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 va a usar

2.5 Análisis de Algoritmos

Escribiremos dos diferentes programas que resuelvan el siguiente problema:

- Una máquina se encarga de detectar continuamente los sismos causados por las erupciones del volcán Popocatépetl.
- Usted debe diseñar dos diferentes programas que generen al azar un dato real entre .001 y 10.000 correspondiente a valores de la escala de Richter, indicativo de un sismo detectado.
- Durante cierto período de tiempo se recibirán N datos, y cada programa encontrará los valores mínimo y máximo alcanzados.

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 (**Complejidad en el Espacio**).
- El objetivo del análisis de algoritmos es crear programas con la menor complejidad espacial y temporal.

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.

.5.1 Complejidad en el Tiempo

```
// --------- Programa 1 ------------
#include <iostream>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#define LIMINF 1
#define LIMSUP 10000
using namespace std;
```

1.5.1 Complejidad en el Tiempo

```
int main()
{
    int const n=100;
    float datos[n];
    srand(time(NULL));
    for(int i=0;i<n;i++)
        datos[i]=
       (rand()%LIMSUP+LIMINF)/1000.0;
    for(int i=0;i<n;i++)
        cout << datos[i] << endl;
```

1.5.1 Complejidad en el Tiempo

```
// ------ Inicia Ordenamiento ----
system("cls");
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 -----
```

1.5.1 Complejidad en el Tiempo

```
    system("pause");
    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;
}
```

1.5.1 Complejidad en el Tiempo

```
// ------------ Programa 2 ------------
#include <iostream>
#define LIMINF 1
#define LIMSUP 10000
using namespace std;
int main()
{
   int const n=100;
   float datos[n];
   srand(time(NULL));
   for(int i=0;i<n;i++)
      datos[i]=(rand()%LIMSUP+LIMINF)/1000.0;
```

1.5.1 Complejidad en el Tiempo

```
// --- Inician Comparaciones ---
   float Menor=LIMSUP;
   float Mayor=LIMINF;
   for(int i=0;i<n;i++){
      if (datos[i]<Menor)
          Menor=datos[i];
      if (datos[i]>Mayor)
          Mayor=datos[i];
   }
```

1.5.1 Complejidad en el Tiempo

```
    // --- Imprime Resultados ----
    printf("Menor: %7.3f\n",Menor);
    printf("Mayor: %7.3f\n",Mayor);
    system("pause");
    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

Práctica:

- 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 (tamaño de la entrada), en este caso N.
- Registre, en una tabla como la que se muestra a continuación, los resultados que se piden para cada uno de los dos programas.

1.5.1 Complejidad en el Tiempo

Programa 1	
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	

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

Modifique los programas para que los datos se generen de la forma que hemos hablado, además de al azar, obtenga nuevas tablas e identifíquelas de la forma siguiente:

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.

1.5.1 Complejidad en el Tiempo

Siguiendo con el **enfoque empírico**, para determinar las diferencias de eficiencia entre ambos programas, añada 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.

Registre en las 6 tablas (como la que se muestra enseguida) los resultados que se piden.

1.5.1 Complejidad en el Tiempo

Programa 1		
Tamaño del Vector (n)	Número de instrucciones ejecutadas	Tiempo de ejecución (segundos)
1,000		
2,000		
5,000		
10,000		
20,000		
50,000		
100,000		
200,000		
500,000		
1,000,000		

1.5.1 Complejidad en el Tiempo

- Caso Mejor

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

- Caso Peor

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

- Caso Promedio

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

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 como se pueden expresar los límites del comportamiento de los programas mediante un polinomio. Iniciaremos con el **Programa_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

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 no se ejecutan ni una vez.
2+n-1+(n-1)(n-1+1)/2 → n(n+1)/2+1
$0.5n^2+0.5n+1$

Recordemos que, siendo **x** un numero 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: **3 n + 2**

1.5.1 Complejidad en el Tiempo

Programa 1 (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	16,339,877	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

1.5.1 Complejidad en el Tiempo

Se generaron solo valores diferentes

Programa 1 (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		

Mejor Caso

1.5.1 Complejidad en el Tiempo

Se generaron solo valores diferentes

Programa 1 (orden inverso)		
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		

Peor Caso

1.5.1 Complejidad en el Tiempo

PROGRAMA 2
DATOS ALEATORIOS

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
DATOS 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

1.5.1 Complejidad en el Tiempo

PROGRAMA 2
DATOS 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

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:

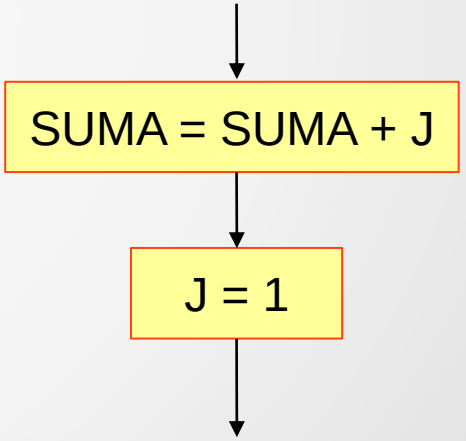
3n + 2, en consecuencia será de orden **O(n)**.

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).

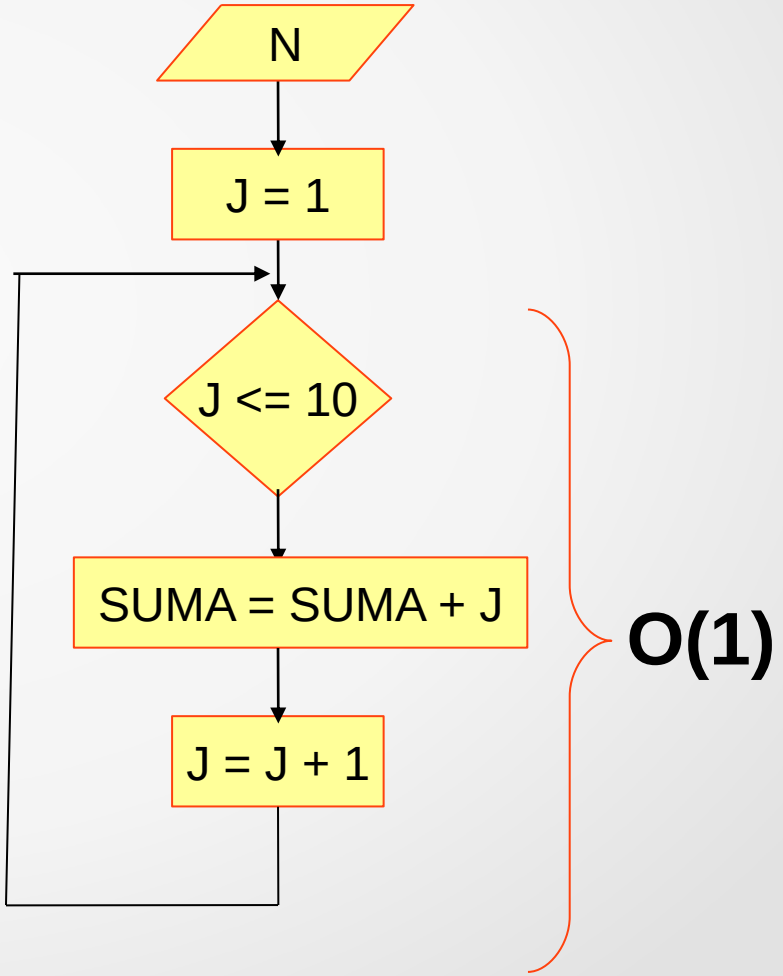
1.5.1 Complejidad en el Tiempo

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

10 * O(1) → ?

