo **x** un número natural, la sumatoria de los números entre **1** y **x** es **$x(x+1)/2$**

1.5.1 Complejidad en el Tiempo

Peor Caso

(mayor cantidad de instrucciones ejecutadas)

La cantidad de instrucciones ejecutadas es la correspondiente al mejor caso (diapositiva anterior), más las instrucciones del intercambio:

$$0.5n^2 + 0.5n + 1 + 3(n-1)(n-1+1)/2$$

$$0.5n^2 + 0.5n + 1 + 3n(n-1)/2$$

$$0.5n^2 + 0.5n + 1 + 1.5n^2 - 1.5n$$

$$2n^2 - n + 1$$

(El polinomio de la diapositiva anterior y este representan los límites del comportamiento del programa 1, siempre y cuando para cualquier tamaño de N, los datos sean todos diferentes)

1.5.1 Complejidad en el Tiempo

Para el caso del **Programa 2**, el polinomio para el peor caso es el único que podemos determinar empíricamente ya que el mejor caso se presenta con datos al azar.

Peor Caso: $3n + 2$

En las tablas se comprobó cual es el mejor caso, caso promedio y peor caso para cada uno de los dos programas.

1.5.1 Complejidad en el Tiempo

A este tipo de problemas, se les llama problemas P (Polinómicos). No todos los miembros de este conjunto de programas son muy eficientes.

- Los lineales son los más eficientes.
- Los problemas de segundo grado o mayor, se encuentran en los **límites de lo tratable**. Bajo ciertas condiciones se pueden resolver en un tiempo razonable, bajo otras condiciones no.
- ¿Por qué?
 - Porque un **programa útil** se usará para problemas cada vez **más grandes**.
 - **La magnitud** del trabajo que realiza un programa **depende generalmente** de una variable que comúnmente llamamos **N**.

1.5.1 Complejidad en el Tiempo

Por lo tanto, se debe pensar en que un programa debe ser eficiente para resolver problemas de **magnitud creciente**. Es decir, cuando **N tiende a infinito**, o sea, su **comportamiento asintótico**.

Al grupo de algoritmos que comparten el mismo comportamiento asintótico se dice que pertenecen al mismo **orden de complejidad**.

El orden de complejidad se denomina **notación O**.

1.5.1 Complejidad en el Tiempo

No es necesario conocer el comportamiento exacto de un algoritmo, solo su límite de comportamiento: la **cota o acotamiento superior**.

En el análisis asintótico lo importante es el comportamiento cuando **N tiende a infinito** por lo que el grado del polinomio es lo que determina el orden de complejidad.

$$2n^2 - n + 1$$

Los términos lineal e independiente **se hacen despreciables cuando N tiende a infinito**. Por otro lado, podemos ignorar la constante multiplicativa del término de mayor grado, porque $2 * \infty = \infty$.

1.5.1 Complejidad en el Tiempo

Por tanto el algoritmo con el polinomio anterior y el programa equivalente son de orden **$O(n^2)$** .

El polinomio de primer grado correspondiente al programa 2, peor caso:

$3n + 2$, en consecuencia, será de orden **$O(n)$** .

1.5.1 Complejidad en el Tiempo

Para obtener **A PRIORI** la complejidad de un algoritmo, y por lo tanto del programa equivalente contamos con las siguientes **reglas**:

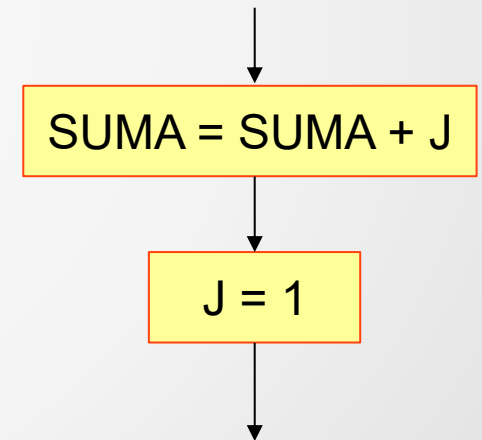
a) Instrucciones Sencillas:

Instrucciones de declaración, entrada y salida o asignación que **no dependan** del tamaño **N** del problema.

Todas son de orden **$O(1)$** , es decir, son de orden constante, y siempre se ejecutará la misma cantidad de ellas.

Si hay varias de ellas en secuencia, todas se toman como una:

$$O(1) + O(1) = O(1).$$



1.5.1 Complejidad en el Tiempo

b) Ciclos:

Debemos considerar 2 casos:

Primero.

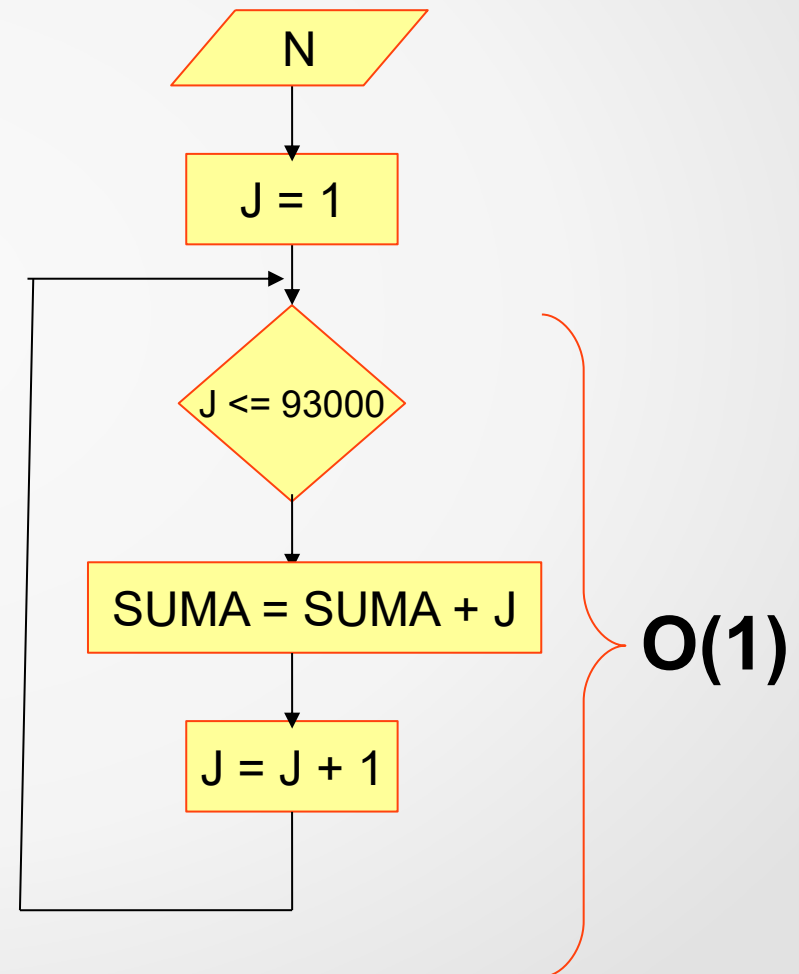
si el ciclo se ejecuta un cierto número de veces, pero independiente del tamaño **n** del problema, en tal caso solo se involucra a una constante.

