# 1. Eficiencia de un algoritmo

Actualmente, la velocidad de procesamiento de las computadoras ha crecido en forma considerable, esto hace pensar si es necesario “prestar atención” al estudio de la eficiencia de los algoritmos. Si bien existen problemas que son resueltos sin consumir demasiado almacenamiento y tiempo, otros dependen del tamaño de los datos considerados, pudiendo ocurrir que un pequeño aumento en su número ocasione un incremento considerable en el tiempo de ejecución.

Así por ejemplo, si para solucionar un problema se cuenta con un algoritmo que requiere de 10-4.2N segundos, para N elementos, si el número de elementos es tan solo 38, se necesitaría una computadora trabajando sin interrupción durante gran cantidad de tiempo para resolverlo.

Si es posible encontrar un algoritmo que en lugar de utilizar un tiempo exponencial lo haga en uno polinomial por ejemplo, se podría incluso resolver el problema para mayor cantidad de elementos en menor tiempo.

Se llama **complejidad de un algoritmo** a la medida de los recursos que necesita para su ejecución.

La Algorítmica, estudia técnicas para realizar algoritmos eficientes.

Un **algoritmo es más eficiente** que otro, cuando optimiza los recursos del sistema en el que se ejecuta en la resolución de un determinado problema.

Los principales recursos a los que hacemos referencia son **tiempo de ejecución y almacenamiento en memoria,** por ello se habla de complejidad espacial y complejidad temporal.

**Complejidad espacial** es una medida de la cantidad de espacio o almacenamiento ocupado por un programa.

**Complejidad temporal** es una medida de la cantidad de tiempo que requiere la ejecución de un programa.

Si bien esta unidad se centrará en el análisis teórico de la **Complejidad Temporal**, es importante considerar el almacenamiento **en memoria**, dado que serán más eficientes aquellos algoritmos que utilicen una estructura de datos adecuada que minimice el espacio de memoria utilizado.

Desde el punto de vista del tiempo de ejecución, se considerarán más eficientes aquellos algoritmos que resuelvan el mismo problema en el menor tiempo posible.

El tiempo de ejecución de un programa depende de factores tales como:[[1]](#footnote-1)

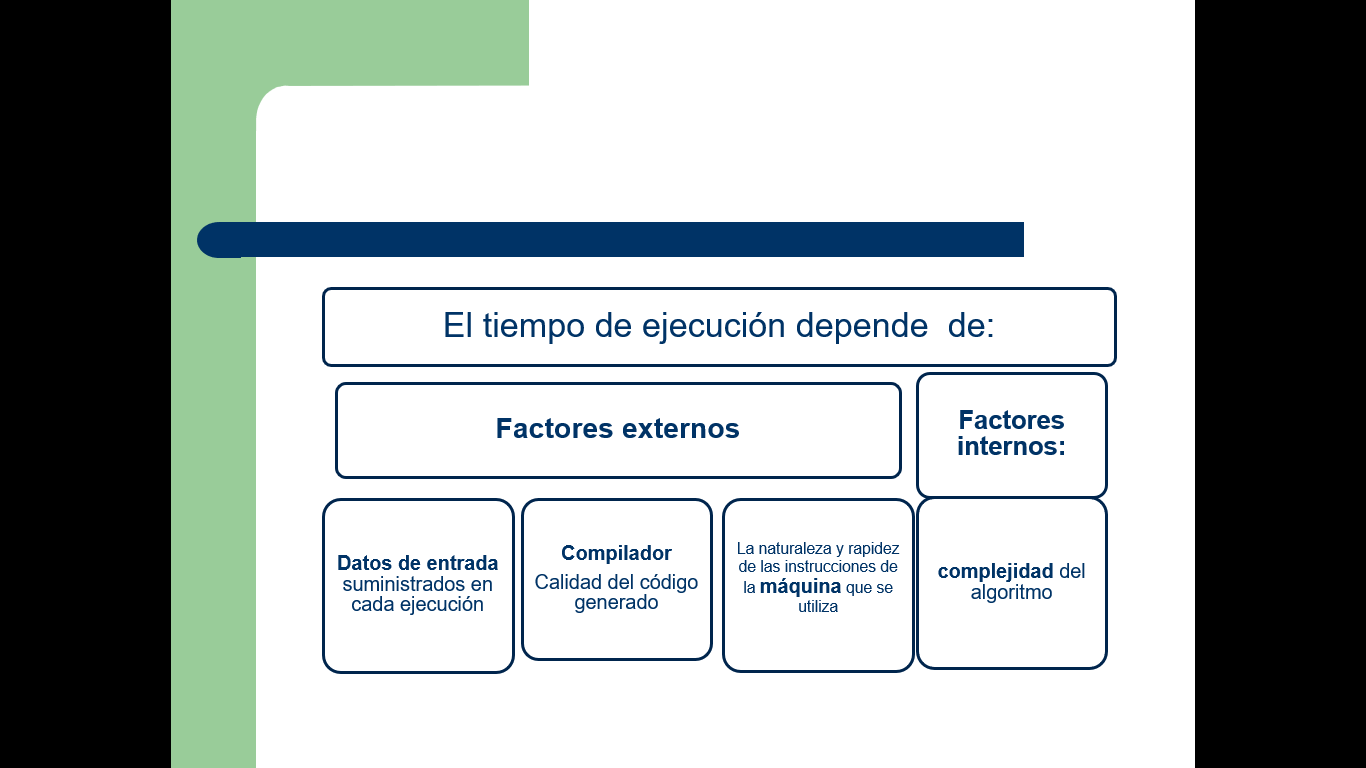
1. los datos de entrada al programa

2. la calidad del código generado por el compilador utilizado para crear el programa objeto

3. la naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa

4. la complejidad de tiempo del algoritmo de base del programa

Como se observa, algunos factores son propios del problema o factores internos (4) y otros dependen de la máquina utilizada o factores externos (1, 2 y 3).