K * O(1) → ?

1.5.1 Complejidad en el Tiempo

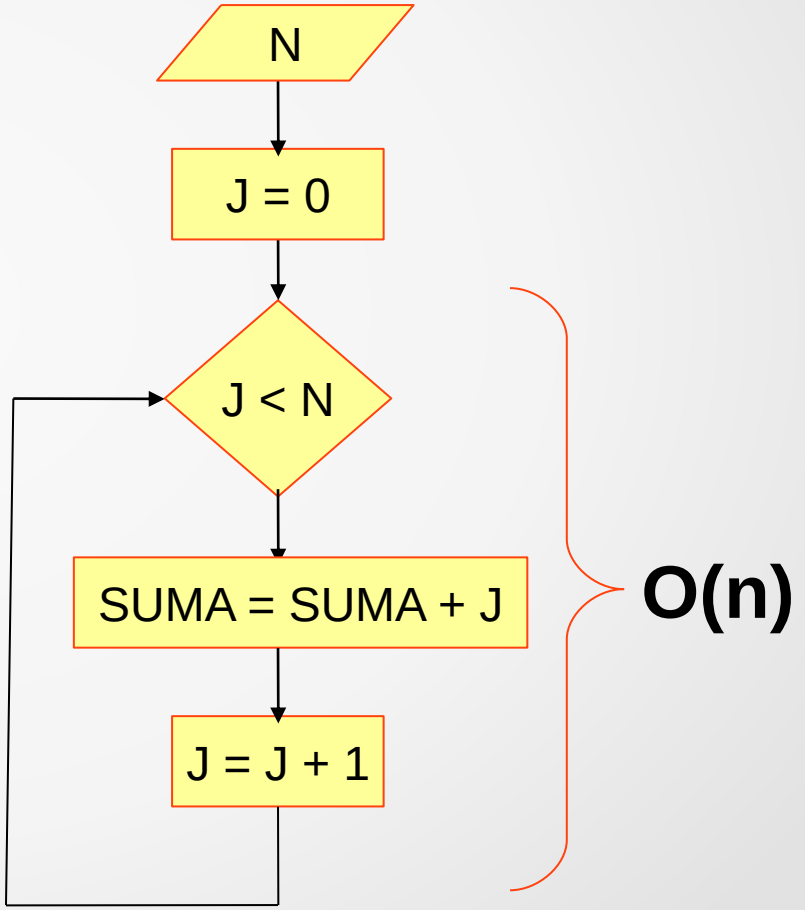
b) Ciclos (continuación) :

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) → ?



1.5.1 Complejidad en el Tiempo

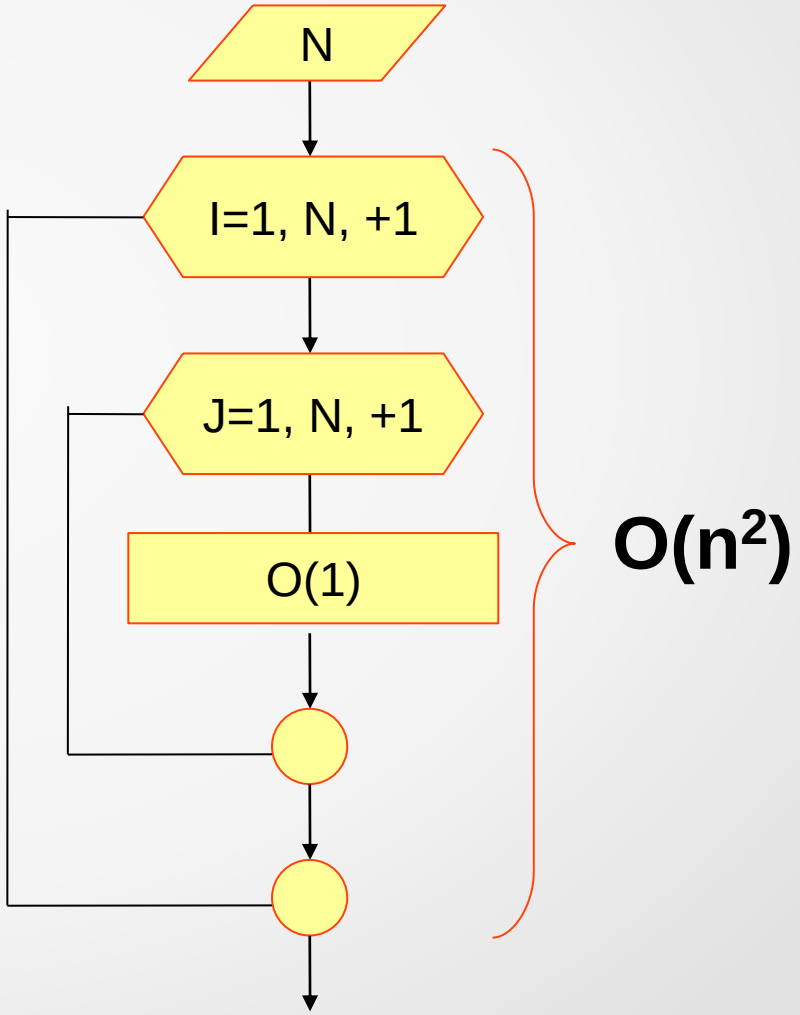
b) Ciclos (continuación) :

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) → ?

N * O(n) → ?