Ejemplos

93000 es despreciable cuando "n" tiende a infinito

$$93000 * O(1) \rightarrow ?$$

$$K * O(1) \rightarrow ?$$



1.5.1 Complejidad en el Tiempo

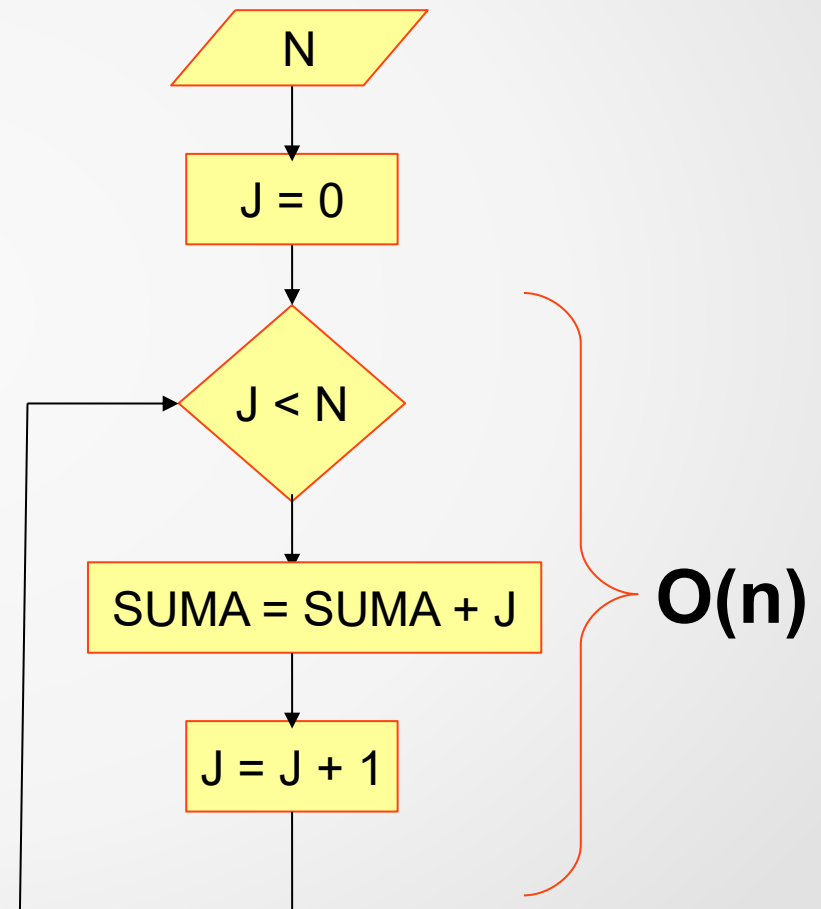
b) Ciclos (continuación) :

Si el tamaño **N** del problema, aparece como límite del ciclo, hay varias situaciones.

Un ciclo, con una secuencia de instrucciones simples en su interior y **el contador del ciclo avanza uno a uno**.

Ejemplo

$N * O(1) \rightarrow ?$



1.5.1 Complejidad en el Tiempo

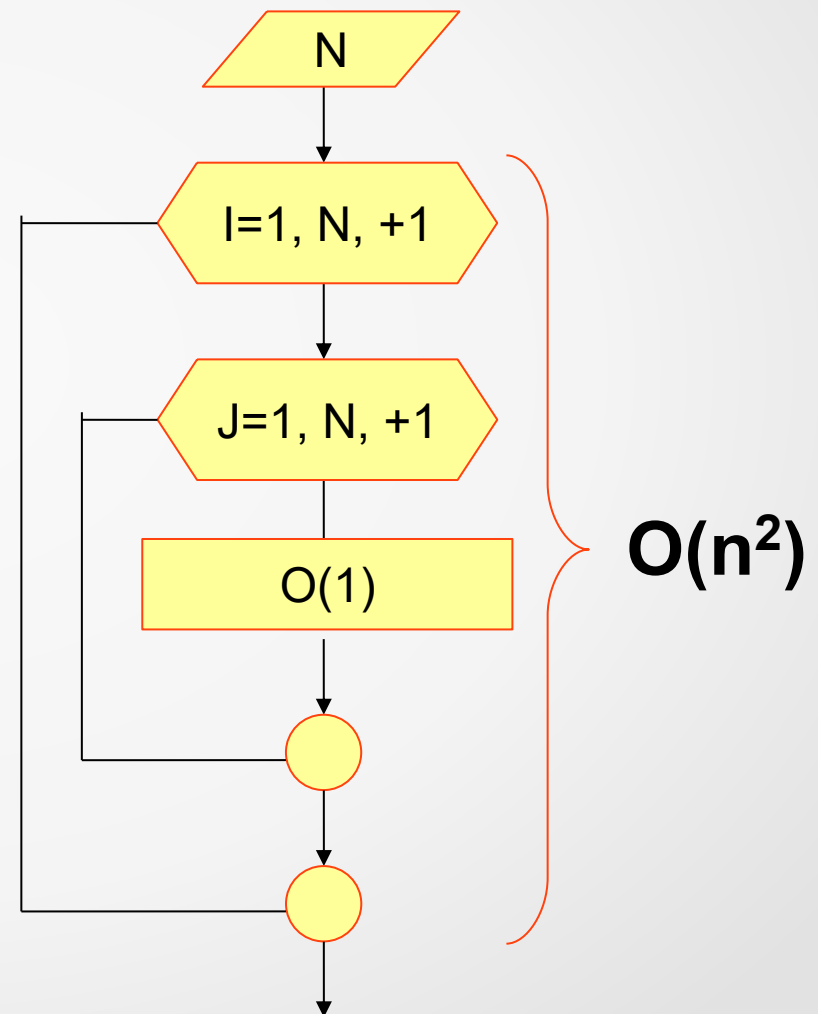
b) Ciclos (continuación) :

Si el tamaño **N** del problema, aparece como límite de **2 ciclos anidados** en que ambos índices se suman de 1 en 1 y en el interior hay una serie de instrucciones simples.

Ejemplo

$$N * N * O(1) \rightarrow ?$$

$$N * O(n) \rightarrow ?$$



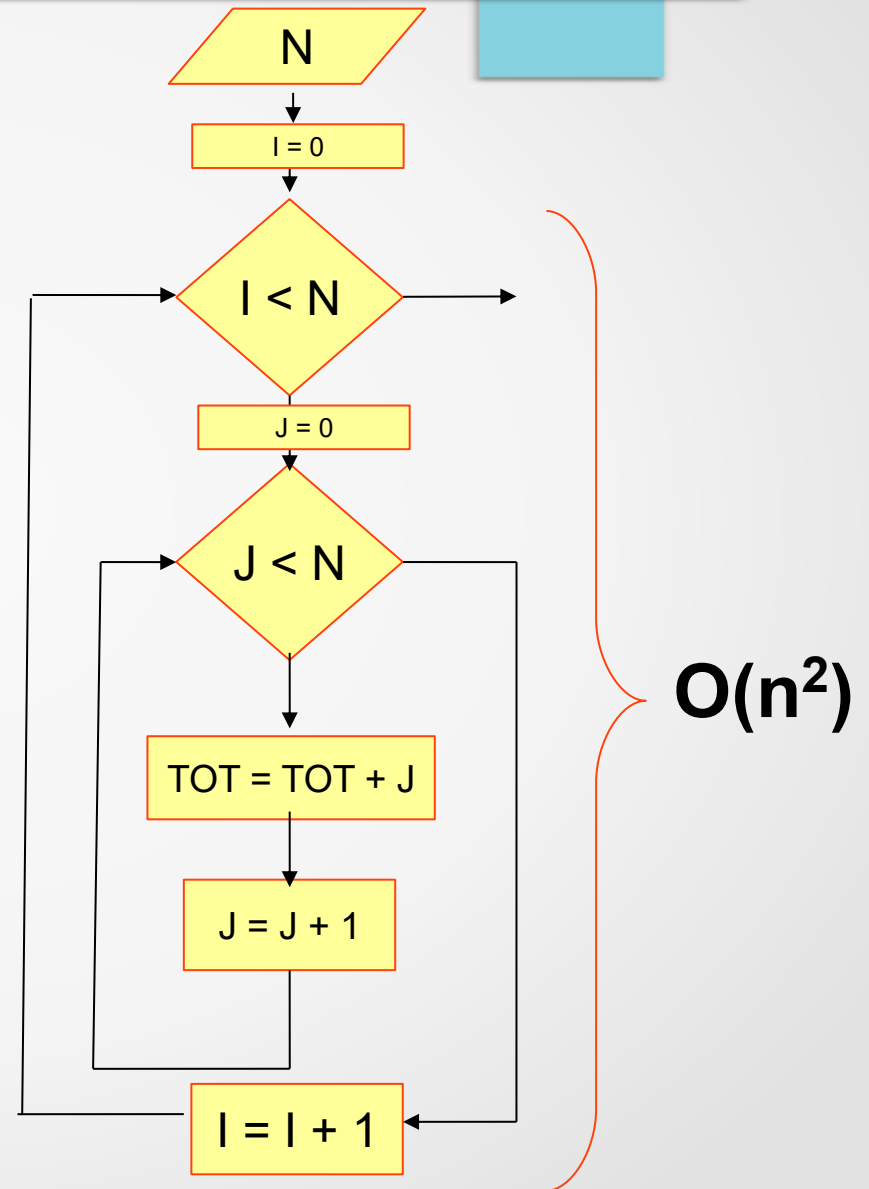
1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación) :

Esta es otra forma de ver los mismos ciclos anidados del ejemplo anterior.