# 2 Análisis de algoritmos según el tiempo de ejecución

Los siguientes ejemplos se refieren a problemas sencillos que pueden resolverse de distintas formas, lo que permitirá realizar un análisis somero de los aspectos a tener en cuenta para la optimización de los tiempos de ejecución.

**Ejemplo 1**:

Dados tres valores distintos, calcular el mayor.

Las siguientes son algunas posibilidades de solución de este problema:

**Forma 1**

Algoritmo Mayor1

entero **mayor1**(entero xm,xn,xp)

comienzo

Si ((xm> xn) y (xm>x p))

entonces retorna(xm)

Sino Si (xn> xp)

entonces retorna( xn)

sino retorna(xp)

FinSi

FinSi

fin

**Comienzo /\* Algoritmo Principal \*/**

entero m, n, p, mayor

Leer m,n,p

Escribir “el mayor número leído es “, **mayor1**(m,n,p)

**Fin**

**Forma 2**

Algoritmo Mayor2

entero mayor2(entero xm,xn,xp)

comienzo

Si (xm> xn)

Entonces

Si (xm>xp)

Entonces retorna( xm)

Sino retorna(xp)

FinSi

Sino Si (xn>xp)

Entonces retorna(xn)

Sino retorna(xp)

FinSi

FinSi

Fin

**Comienzo /\* Algoritmo Principal \*/**

Entero m,n,p,mayor

Leer m,n,p

Escribir “ el mayor número leído es “, **mayor2(**m,n,p)

**Fin**

**Forma 3**

Algoritmo Mayor3

entero mayor3 (entero xm,xn,xp)

comienzo

Si ((xm> xn) y (xm>xp))

Entonces retorna(xm)

FinSi

Si (( xn>xm) y (xn>xp))

Entonces retorna(xn)

FinSi

Si ((xp>xm) y (xp>xn))

Entonces retorna( xp)

FinSi

Fin

**Comienzo /\* Algoritmo Principal \*/**

entero m,n,p,mayor

Leer m,n,p

Escribir “el mayor número leído es “, **mayor3**(m.n.p)

**Fin**

Comparando las tres alternativas de solución, se puede concluir que en el caso de la **forma 1**, dependiendo de los valores ingresados, se puede realizar como mínimo 2 comparaciones (si m es el mayor) y como máximo 3. En cualquier caso se realiza un único retorno (que equivale a una asignación).

Para el caso de la **forma 2**, se realizan 2 comparaciones y un único retorno, independientemente del valor de las variables.

De estas 2 formas, la segunda arroja más eficiencia.

La **forma 3** es la menos óptima ya que se realizan entre 2 y 6 comparaciones dependiendo de los valores ingresados. Para el caso en que el valor m fuera el mayor, se harían 2 comparaciones sin embargo si p es el mayor se harían 4 comparaciones demás respecto del algoritmo más óptimo. Cuatro comparaciones demás para este ejemplo sencillo no es relevante. Pero ¿qué ocurre si el analista de una compañía de teléfonos de una ciudad utiliza este algoritmo para determinar, para cada uno de sus abonados, el consumo máximo de los 3 bimestres correspondientes al primer semestre del año?. Utilizar en este caso la tercera forma de este algoritmo, implicaría un derroche innecesario de tiempo.

**Ejemplo 2:** Otro aspecto a tener en cuenta para reducir tiempos de ejecución, consiste en no repetir cálculos, sobre todo si ellos forman parte de un ciclo, como se muestra en el ejemplo.

*Situación problemática***:** Una empresa desea realizar el control de calidad de los envases de conservas que en ella se fabrican. Se sabe que diariamente se producen 20.000 envases de forma cilíndrica y que las especificaciones son las siguientes: la superficie de la tapa debe estar comprendida en el rango 50± 0.05 cm2 y el volumen del recipiente entre 600± 0.5 cm3 respectivamente. Se necesita especificar las tapas o envases cilíndricos que deben ser eliminados por no cumplir los requisitos establecidos. Además se debe informar a gerencia, la cantidad de piezas defectuosas de cada tipo (tapas y recipientes) y el porcentaje que esas cantidades representan sobre el total producido. Los datos que se conocen para cada envase fabricado son el diámetro de la tapa y la altura del recipiente cilíndrico.

El algoritmo siguiente muestra una forma de resolución del problema.

Algoritmo Envases

constante pi= 3.14159

real superficie (real xdiametro)

comienzo

superficie = pi\* (xdiametro/2 ) \*(xdiametro/2)

fin

real volumen( real xdiametro, real xaltura)

comienzo

volumen= pi\* (xdiametro/2 ) \*(xdiametro/2) \*xaltura

fin

**Comienzo /\* Algoritmo Principal \*/**

entero Ct, Cc, i

real diametro, altura, sup, Vol

Ct=0

Cc=0

Para i Desde 1 Hasta 20000

Leer diametro, altura

sup= superficie (diametro)

Si ((( **sup- 50**)> 0.05 ) o (( **sup- 50**) < -0.05)))

Entonces Escribir “ Tapa defectuosa”

Ct= Ct+1

FinSi

Vol= volumen(diametro,altura)

Si ( ( **Vol- 600**)> 0.5) o ( ( **Vol- 600**)<- 0.5))

Entonces Escribir “ Recipiente defectuoso”

Cc= Cc+1

FinSi

FinPara

Escribir “ Detectadas ”, Ct ,“ tapas defectuosas Porcentaje ”, Ct\***100/ 20000**

Escribir “Detectados ”, Cc, “ cilindros defectuosos Porcentaje ”, Cc\***100/ 20000**

**Fin**

Este algoritmo optimizará su tiempo de ejecución si las operaciones resaltadas son reemplazadas como se muestra a continuación.