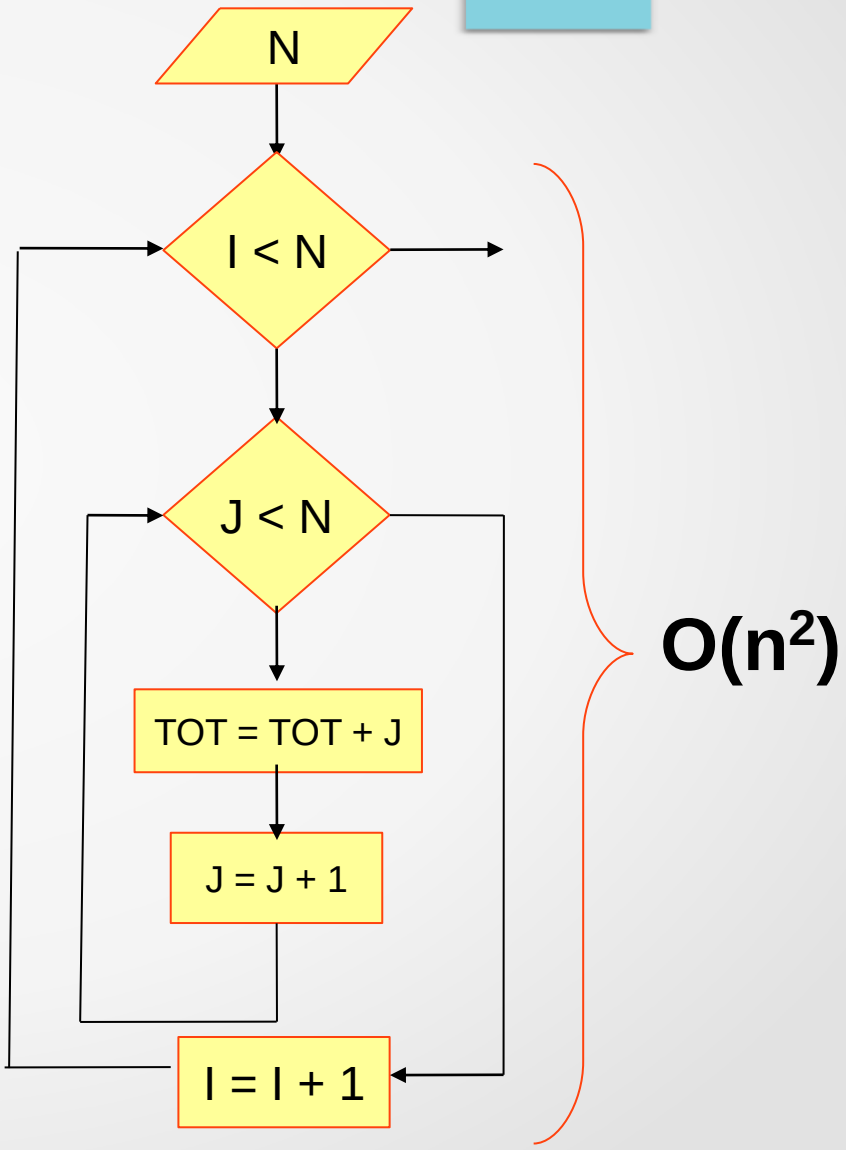
1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación) :

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

Ejemplo

N * N * O(1) → ?



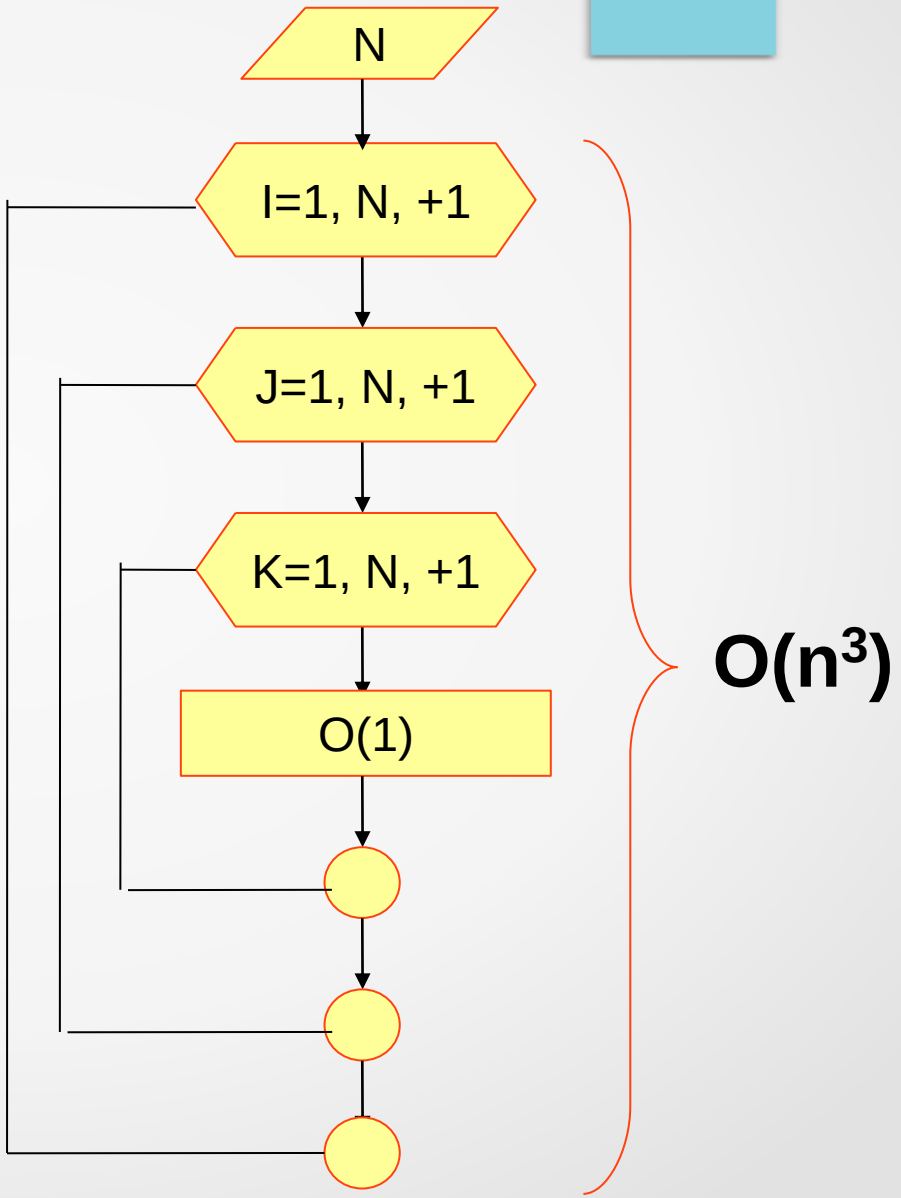
1.5.1 Complejidad en el Tiempo

b) Ciclos (continuación):

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

Ejemplo

N * N * N * O(1) → ?
N * O(n^2) → ?