Ejemplo

$N * N * O(1) \rightarrow ?$



1.5.1 Complejidad en el Tiempo

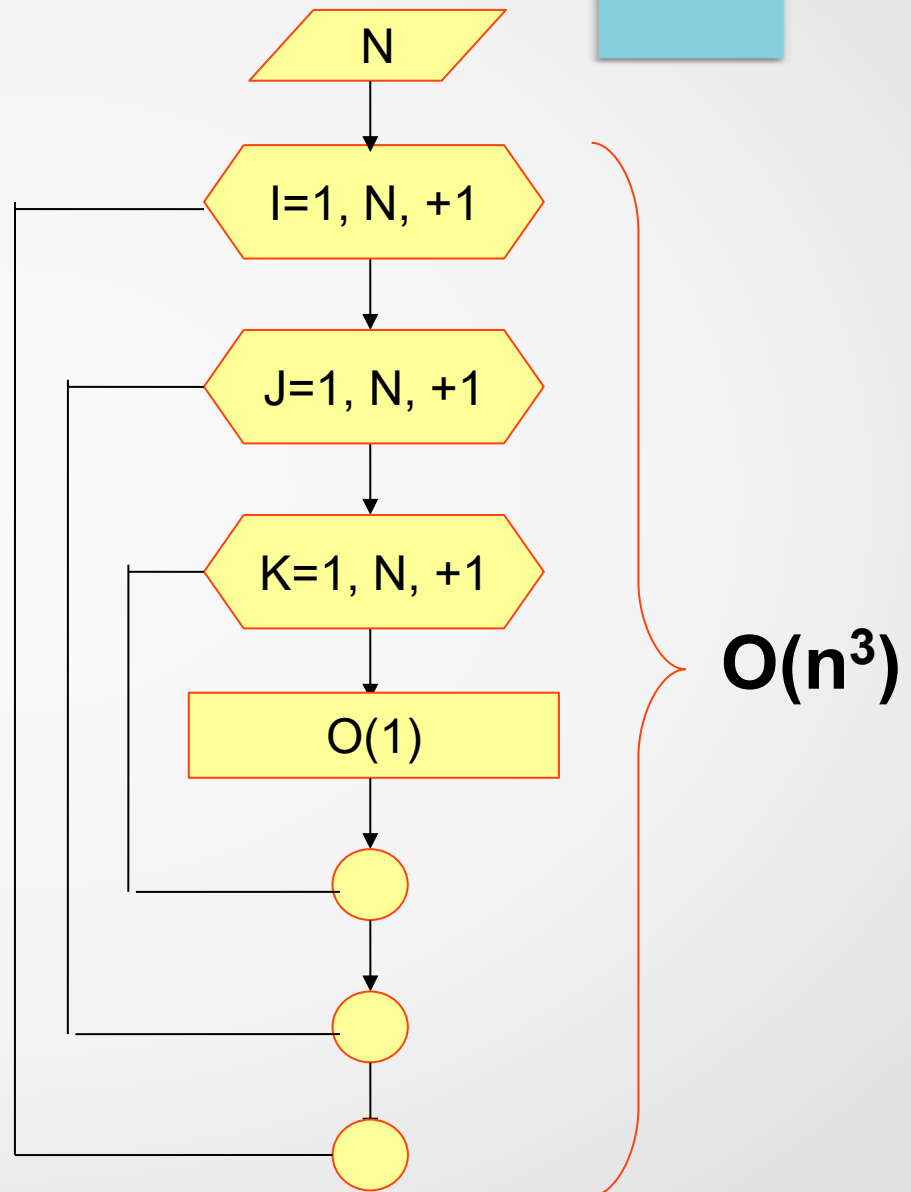
b) Ciclos (continuación):

Tres ciclos anidados con los índices incrementándose de 1 en 1.

Ejemplo

$$N * N * N * O(1) \rightarrow ?$$

$$N * O(n^2) \rightarrow ?$$



1.5.1 Complejidad en el Tiempo

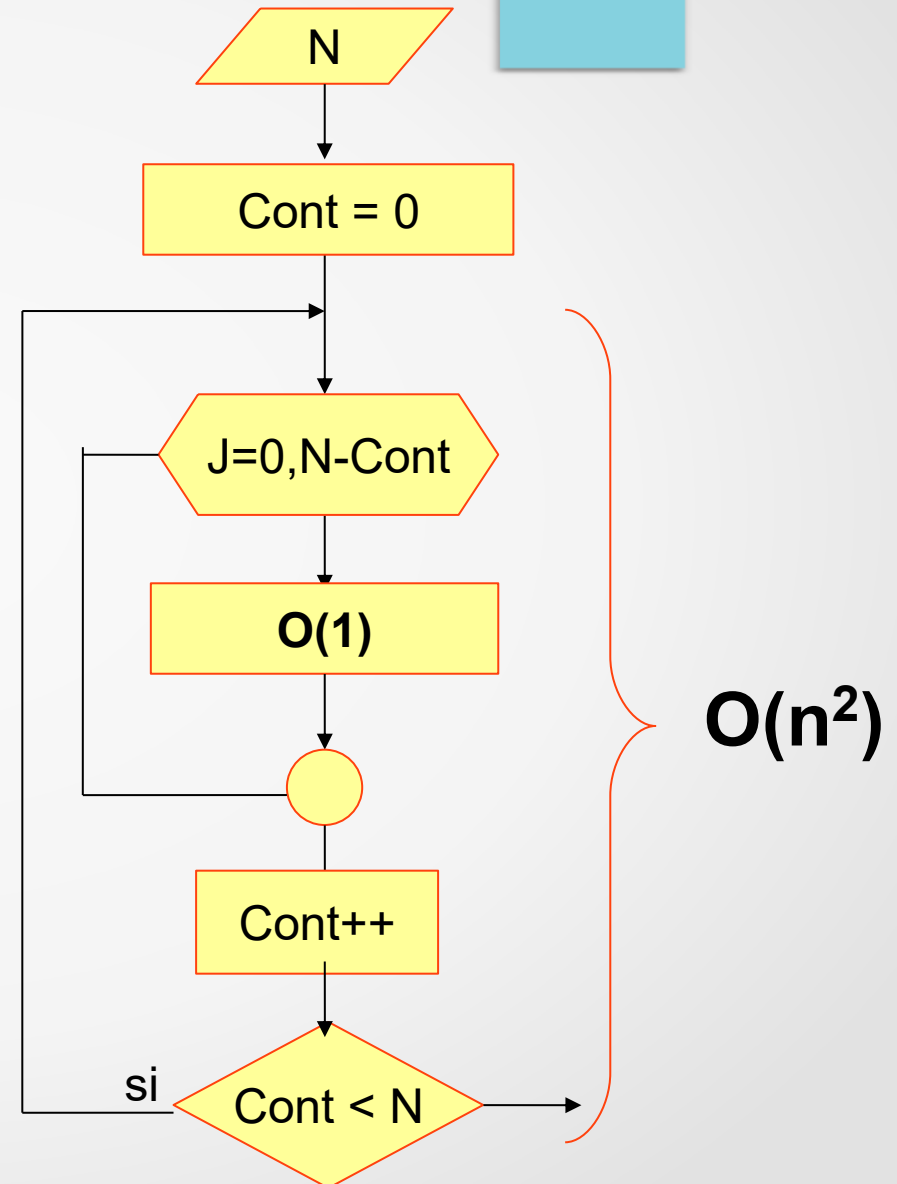
b) Ciclos (continuación) :

Un ciclo se realiza **N veces**.

Este contiene otro ciclo en su interior que se ejecuta N veces la primera, N-1 veces la segunda, N-2 veces la tercera y así sucesivamente, hasta **1** vez.