Algoritmo EnvasesOptimo

Pi= 3.14159

real superficie (real xradio)

comienzo

superficie = pi\* (xradio/2 ) \*(xradio/2)

fin

real volumen( real xradio, real xaltura)

comienzo

volumen= pi\* (xradio ) \*(xradio) \*xaltura

fin

**Comienzo /\* Algoritmo Principal \*/**

entero Ct, Cc, i

real diametro, altura, sup, Vol, radio, dif

Ct=0

Cc=0

Para i Desde 1 Hasta 20000

 Leer diametro, altura

radio = diametro/2

sup = superficie( radio)

**dif = sup-50**

Si (( **dif**)> 0.05 ) o (( **dif**) < -0.05)

Entonces Escribir “ Tapa a eliminar”

Ct= Ct+1

FinSi

Vol = volumen( radio,altura)

**dif =Vol-600**

Si ( ( **dif**)> 0.5) o ( ( **dif**)<- 0.5))

Entonces Escribir “ Cilindro a eliminar”

Cc= Cc+1

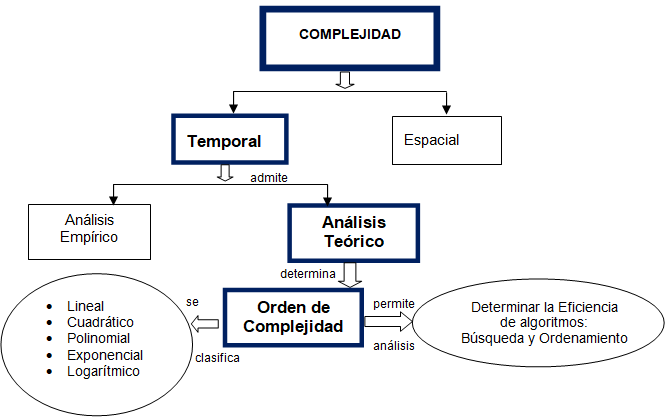
FinSi

FinPara

Escribir “ se han detectado ”, Ct, “ tapas defectuosas Porcentaje ”, Ct/ 200

Escribir “ se han detectado ”, Cc, “ cilindro defectuosos Porcentaje ”, Cc/ 200

**Fin**





## 2.1. Tiempo de ejecución de un Algoritmo: Análisis empírico y Análisis teórico

Existen **dos maneras de estimar el tiempo de ejecución** de un algoritmo, mediante un **análisis empírico o uno teórico**.

El **análisis empírico** o a posteriori, consiste en la comparación de los tiempos de ejecución de distintos algoritmos que resuelven un problema, realizando pruebas con diferentes lotes de datos.

Los algoritmos deberán codificarse en un lenguaje de programación y así se podrán calcular los tiempos, mediante la utilización de acciones que soliciten al sistema los momentos de comienzo y finalización de ejecución. Se obtiene de esta manera una medida real del tiempo en el entorno de aplicación.

En general los problemas del análisis empírico se pueden resumir en la dependencia de los resultados obtenidos del tipo de computadora, del lenguaje de programación usado, del traductor con el que se obtenga el código ejecutable, y de la habilidad del programador. Es decir que depende de los factores externos e internos.

Si se modifica alguno de estos elementos probablemente se obtengan resultados diferentes, con lo cual no se puede establecer una eficiencia empírica absoluta.

Por otra parte, existen algoritmos que pueden ser comparados con esta técnica sólo cuando el número de datos con los que se ejecutan es relativamente pequeño.

El **análisis teórico**, en cambio, favorece la independencia con todos los factores anteriores. Es válido para cualquier entrada, depende sólo de las instrucciones que componen el algoritmo y del tamaño del problema; es decir, depende solamente de factores internos.

Un análisis teórico permitirá estimar matemáticamente el orden del tiempo de respuesta, independientemente de los datos experimentales.



Del análisis teórico se obtiene una expresión matemática que indica cómo se produce el crecimiento del tiempo de ejecución a medida que aumenta el tamaño del problema. No necesita la implementación y ejecución del algoritmo.

Se denomina **tamaño de un problema** al número de datos del problema. Por ejemplo, si se desea ordenar un arreglo de N elementos, el tamaño del problema es N.

Se llama **instancia de un problema** a un caso particular del problema.

Debe tenerse en cuenta que, para un mismo tamaño, el algoritmo puede comportarse de distintas maneras. Así por ejemplo si se desea buscar un elemento en un arreglo, el tamaño del problema es la cantidad de elementos que tiene el arreglo. Sin embargo, el comportamiento de una búsqueda secuencial será distinto si el elemento se encuentra en la primera posición (el algoritmo finaliza enseguida) o si se encuentra en la última posición (el algoritmo debe recorrer todo el arreglo).