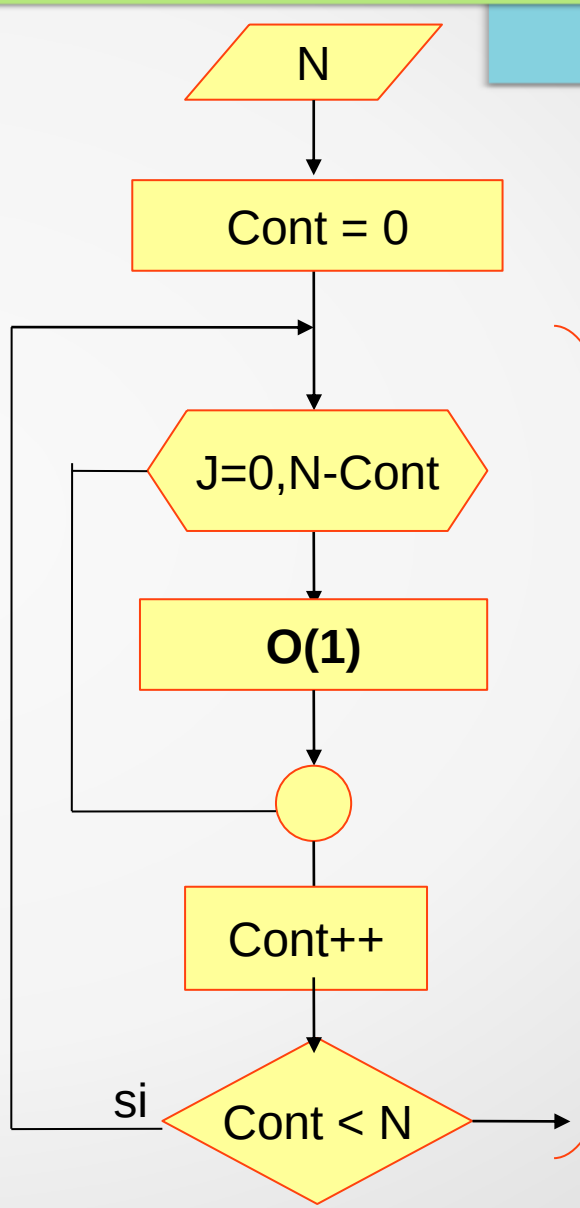


1.5.1 Complejidad en el Tiempo

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) → ?



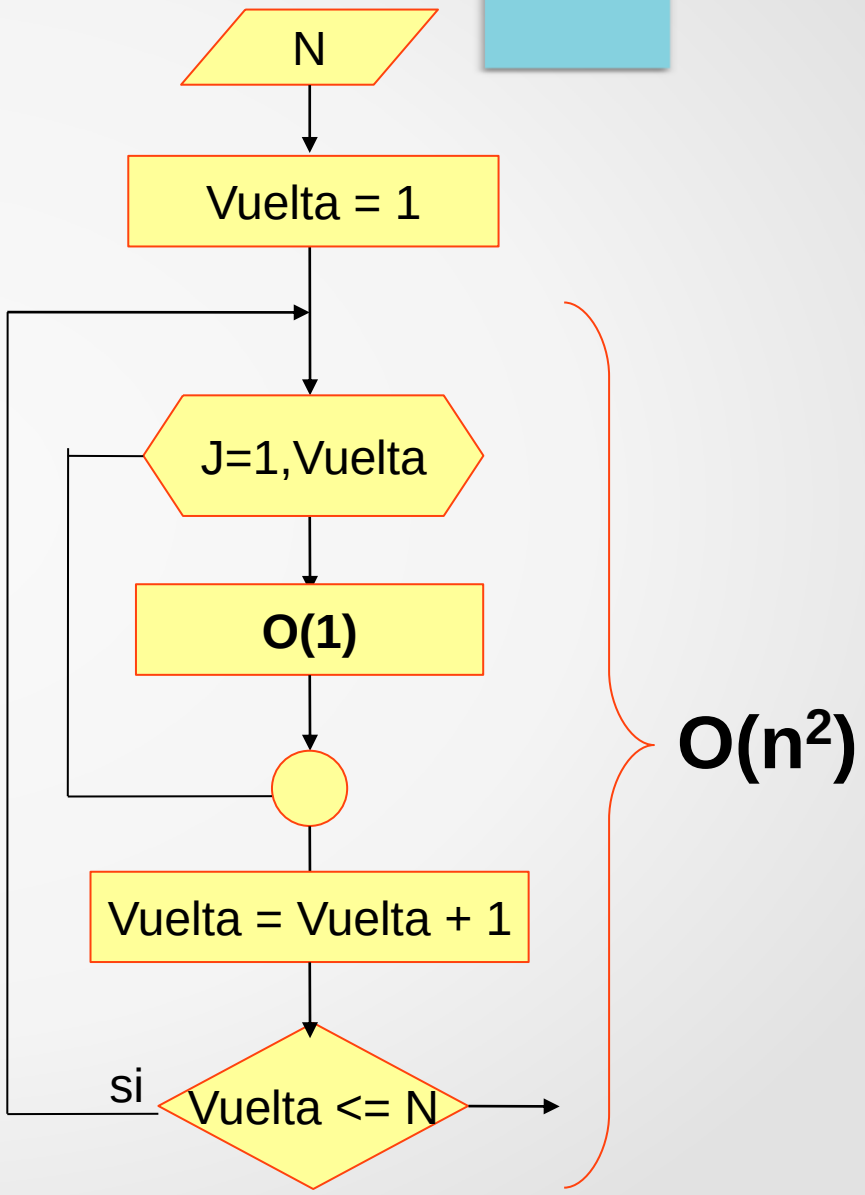
$O(n^2)$

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) → ?

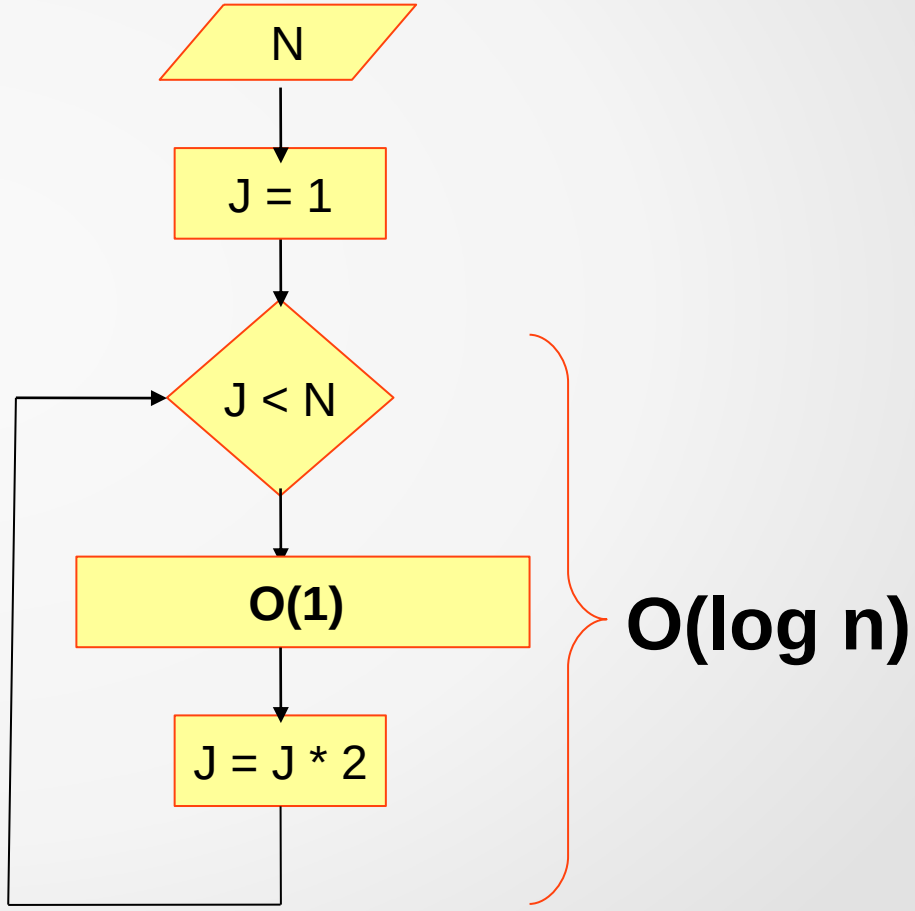


1.5.1 Complejidad en el Tiempo

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(N) * O(1) → ?