Dentro del segundo ciclo, hay una secuencia simple de instrucciones.

$$n*(n+1)/2 * O(1) \rightarrow ?$$



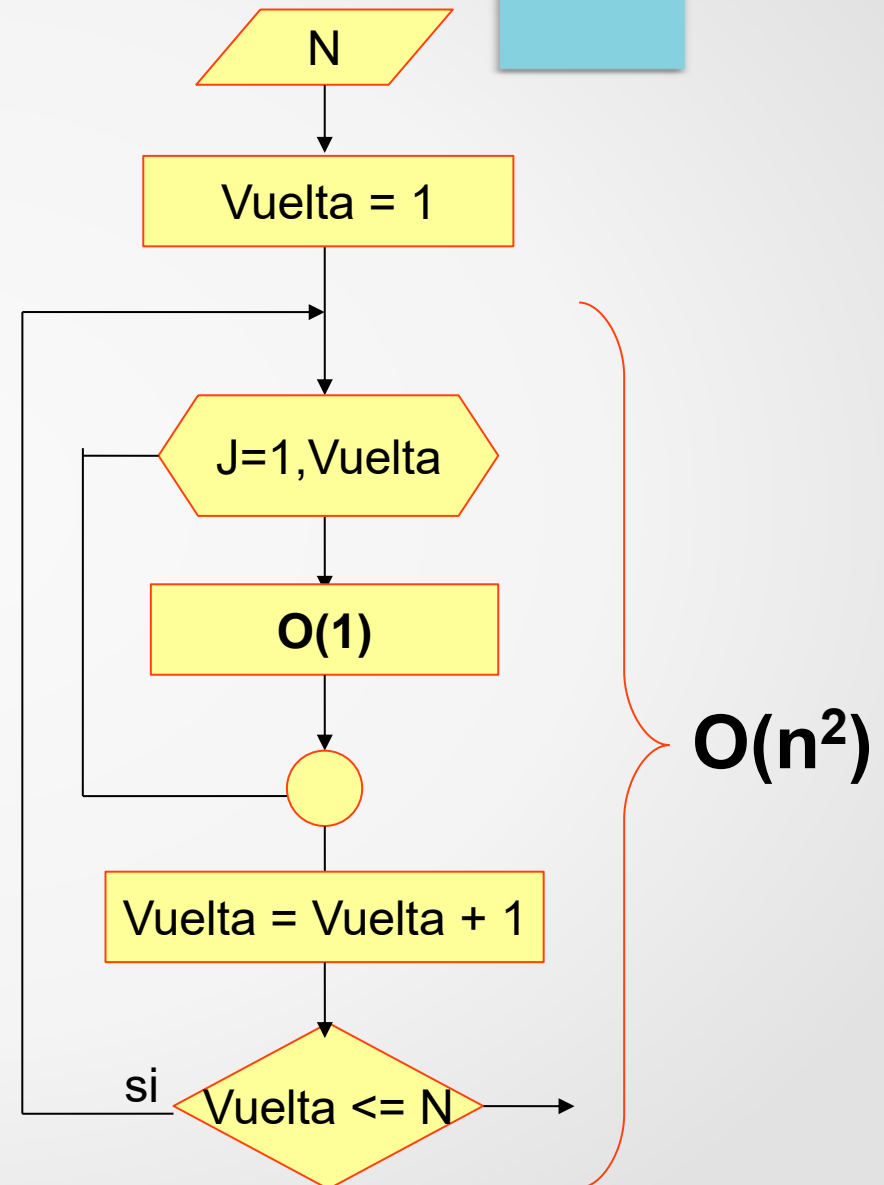
1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación) :

Un ciclo se realiza **N veces**.

Este contiene otro ciclo en su interior que se ejecuta 1 vez la primera, 2 veces la segunda, 3 veces la tercera y así sucesivamente, hasta **N** veces. Dentro del segundo ciclo, hay una secuencia simple de instrucciones.

$$n*(n+1)/2 * O(1) \rightarrow ?$$

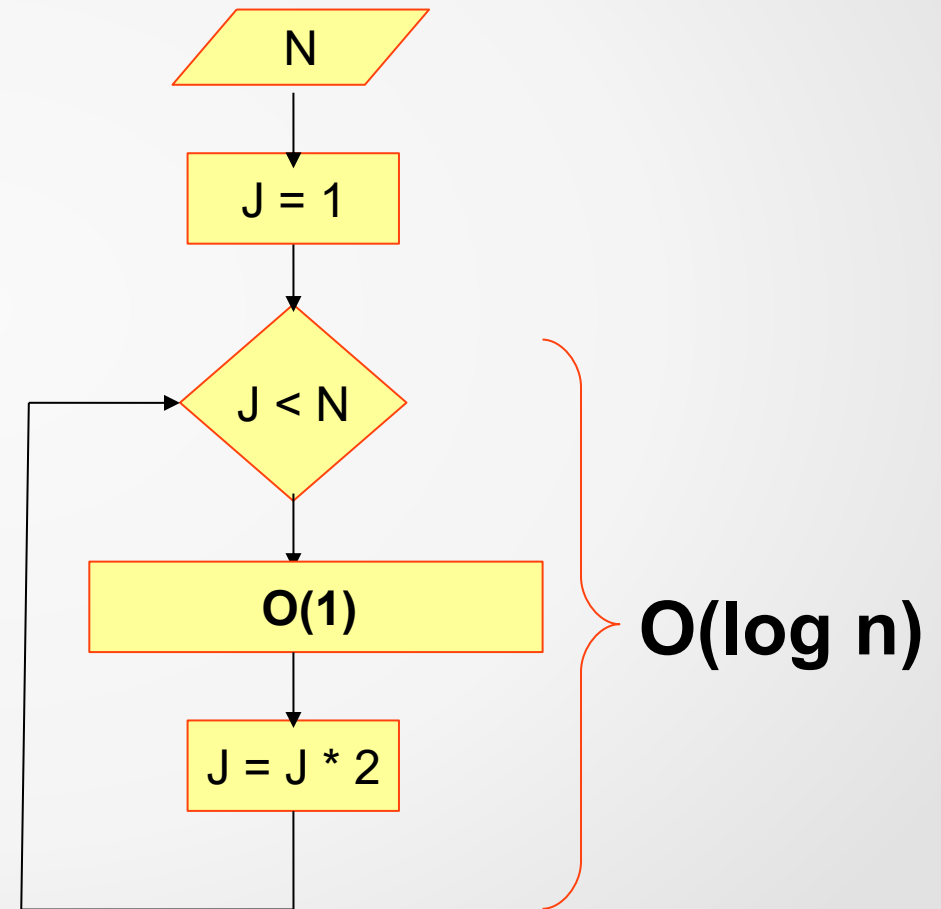


1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación) :

Ciclos en los que el contador tiene un valor inicial de 1 y debe llegar a N pero multiplicándose por una constante y en su interior hay una secuencia simple de instrucciones:

$$\log_2(N) * O(1) \rightarrow ?$$

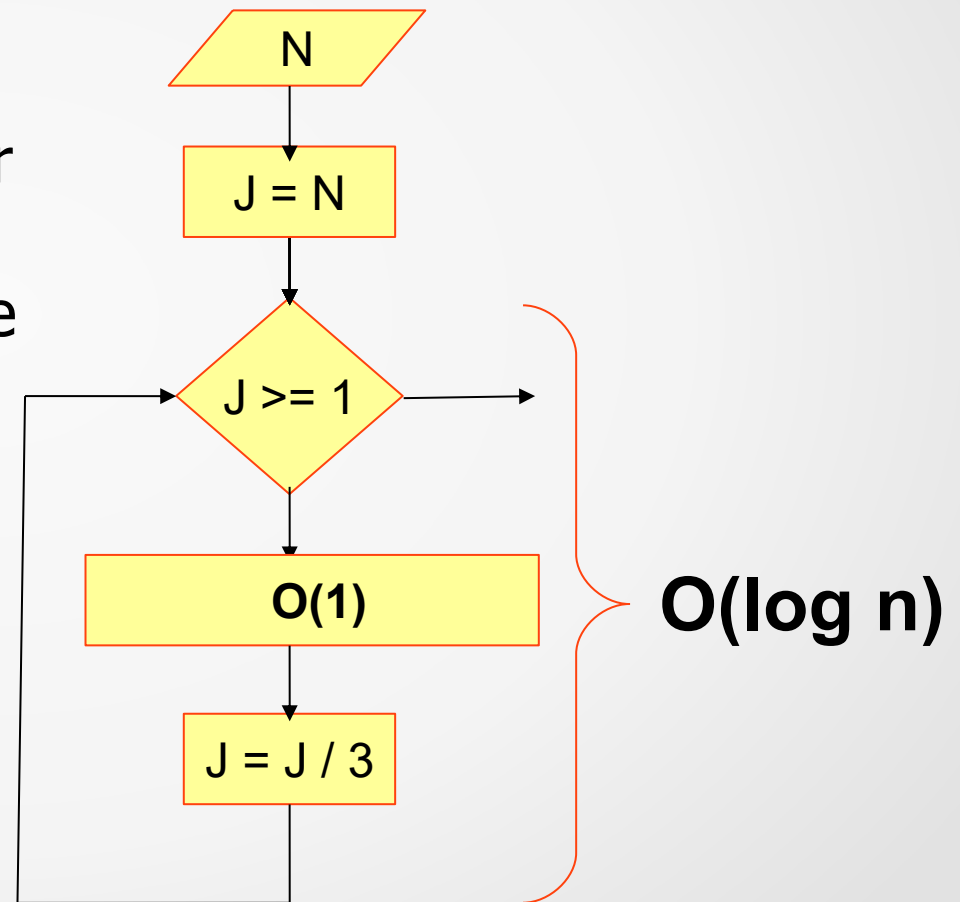


1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación) :

Ciclos en los que el contador tiene un valor inicial de N y debe llegar a 1 pero dividiéndose sucesivamente por una constante y en su interior hay una secuencia simple de instrucciones:

$$\log_3(N) * O(1) \rightarrow ?$$



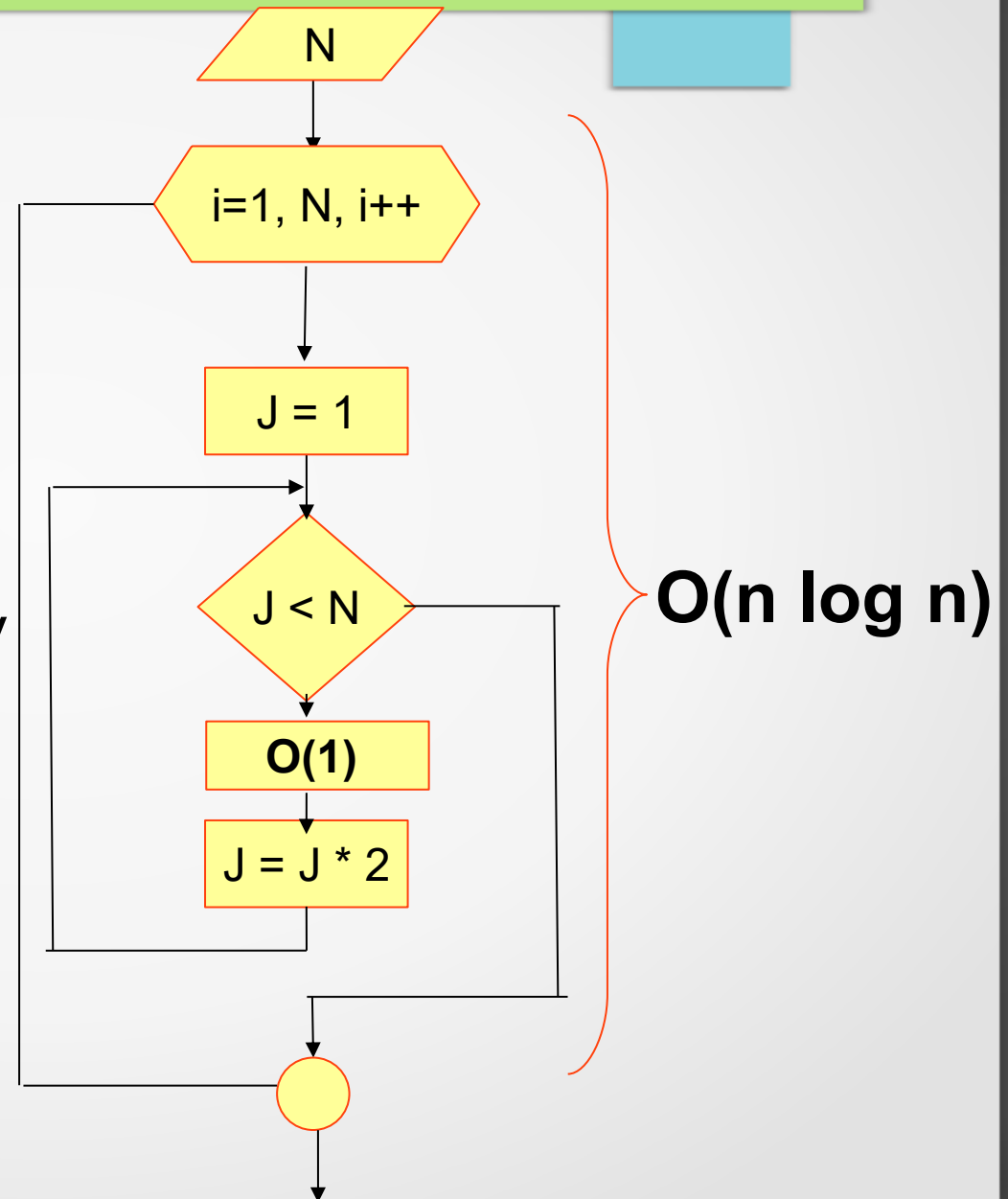
7.1 Complejidad en el Tiempo

Ciclos (continuación) :

Un ciclo se realiza **N veces**.
Este contiene otro ciclo en su interior que se ejecuta **$\log n$** veces.

Dentro del ciclo más interior, hay una secuencia simple de instrucciones.

$$n * \log n \rightarrow ?$$



1.5.1 Complejidad en el Tiempo

c) Jerarquía

Los órdenes de complejidad, se agrupan jerárquicamente ya que los ordenes menores están todos acotados por los mayores.

$O(1)$	Constante
$O(\log n)$	Logarítmico
$O(n)$	Lineal
$O(n \log n)$	Cuasi Lineal
$O(n^2)$	Cuadrático
$O(n^3)$	Cúbico
$O(n^k) \ k > 3$	Polinómico
$O(k^n) \ k > 1$	Exponencial
$O(n!)$	Factorial

1.5.1 Complejidad en el Tiempo

d) Secuencias

En las sumas del polinomio correspondiente a un algoritmo, un orden superior, “absorbe” al inferior y la suma de ordenes de la misma jerarquía equivalen al mismo orden.