Por esto, cuando se analiza la eficiencia de un algoritmo, se podrían **estudiar tres situaciones importantes:**

* el mejor caso o cota inferior
* el peor caso o cota superior
* el caso medio o cota promedio.

Por ejemplo, si el algoritmo consiste en ordenar ascendentemente un vector, el mejor caso es cuando el arreglo está ordenado de forma creciente, el peor caso cuando esté ordenado de manera decreciente y el caso medio es cuando sus componentes están organizadas en forma aleatoria. En todos los casos el tamaño del problema es el mismo.

Aunque el análisis del término medio parece ser el más adecuado, puede ocurrir que algunos datos de entrada tengan mayor probabilidad que otros. En estos casos la media no es el promedio del peor y mejor caso, sino que debe ser ponderada, lo que implica realizar cálculos de probabilidades, situación que queda fuera en términos generales de nuestro objetivo.

El estudio del peor caso, es útil para problemas cuyos tiempos de respuesta son críticos, como por ejemplo los sistemas de tiempo real utilizados en medicina, home-banking, comercio electrónico, y sistemas de seguridad controlados por software.

El enfoque teórico ayudará a establecer la frontera entre lo factible y lo imposible y tiene además la ventaja de no depender de la computadora ni del lenguaje de programación utilizado.

En este libro se utilizarán técnicas elementales que permitirán determinar la eficiencia de algoritmos referidos a búsquedas y ordenamiento, para el peor de los casos. Si bien existen algoritmos más eficientes, para resolver algunos de los ejemplos que se presentan, requieren de técnicas avanzadas, cuyo abordaje está sustentado sobre una base teórica que está fuera del alcance de este libro, destinado a cursos de inicio de programación.

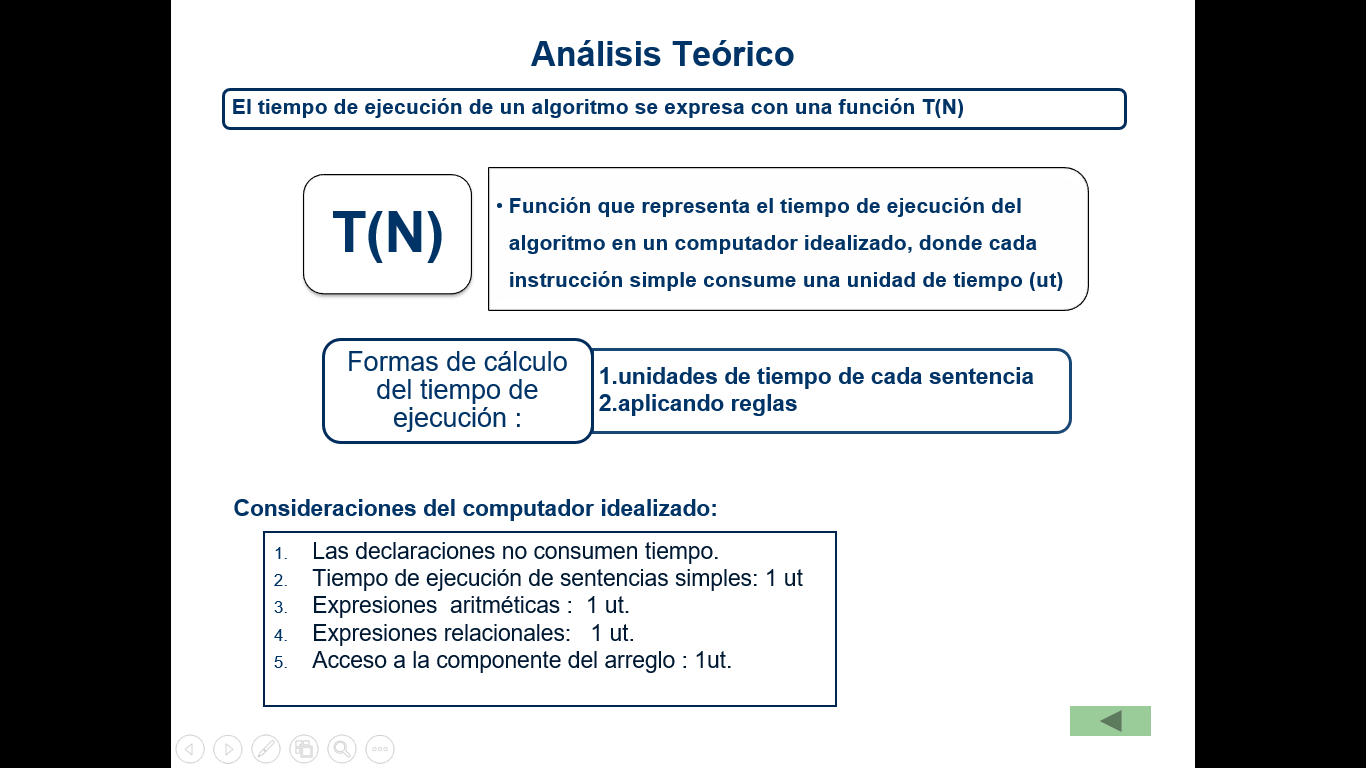
## 2.2 Orden de Complejidad de un algoritmo - Eficiencia asintótica

Un análisis teórico determina el orden del tiempo de respuesta u ***orden de complejidad****,* que es utilizado para la comparación de diferentes soluciones algorítmicas sin depender de datos experimentales.

Para poder obtener el Orden de complejidad de un algortimo, es necesario en primer lugar calcular el Tiempo de ejecución del mismo, representado por una función T(N).