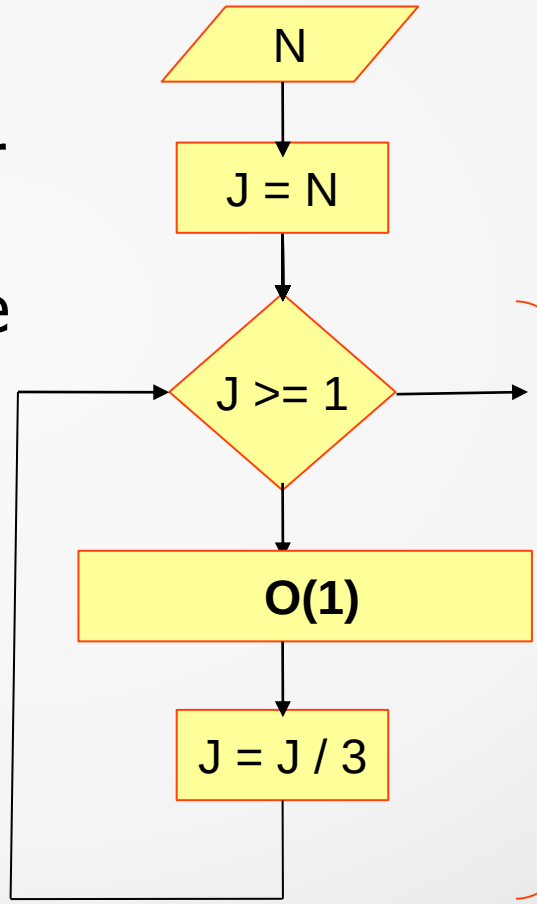


1.5.1 Complejidad en el Tiempo

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(N) * O(1) → ?



O(log n)

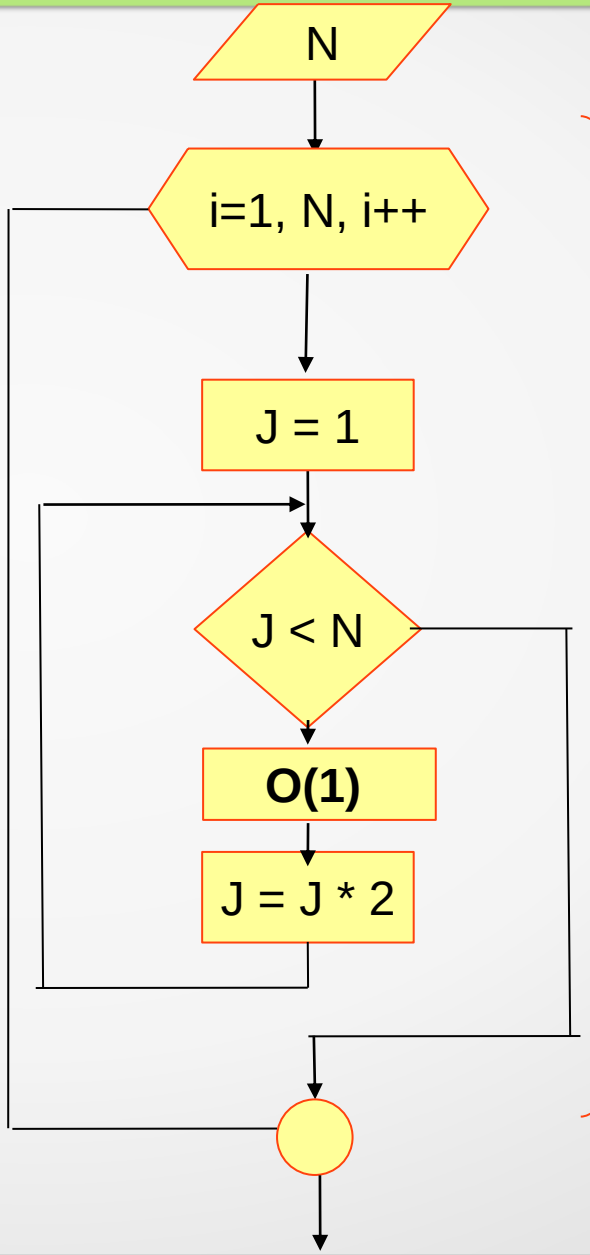
7.1 Complejidad en el Tiempo

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 → ?



O(n log n)

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
$O(n^3)$	Cubíco
$O(n^k)$ $k>3$	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

$\mathbf{O(1)+O(n^2)+O(n)} \rightarrow \mathbf{O(n^2)}$

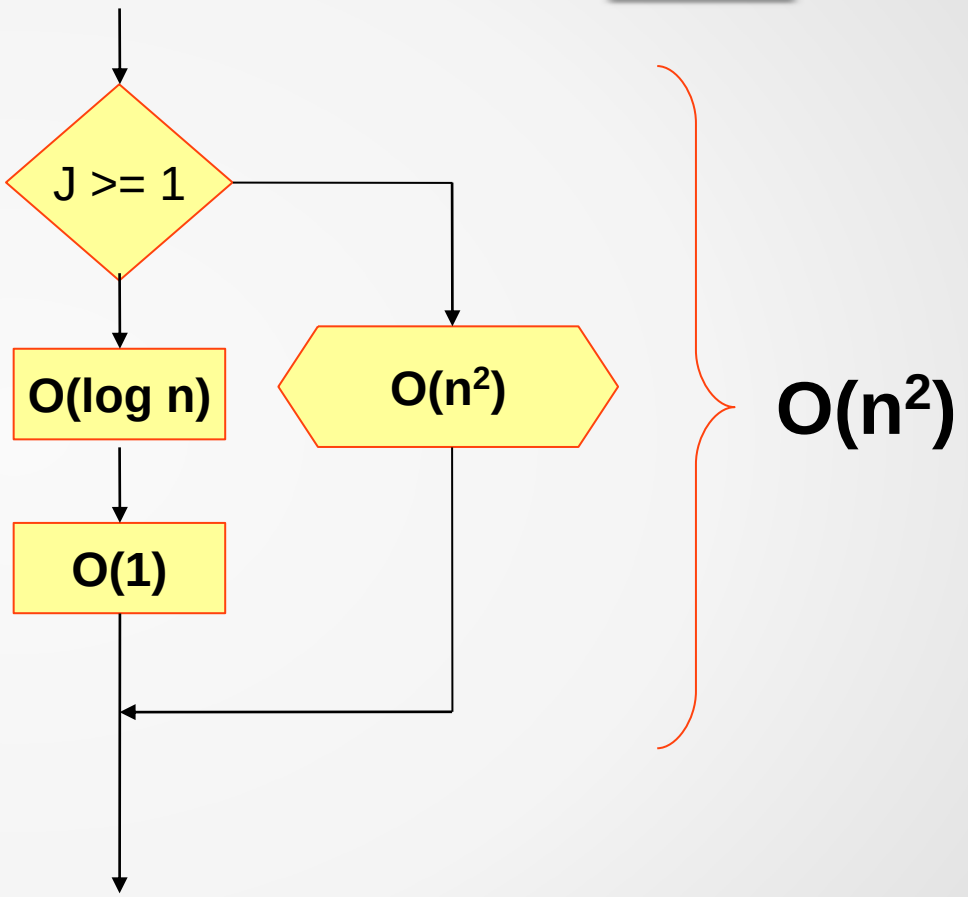
$\mathbf{O(1)+O(1)+O(1)} \rightarrow \mathbf{O(1)}$

$\mathbf{O(n)+O(n)+O(n)} \rightarrow \mathbf{O(n)}$

1.5.1 Complejidad en el Tiempo

e) Decisiones (if)

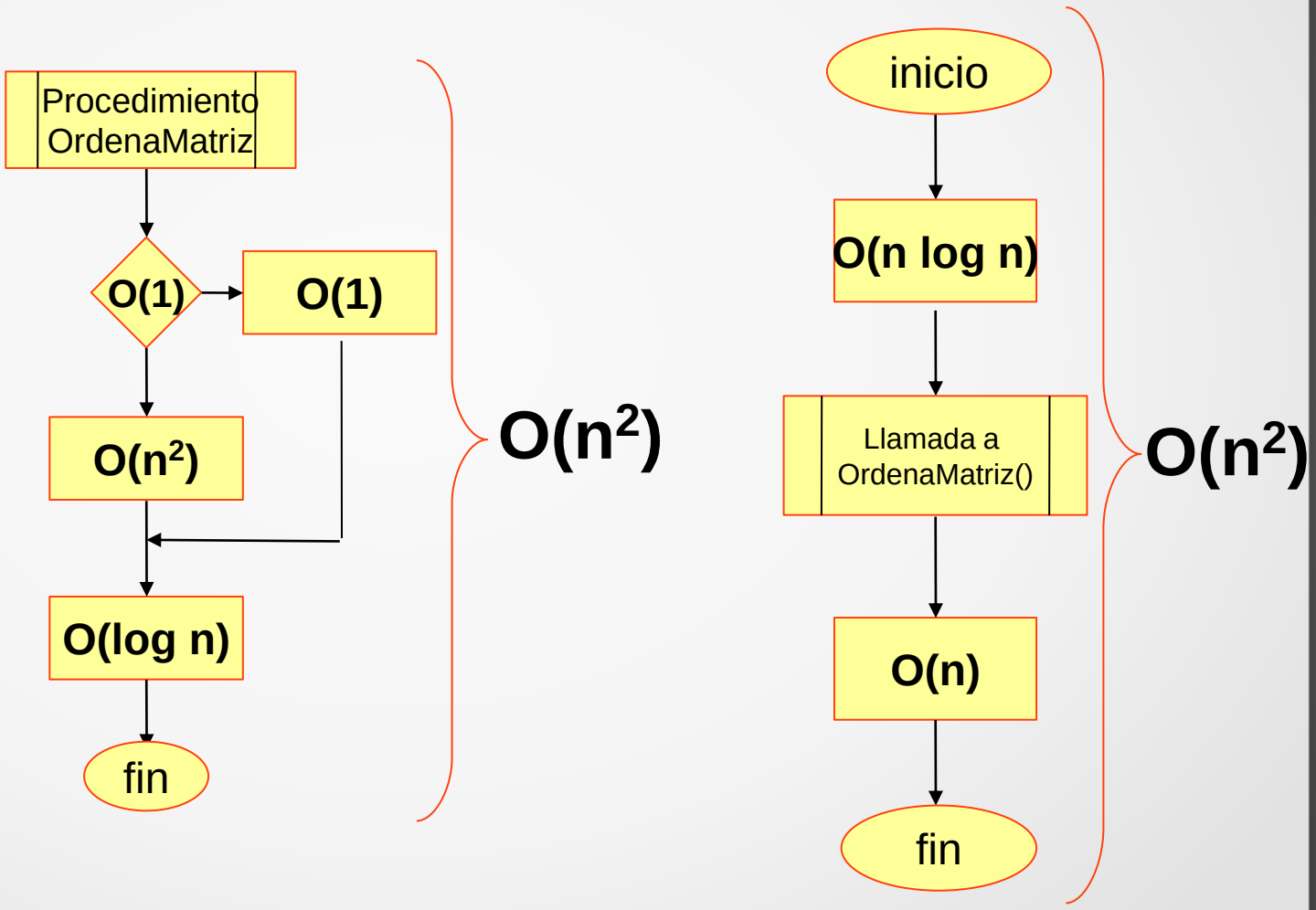
La evaluación de la condición generalmente es de orden **O(1)**, complejidad que **se sumará** con la **peor de las 2 ramas del if**, sea del **then o el else**, siguiendo las reglas de las secuencias.



1.5.1 Complejidad en el Tiempo

f) Procedimientos

La complejidad de **la llamada** 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 Compejidad 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 la función es **O(1)**, que se debe multiplicar por el número de veces que se ejecuta recurrentemente: $\mathbf{2^n}$ aproximadamente, como lo vimos en el árbol de ejecución (ver el tema correspondiente). Por lo tanto la complejidad de Fibonacci Recursivo es

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