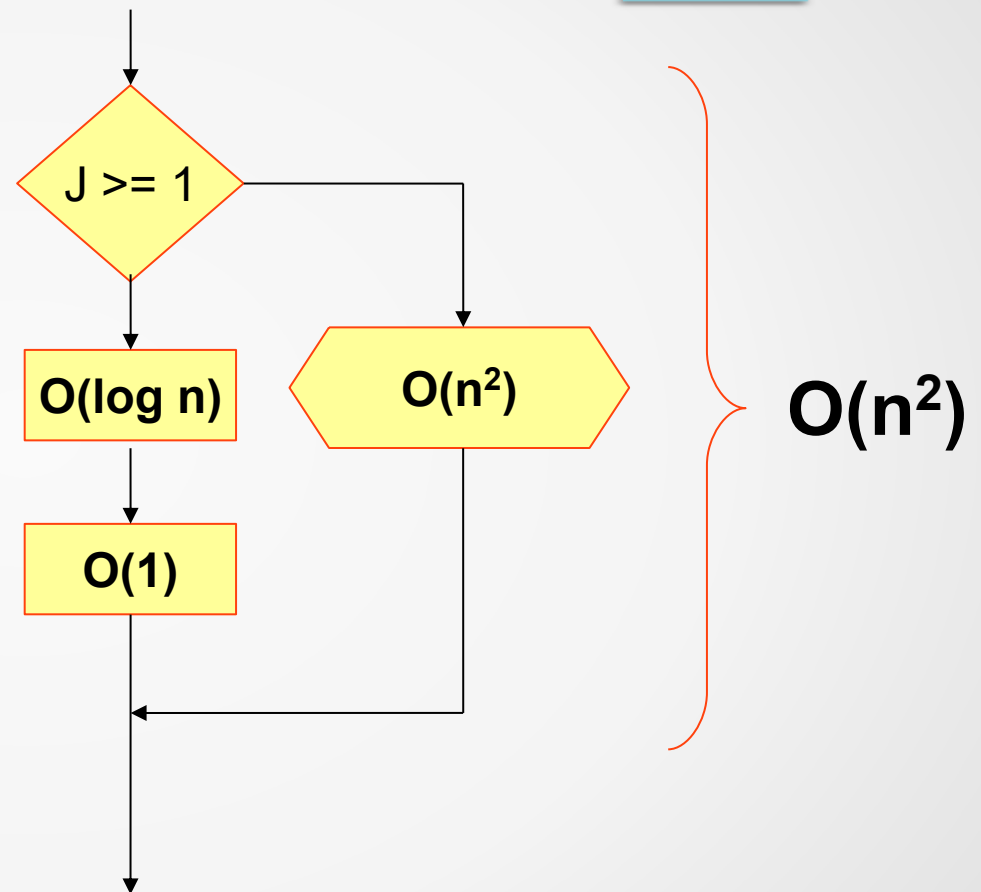
Ejemplos

$O(1) + O(n^2) + O(n)$	$\rightarrow O(n^2)$
$O(1) + O(1) + O(1)$	$\rightarrow O(1)$
$O(n) + O(n) + O(n)$	$\rightarrow O(n)$
$O(n) + O(n \log n) + O(1)$	$\rightarrow O(n \log n)$

1.5.1 Complejidad en el Tiempo

e) Decisiones (if)

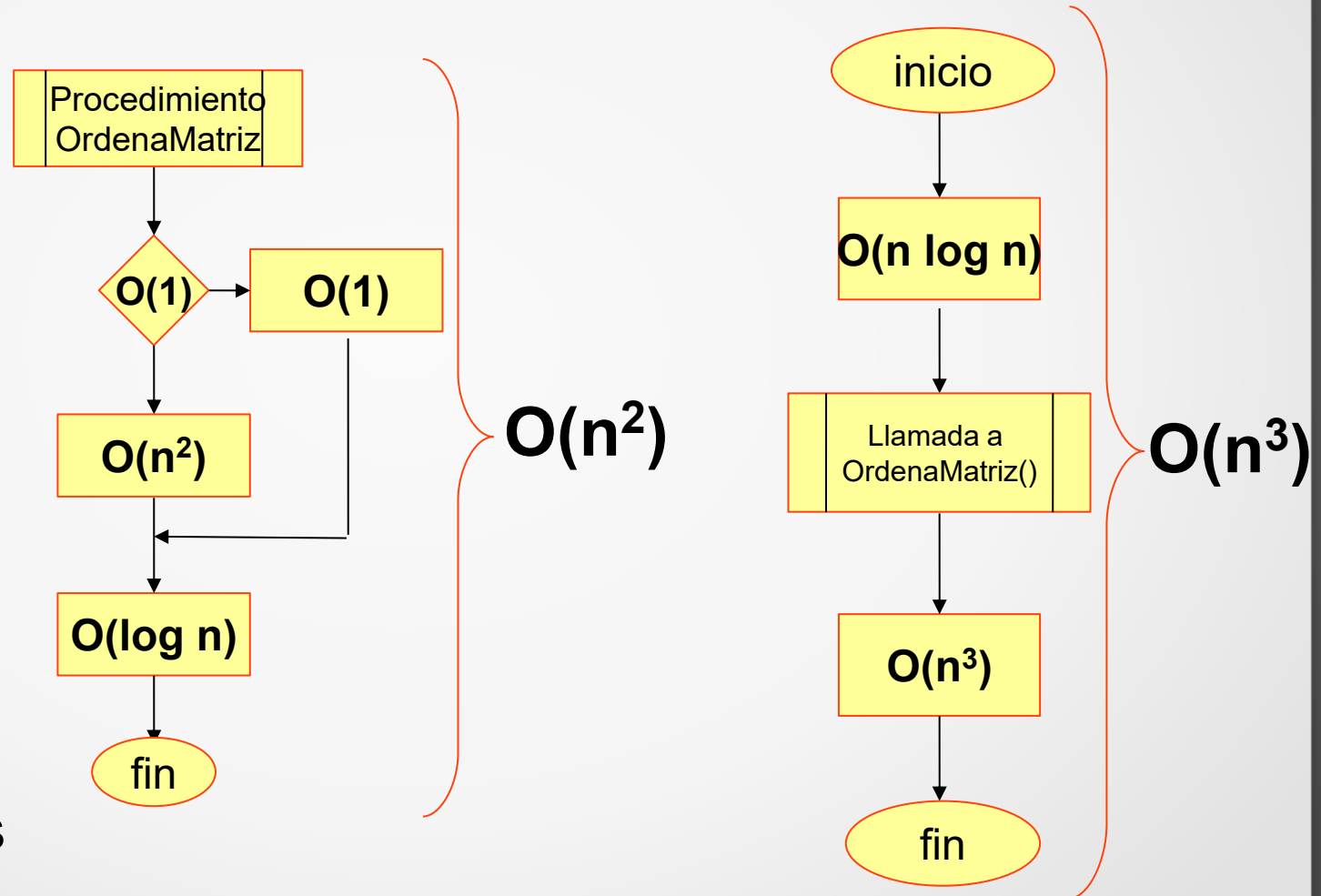
La evaluación de la condición generalmente es de orden **$O(1)$** , complejidad que **se sumará** con la **peor de las 2 ramas del if**, sea del **then** o el **else**, siguiendo las reglas de las secuencias.



1.5.1 Complejidad en el Tiempo

f) Procedimientos

La complejidad de **la llamada** a ejecución de un procedimiento es **$O(1)$** , pero la complejidad del procedimiento en sí, se calcula de acuerdo a las instrucciones contenidas en él, basándonos en las reglas que hemos aprendido.



Orden de Complejidad de Recursividad

Orden de complejidad de procedimientos recursivos

Ejemplo **fibo()**

```
int fibo(int posicion)
{
if ((posicion==1) or (posicion==2) )
    return posicion-1;
else
    return fibo(posicion-1)+fibo(posicion-2) ;
}
```

El orden de complejidad temporal de esta función en particular es **$O(1)$** , que se debe multiplicar por el número de veces que se ejecuta recurrentemente: **2^n** aproximadamente, como lo vimos en el árbol de ejecución (ver Tema Recursividad). Por lo tanto la complejidad de Fibonacci Recursivo es