**T(**N)representa lasunidades de tiempo (segundos, milisegundos,...) que un algoritmo tardaría en ejecutarse con unos datos de entrada de tamaño N.

T(N) corresponde al tiempo de ejecución del algoritmo en un ordenador idealizado, donde cada una de las instrucciones simples consume una unidad de tiempo.



Al estudio de la complejidad para tamaños grandes de problema se lo conoce como **eficiencia asintótica** del algoritmo. Este análisis clasifica las funciones T(N) en clases constituidas por funciones que crecen de la misma forma. De esta forma pueden compararse fácilmente entre sí las funciones correspondientes a los algoritmos que resuelven un mismo problema.

La notación matemática que se utiliza para representar el orden de complejidad de un algoritmo cuando el tamaño del problema es grande (notación asintótica) es una **O** mayúscula, **O**(f(N)) , conocida también como **o grande** para representar la cota superior y **Ω** (omega) para representar la cota inferior .

## Eficiencia asintótica para el peor caso. Notación O

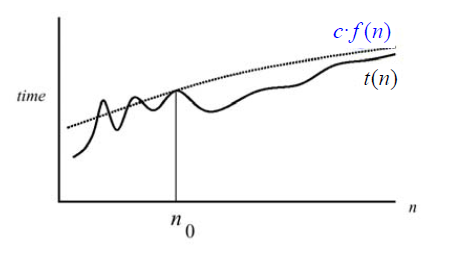
Si el tiempo de ejecución de un algoritmo se expresa con T, se dice que *T es de orden f(*N) y se simboliza T **ϵ** O(f(N)), si existe una función matemática f(N) que lo acota.

Dicho de otro modo:

T **ϵ** O(f(N)) si existen constantes c y N0 / T ≤ c.f(N) para N ≥ N0

Al decir que T **ϵ** O(f(N)) se está garantizando que la función T no crece más rápido que f(N), esto es que f(N) es un límite superior para T- La notación O-grande hace referencia al peor caso.

Cuando el tiempo de ejecución de un programa es O(f(N)), se dice que tiene velocidad de crecimiento f(N).