Orden de Compejidad 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

O(1) * N → **O(n)**

1.5.2 Complejidad en el Espacio

1.5.2 Complejidad en el espacio

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

- RAM
- Discos Duros
- 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

Problema.

Se cuenta con un archivo que contiene los datos de todos los habitantes de algún país de **100 Millones** de habitantes y se busca **ordenarlos** con base a sus apellidos y nombre.

Si aplicamos el método de ordenamiento que se usó para el análisis empírico de la complejidad temporal, el tiempo de ejecución del programa **iséría de de años!**, ¿qué se podrá hacer para disminuir ese tiempo?

1.5.2 Complejidad en el espacio

Para los problemas de ordenamiento, una de las primeras aproximaciones para mejorar los algoritmos de orden $O(n^2)$, fue un método conocido como **Distribución Simple**.

Este método aplica de manera muy sencilla una técnica conocida como **"Divide y Vencerás"**.

Distribución Simple consiste en dividir de cierta forma el archivo grande en varias partes pequeñas.

- Las partes se ordenan por separado.
- Las partes se unen una después de otra.

1.5.2 Complejidad en el espacio

Los archivos pequeños se ordenan con menos esfuerzo que los grandes, esto es evidente para algoritmos **$O(n^2)$** observando las tablas que se crearon durante el análisis empírico de la complejidad temporal.