$$O(1) * 2^n \rightarrow O(k^n)$$

Orden de Complejidad de Recursividad

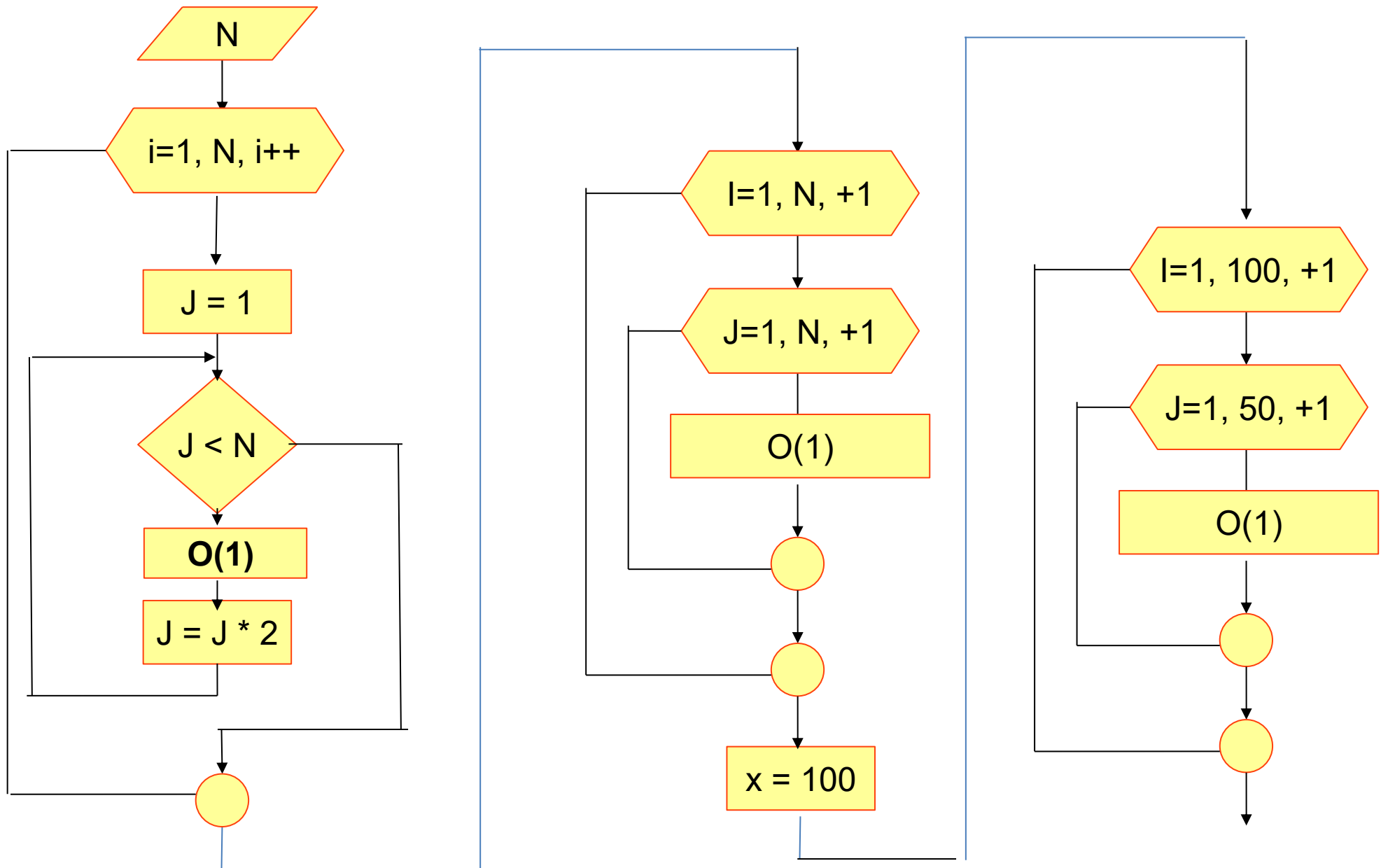
Ejemplo **factorial()**

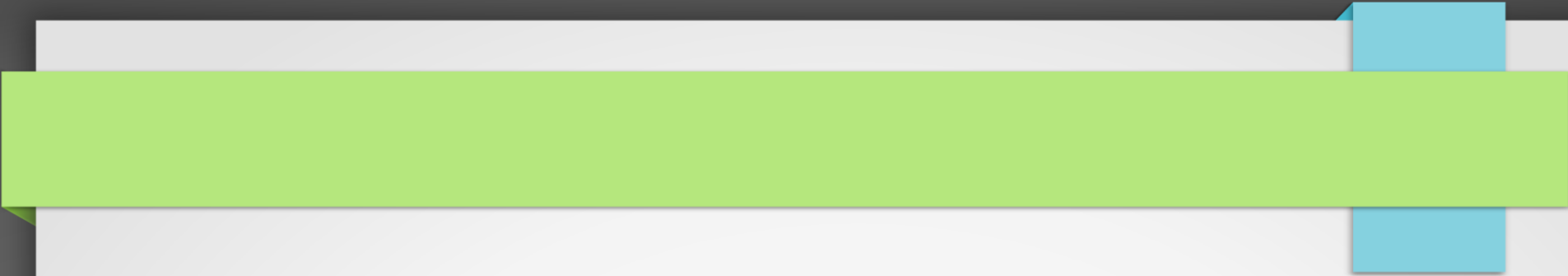
```
double factorial(char x)
{
    if (x==0)
        return 1;
    else
        return x*factorial(x-1);
}
```

El orden de complejidad temporal de la función es **O(1)** porque todas las instrucciones son sencillas, que se debe multiplicar por el número de veces que se ejecuta recurrentemente, en este caso, **n** veces. Por lo tanto, la complejidad de Factorial Recursivo es

$$\mathbf{O(1) * N \rightarrow O(n)}$$

EJERCICIO: Escriba la secuencia que representa este diagrama de flujo y determine el Orden de Complejidad temporal





1.5.2 Complejidad en el Espacio

1.5.2 Complejidad en el espacio

Se refiere al **espacio de memoria** proporcional a N que un algoritmo requerirá durante su ejecución.

Cuando nos referimos al espacio de memoria puede ser RAM, Discos Duros o Unidades de Estado Sólido (SSD *Solid State Drive*).

El monto de la memoria empleada para resolver el problema determina el grado de complejidad espacial.

1.5.2 Complejidad en el espacio

El grado de complejidad espacial se refiere al espacio proporcional al tamaño del problema requerido para guardar los datos del propio problema (**no se refiere al espacio auxiliar requerido adicionalmente al espacio de los datos**).

1.5.2 Complejidad en el espacio

Problema planteado anteriormente

N personas participan en un juego de lanzamiento de dados en el cual se asigna un **número de 1 al N** a cada una de ellas.

Cada jugador lanzará un **par de dados** tantas veces como indique el número que le haya tocado (el jugador **# N** lanzará **N veces**, el jugador **# $N-1$** lanzará **$N-1$ veces** hasta terminar con el **jugador # 1 que lanzará solo una vez**).

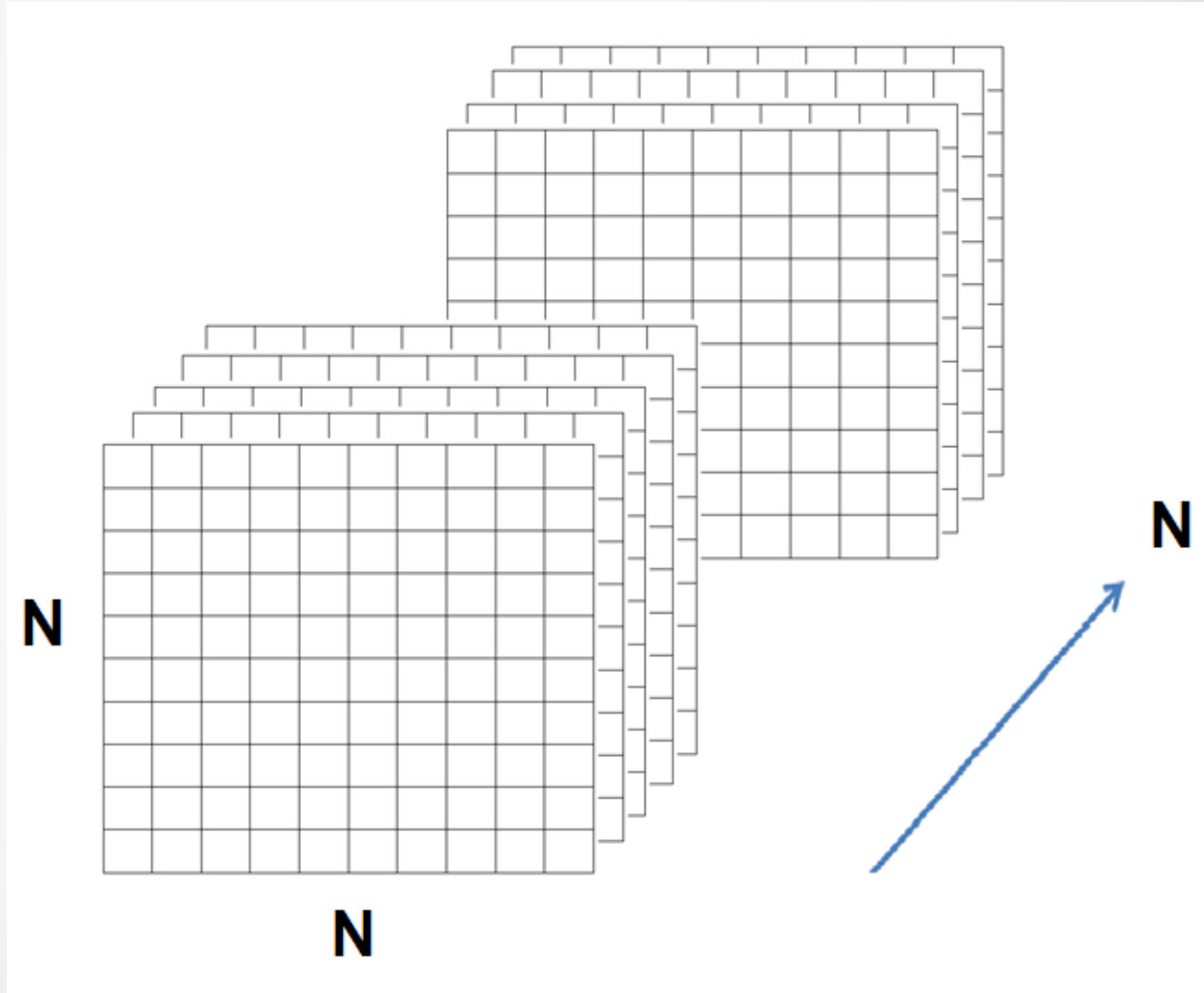
Los resultados de los lanzamientos deberán conservarse.