****

***c.f(N)***

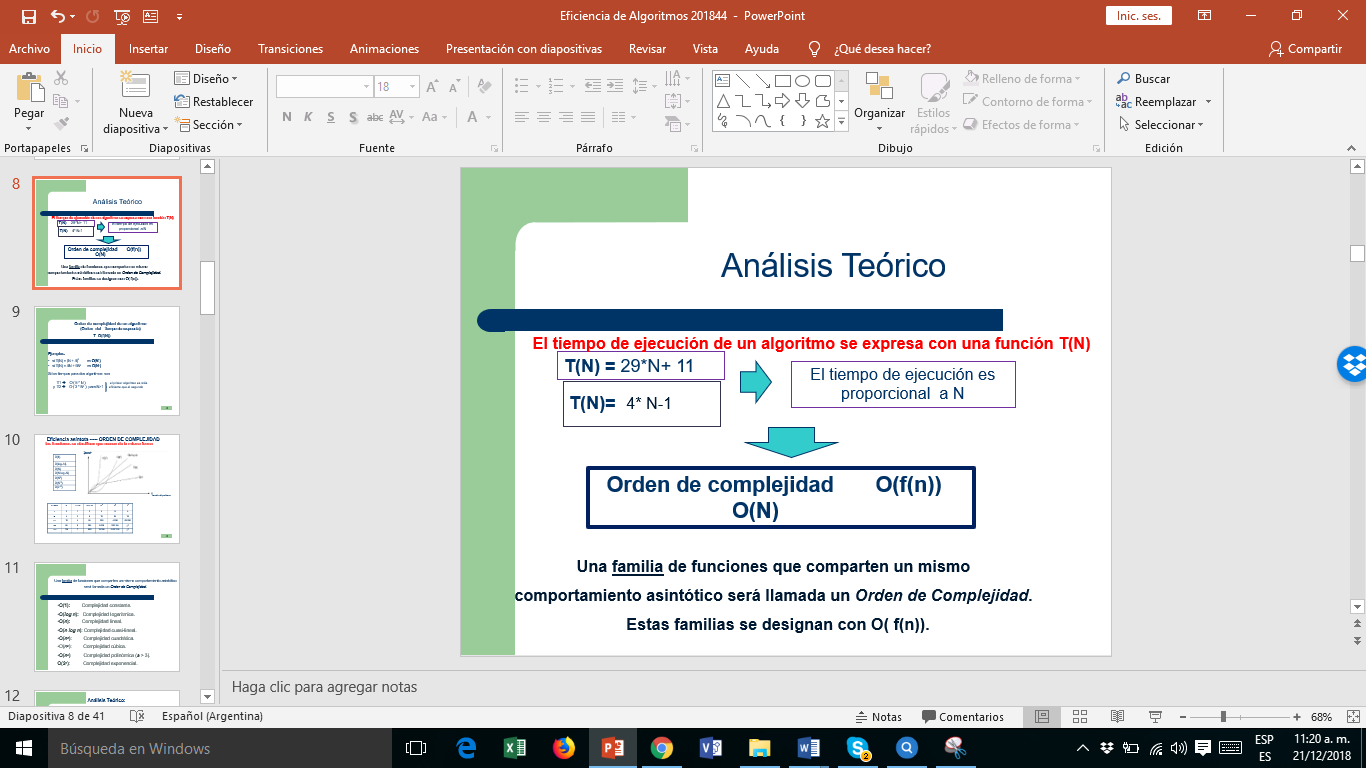
***N0***

***T(N)***

***n***

***Tiempo***

*Figura 4.2 Cota superior*



**Ejemplo 3**

* + *n+5*  **ϵ** **O(n)** pues *n+5* ≤ *2n* para toda ***n*** ≥ 5
  + 3n + 2 **ϵ** **O(n),** pues 3n+2 ≤ 4n para toda ***n*** ≥ 2
  + ***(n+1)2*****ϵ** **O(n2)** pues ***(n+1)2***≤ ***4n2***para ***n***≥ 1.
  + 1000n2 +100n -6 **ϵ** **O(n2**), porque 1000n2 +100n -6 ≤ 1.001n2 para n≥ 100
  + ***(n+1)2*** **NO** **ϵ** **O(n)** pues para cualquier ***c > 1*** no se cumple que ***(n+1)2***≤ ***c\*n***

Entones, en general:

si T(n) = (n + 5)2 se dice que **ϵ** O(n2)

si T(n) = 4n + 6n3 se dice que **ϵ** O(n3)

Los órdenes de complejidad permiten comparar la eficiencia de algoritmos, por lo cual:

Si se determina que los tiempos para dos algoritmos que resuelven un determinado problema son T1 = O(3 \* N ) y T2 = O ( 2 \* N2 ) para N>1 , se puede inferir que el primer algoritmo es más eficiente que el segundo.

Un aspecto importante a considerar al comparar los tiempos de ejecución de algoritmos es la velocidad de crecimiento de las funciones. Así por ejemplo si se deben comparar las funciones:

T1 = O ( 500 \* N ) y T2 = O ( N2 )

Para N pequeño el algoritmo 2 es más eficiente, pero se debe analizar que ocurre para valores de N mayores que 500.

Hay ciertos órdenes de complejidad que se producen con tanta frecuencia que se les ha dado nombre. Si el tiempo en que un algoritmo resuelve un caso de tamaño N, nunca supera a c\*N segundos, siendo c una constante, se dice que el algoritmo es de orden N o que requiere un *tiempo lineal* de procesamiento.

Si el tiempo no supera a c\*N2 segundos, se dice que el algoritmo es de orden N2 o que requiere un *tiempo cuadrático* de procesamiento.

Si el tiempo no supera a c\*N\*logN segundos, se dice que el algoritmo es de orden N\* logN o que requiere un *tiempo Cuasilineal* de procesamiento

Un orden **O(N)** se llama complejidad lineal, indica que el tiempo de ejecución crece en proporción directa al crecimiento de **N**.

Un algoritmo de orden **O(1**) tiene complejidad constante, suelen ser los más eficientes y preferidos.

La mayoría de los programas tienen complejidad polinomial, **O( Na)** donde **N** es la variable y **a** es una constante mayor o igual que 1. Son ejemplos de este tipo de complejidad, la lineal O(N), la cuadrática O(N2), y la complejidad cúbica O(N3)

Para un tiempo de ejecución cuadrático, si N se duplica, el tiempo de ejecución aumenta cuatro veces.

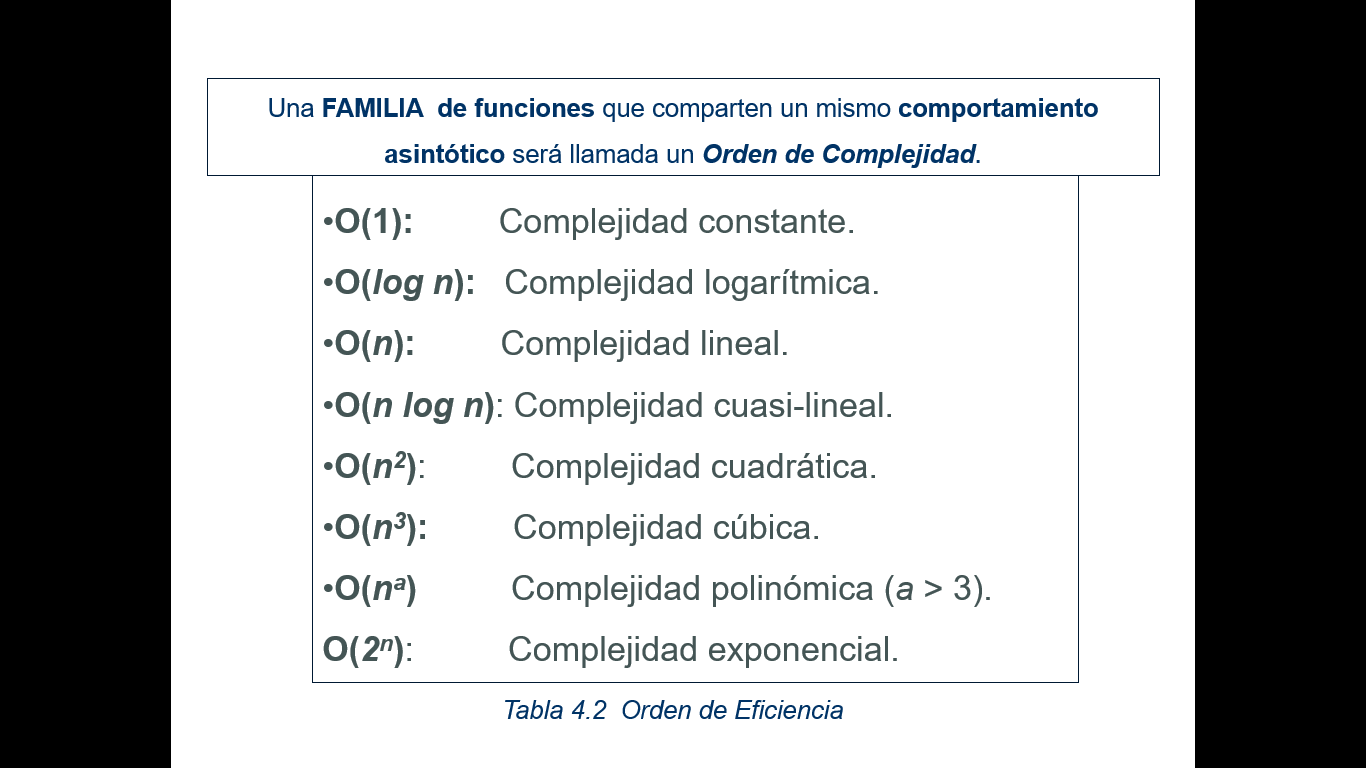
Para un tiempo de ejecución cúbico, si N se duplica, el tiempo de ejecución se multiplica por ocho.