1 archivo de 100,000,000 registros = **2×10^{16}** instrucciones
(usando el polinomio del peor caso: $2n^2 - n + 1$)

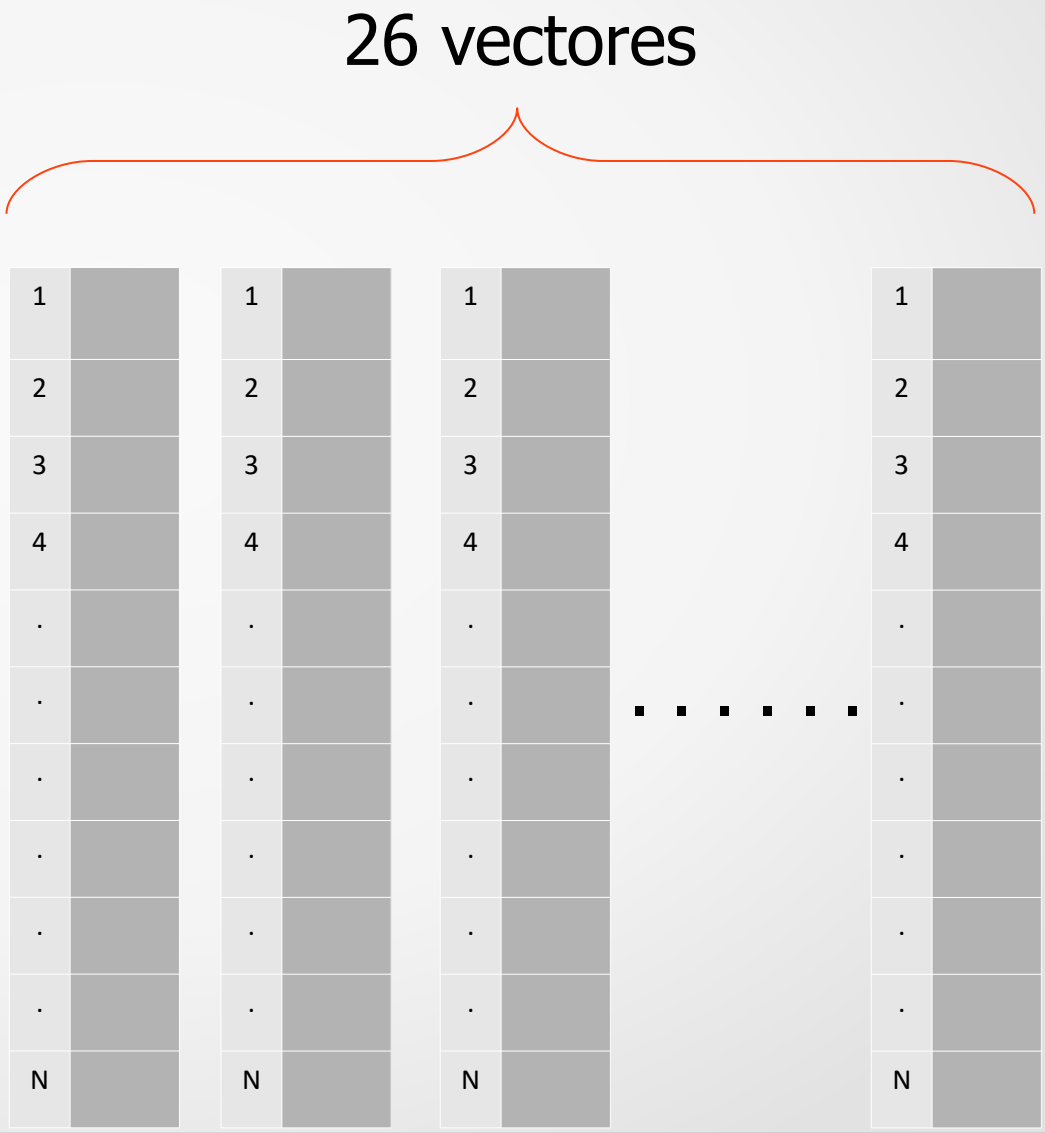
Si agrupamos el archivo según la letra inicial: **26** partes (subarchivos).

Tamaño aproximado de cada subarchivo: **4,000,000**
Número de Instrucciones para ordenar **c/subarchivo: 3.2×10^{13}**
Número total de instrucciones = **3.2×10^{13}** x **26 = 8.3×10^{14}**
(hay cierta mejoría)

1.5.2 Complejidad en el espacio

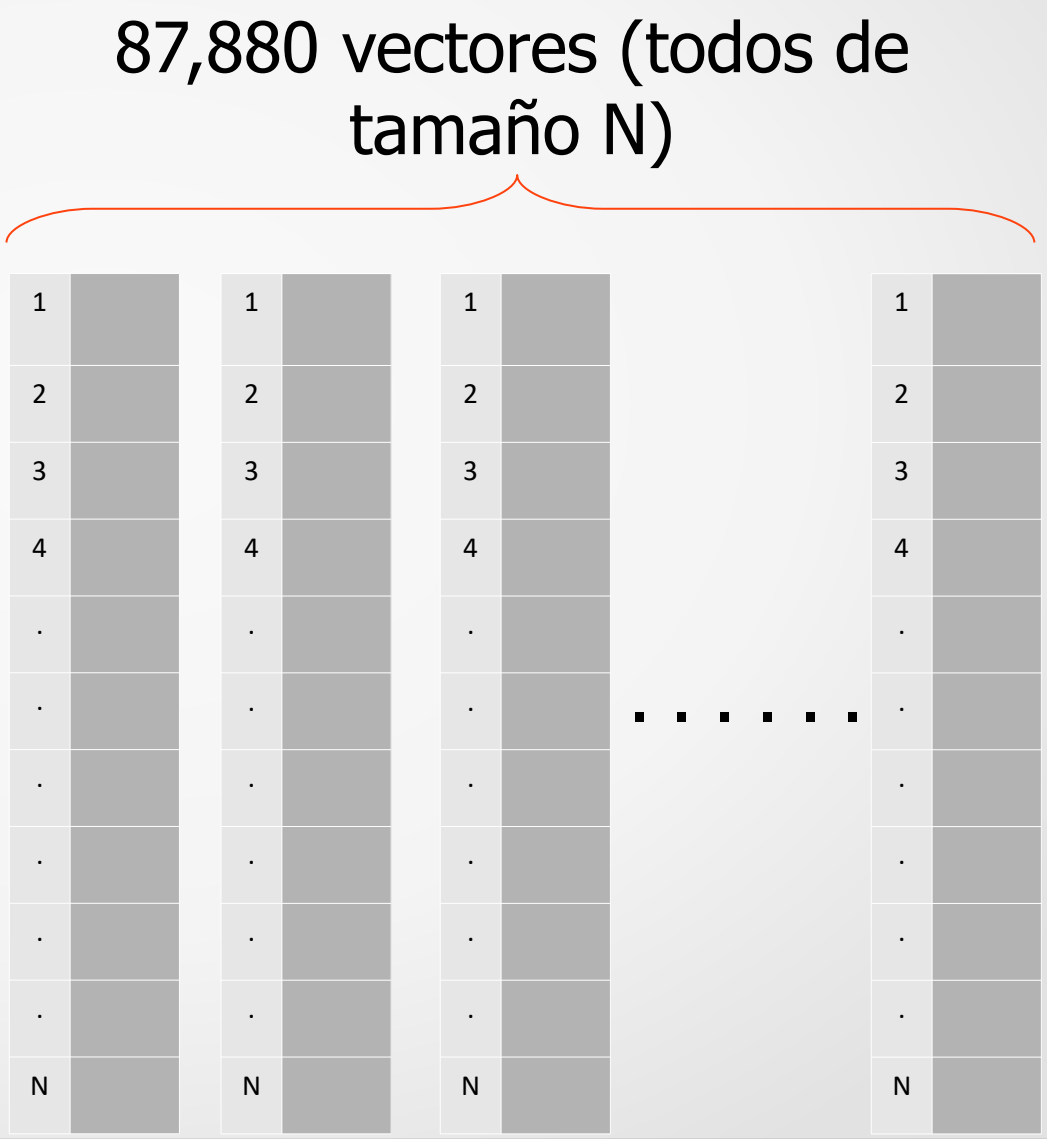
Estos intentos por disminuir el tiempo de ejecución causan un aumento en la complejidad espacial porque se requiere espacio extra para conservar los datos en archivos por separado.