1.5.2 Complejidad en el espacio

Supongamos que el planteamiento del problema estableciera que se requiere hacer N simulaciones del juego, es decir, tantas simulaciones como número de jugadores.

En tal caso, el espacio requerido para conservar los resultados de todos los juegos es exactamente N^3 .
para $N=1000$, mil millones de lugares.

El orden de complejidad espacial para este problema es $O(n^3)$.



Aun así, si fueran 1000 jugadores los participantes, el espacio requerido sería medio millón de lugares.

Si guardamos los datos en un vector para ahorrar espacio, aún así el espacio requerido será **$N*(N+1)/2$** .

Y el orden de complejidad espacial es el mismo: **$O(n^2)$** .

[illegible]

1.5.2 Complejidad en el espacio

El orden de complejidad espacial **no depende del algoritmo que elijamos sino de la naturaleza del problema**, por lo que, a diferencia de la complejidad temporal no podemos hacer nada para disminuir el orden de complejidad espacial.

Por eso el problema del lanzamiento de dados planteado con las diferentes condiciones tiene un orden de **Complejidad Espacial de $O(n^2)$ u $O(n^3)$** .

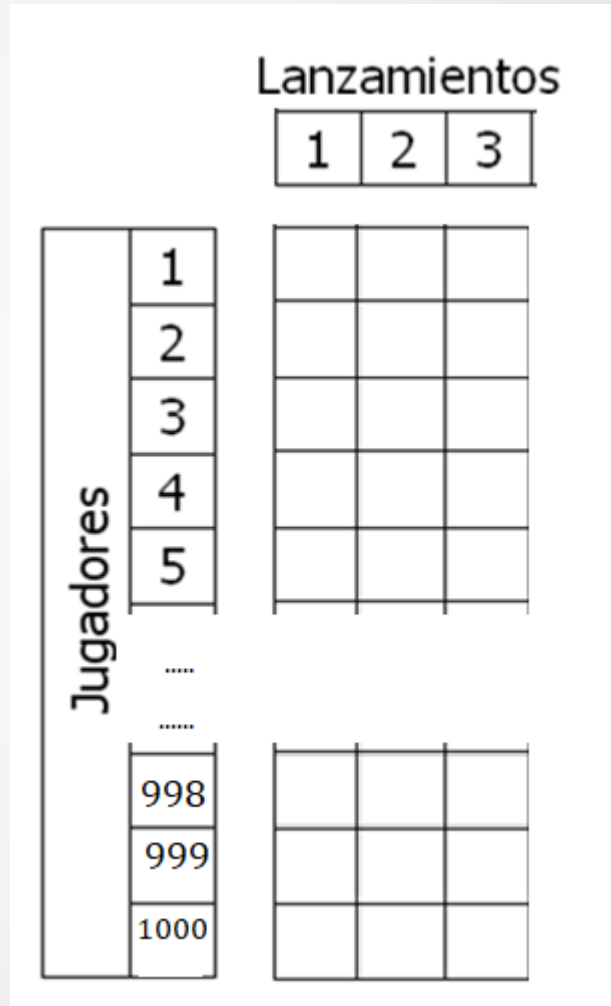
1.5.2 Complejidad en el espacio

Sin embargo, si el planteamiento del juego dice que el número de lanzamientos será el entero mas cercano al logaritmo base 10 de el número total de jugadores, el orden de complejidad espacial será otro.

Para $N=1000$, cada uno de los jugadores haría 3 lanzamientos (es decir, 3,000 lugares en la memoria).

Si $N=1,000,000$, cada uno haría 6 lanzamientos y se requerirían solo 6,000,000 de lugares.

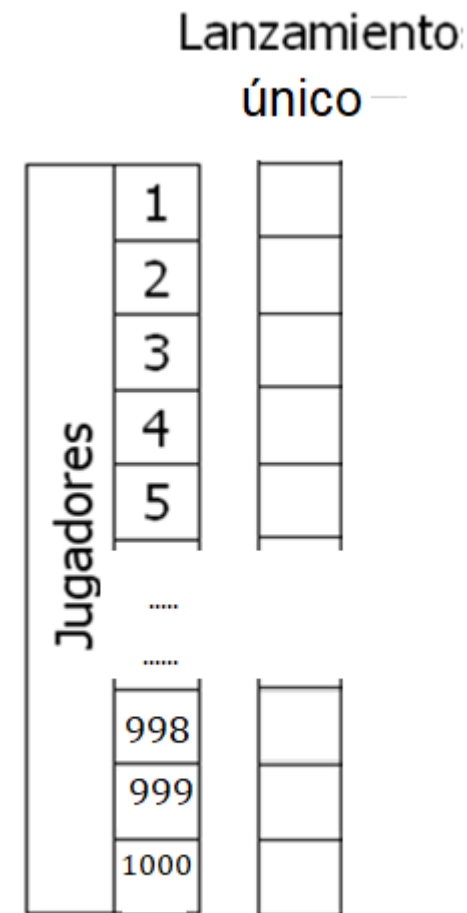
El problema planteado bajo estas condiciones tiene un orden de Complejidad Espacial de **$O(n \log n)$** .



1.5.2 Complejidad en el espacio

Ahora supongamos que cada jugador solo hará un lanzamiento sin importar el número total de jugadores o alguna otra condición.

En tal caso el espacio requerido siempre será proporcional a N , por lo que bajo estas condiciones, el orden de Complejidad Espacial sería **$O(n)$** .



1.5.2 Complejidad en el espacio

De acuerdo a lo anterior, tiene impacto negativo para la complejidad espacial:

- Solo la cantidad de los **datos** producidos por el propio problema y que **dependen de N**.

1.5.2 Complejidad en el espacio

No Tiene impacto negativo para la Complejidad Espacial:

- La estructura lógica del **algoritmo**. Es decir, lo que afecta para la complejidad temporal, no tiene impacto para la espacial. Ciclos anidados por ejemplo.
- Las **estructuras de datos** (arreglos, *array lists*, etc.) cuyo tamaño no depende de **N**.
 - Por ejemplo el tamaño del diccionario que se usa para revisar la ortografía de un texto.
- Los **parámetros** (no importa si son de valor o referencia), ya que es espacio extra auxiliar para resolver el problema.
- La cantidad de **datos individuales**, por ejemplo, las variables declaradas.

1.5.3 Eficiencia de los algoritmos

Los mejores algoritmos son aquellos en los que la Complejidad Temporal es inferior a **$O(n^2)$** .

Respecto a la Complejidad Espacial, al no depender del algoritmo, solo nos queda obtener el orden de complejidad como una medida a observar para prepararse con los recursos necesarios para resolver el problema.

- Decidir si usar almacenamiento secundario (SSD o disco duro) en vez de RAM por ejemplo, ya que el espacio disponible es mayor en el primer caso.

1.5.3 Eficiencia de los algoritmos

Así como para complejidad temporal, los problemas con Orden de Complejidad Espacial mayores a $O(n \log n)$ requieren mucho espacio en memoria por lo que debemos estar preparados para poder resolver esos problemas.

$O(1)$	Constante	
$O(\log n)$	Logarítmico	
$O(n)$	Lineal	
$O(n \log n)$	Cuasi Lineal	
$O(n^2)$	Cuadrático	⊘
$O(n^3)$	Cúbico	⊘
$O(n^k) \ k > 3$	Polinómico	⊘
$O(k^n) \ k > 1$	Exponencial	⊘
$O(n!)$	Factorial	⊘