Existen algoritmos que tienen complejidad logarítmica **O(log N )**.

Generalmente, el tiempo de ejecución de un algoritmo es proporcional a alguno de los tiempos descriptos o una combinación de ellos.

Así, el tiempo de ejecución de un algoritmo puede ser T = 4N2 + 2N. En este caso se dice que el tiempo de ejecución es proporcional a N2 o de orden cuadrático, lo cual se simboliza O(N2)

## Clasificación del Orden de Complejidad



La siguiente tabla[[2]](#footnote-2) muestra una comparación entre diferentes complejidades.

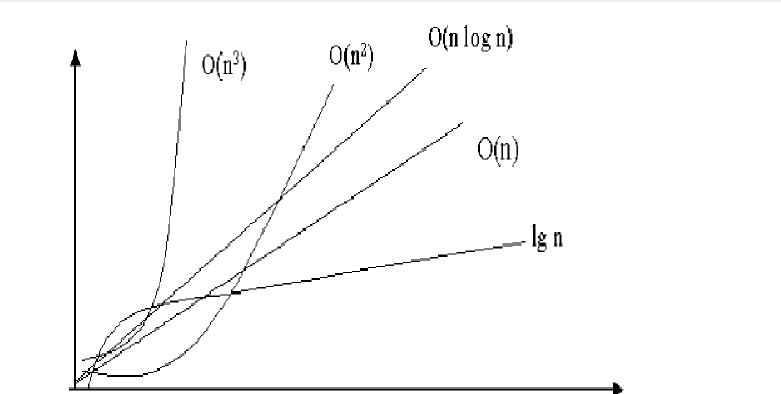
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nº elemen-tos** | **N** | **lg n** | **n lg n** | **n2** | **n3** | **2n** | **3n** | **n!** |
| 2 | 2 | 1 | 2 | 4 | 8 | 4 | 9 | 2 |
| 4 | 4 | 2 | 8 | 16 | 64 | 16 | 81 | 24 |
| 8 | 8 | 3 | 24 | 64 | 512 | 256 | 6.561 | 40.320 |
| 16 | 16 | 4 | 64 | 256 | 4.096 | 65.536 | 43.046.721 | 20.922.789.888.000 |
| 32 | 32 | 5 | 160 | 1.024 | 32.768 | 4.294.967.296 | 1.853.020.188.851.841 | ¿ ? |
| 64 | 64 | 6 | 384 | 4.096 | 262.144 | ¿? | ¿? | ¿ ? |
| 128 | 128 | 7 | 896 | 16.384 | 2.097.152 | ¿? | ¿? | ¿ ? |

*Tabla 4.1 Comparación de órdenes de eficiencia*

**NOTA**: Con el símbolo **¿?** (que aparece en la tabla) se quiere reflejar tiempos extremadamente grandes.

De esta tabla se puede inferir que los algoritmos polinomiales, aquellos que son proporcionales a Nk, es decir de la forma c\*Nk siendo c una constante, obtienen sus resultados en un tiempo limitado. En cambio en el caso de los algoritmos exponenciales, aquellos que son proporcionales a kN, los resultados se obtienen en tiempos desmedidamente grandes y en general se tornan poco probables de utilizar, salvo para tamaños de entrada de datos muy reducidos.

La comparación gráfica de las funciones de complejidad más frecuentes se muestra en la figura 4.2



*Figura 4.3 Comparación de órdenes de eficiencia[[3]](#footnote-3)*

El mejor orden es el logarítmico, si se duplica el tamaño del problema, no afecta al tiempo.

Para los órdenes *cuasilineal* ( N.logN) y lineal, si se duplica el tamaño del problema se duplica, aproximadamente, el tiempo empleado.

Los problemas de orden polinomial necesitan mucho tiempo para resolver un problema que ha crecido relativamente poco en tamaño, sin embargo, se consideran tratables.



*Figura 4.4 Tiempo de ejecución de cuatro programas*

|  |  |  |  |
| --- | --- | --- | --- |
| Tiempo de  Ejecucion T(n) | Tamaño máximo  de problema  para 103 seg | Tamaño máximo  de problema  para 104 seg | Incremento en el tamaño máximo  de problema |
| 100n  5n2  N3/2  2n | 10  14  12  10 | 100  45  27  13 | 10.0  3.2  2.3  1.3 |

*Tabla 4.3 Efecto de multiplicar por diez la velocidad del computador*

En la figura 4.3 y tabla 4.3 [[4]](#footnote-4), se observan los tiempos de ejecución de cuatro algoritmos de distintas complejidades de tiempo, medidas en segundos. Si se dispone de 1000 segundos (aproximadamente 17 minutos) para resolver un determinado problema, se infiere que los tres algoritmos pueden resolver problemas de un tamaño parecido (desde N=10 a N=12).