Si cada "paquete" se guarda en un vector, el espacio requerido es muy grande (los vectores son de tamaño fijo N) ya que no sabemos cuántos datos van a quedar en cada vector (se requiere 26 veces el espacio original).



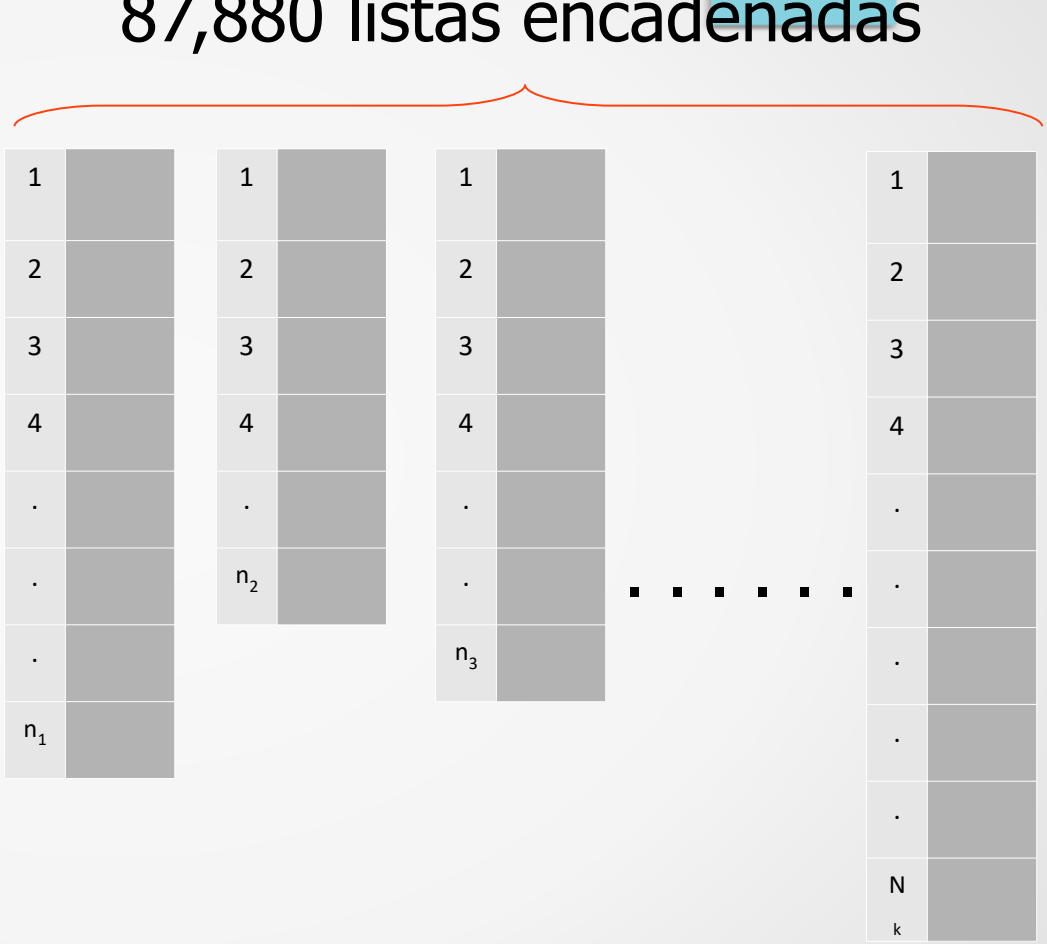
1.5.2 Complejidad en el espacio

Si en vez de solo la primera letra se toman las 4 primeras, y ya que la segunda letra siempre es una vocal, el número de vectores requeridos será $26^3 x 5$ (**87,880**).
Consideremos que la distribución de los datos será uniforme y el numero de registros contenidos en cada vector será de **1138** registros.
Número de instrucciones para ordenar cada subarchivo: **$2.6x10^6$** en total: **$2.3x10^{11}$** instrucciones, que no tomará mucho tiempo en un equipo moderno.



1.5.2 Complejidad en el espacio

Se pueden usar **Array Lists** o **Almacenamiento Secundario (SSD).** Usando estas estructuras aun así se requerirá el doble de espacio (el original y el espacio extra).



El orden de complejidad espacial de un algoritmo como el descrito es O(n) ya que el espacio que se requiere siempre es una k veces n, y cuando n tiende a infinito k es despreciable

1.5.2 Complejidad en el espacio

Tiene impacto negativo para la complejidad espacial:

- La cantidad de **datos** que **dependen de N**, por ejemplo, un vector o vectores que usemos, que contengan un dato para cada uno de los valores del conjunto original.
- Los **parámetros de valor** que dependen del tamaño **N**, ya que en la memoria se hace una copia de los datos (por ejemplo, un vector que se pase por valor).

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** (vectores, listas encadenadas, etc) cuyo tamaño no depende de **N**. Ejemplo: el tamaño del diccionario que se usa para revisar la ortografía de un texto.
- Los **parámetros por referencia**, ya que no se requiere espacio extra para ellos.
- La cantidad de **datos individuales** que no depende de **N**. por ejemplo, las variables declaradas.

1.5.3 Eficiencia de los algoritmos

Los mejores algoritmos son aquellos que poseen un equilibrio entre la Complejidad Temporal y la Complejidad Espacial. Es decir que sacrifican algo de tiempo por espacio o viceversa.

Siempre se puede seleccionar un algoritmo, como el de Distribución Simple que logra bajar el tiempo de procesamiento de años a días o hasta minutos en un problema solucionado mediante un algoritmo de complejidad temporal **$O(n^2)$**. Es importante observar que el algoritmo Distribución simple que de describió sigue siendo **$O(n^2)$**; solo soporta tamaños de N un poco más grandes.

1.5.3 Eficiencia de los algoritmos

Así como para complejidad temporal, la recomendación es buscar algoritmos con orden O(n log n), en complejidad espacial conviene que sean como máximo de orden O(n log n), es decir, que el espacio extra requerido sea como máximo algunas veces el tamaño del problema.

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