Suponga que ahora cuenta con una computadora que funciona 10 veces más rápido, entonces con el mismo costo se puede dedicar para resolver el mismo problema 10.000 seg. Como se observa ahora el tamaño del problema a resolver varía significativamente entre los algoritmos. Se advierte además que los algoritmos de orden exponencial sólo pueden resolver problemas pequeños, independientemente de la rapidez de la computadora.

En la tercera columna de la Tabla 4.3, se observa la superioridad del algoritmo con eficiencia lineal O(n), que permite un aumento de 1000% en el tamaño del problema para un incremento del 1000% en la velocidad de la computadora.

Los programas con eficiencia O(n3) y O(n2) permiten aumentos del 230% y 320% en el tamaño del problema para el mismo incremento de velocidad.

## Relación entre los Órdenes de complejidad



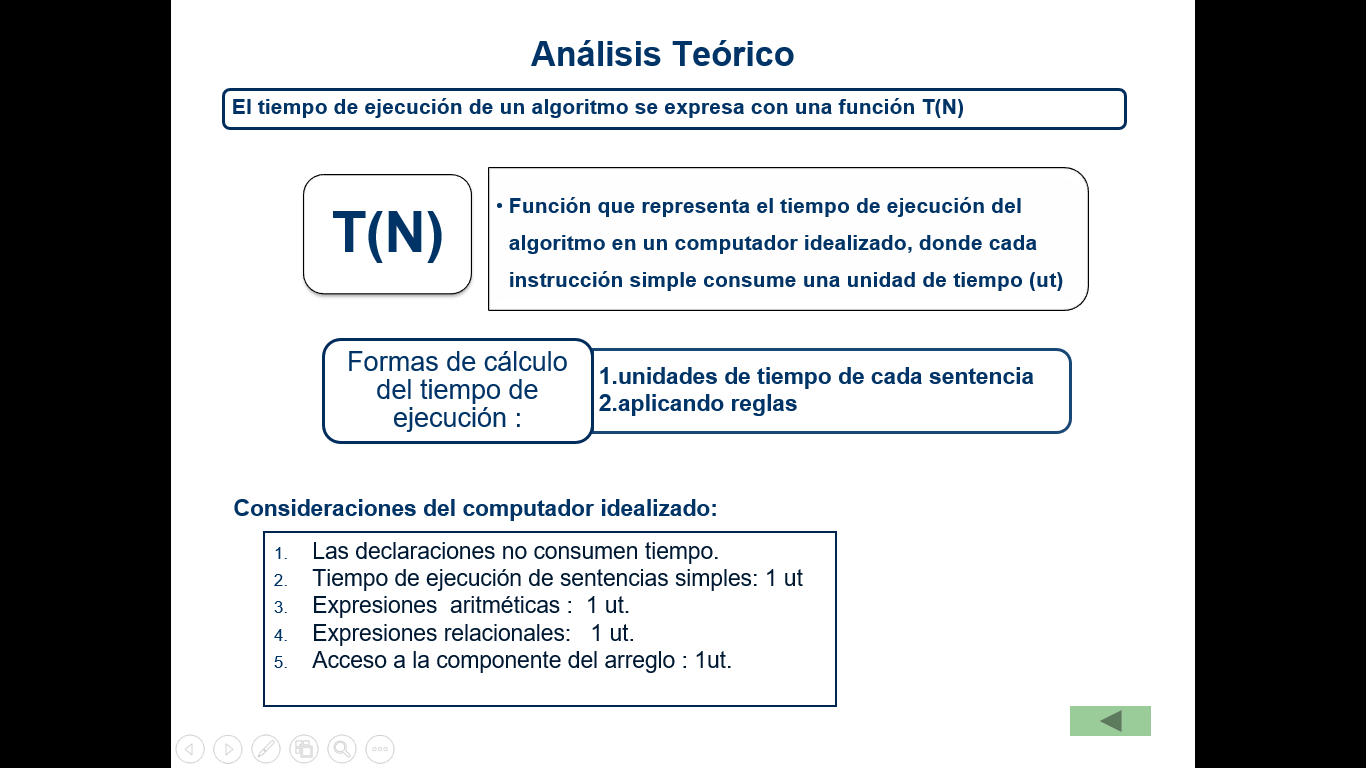
**O (1) < O (log(n))** **< O(n) < O(n.log(n)) < O(n2) < O(n3) < O(2n)**

Mientras exista la necesidad de resolver problemas cada vez más grandes, se producirá una situación casi paradójica. A medida que los computadores aumenten su rapidez y disminuyan su precio, como con toda seguridad seguirá sucediendo, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así, la importancia del descubrimiento y empleo de algoritmos eficientes, aquellos cuya velocidad de crecimiento sean pequeñas, irá en aumento en lugar de disminuir” [[5]](#footnote-5)

## Análisis teórico del tiempo de ejecución de un algoritmo

Existen dos formas de calcular el tiempo de ejecución de un algoritmo:

1. mediante el cálculo de unidades de tiempo de cada una de las sentencias que conforman un algoritmo según determinadas reglas.
2. mediante la aplicación de reglas del orden de complejidad, que optimizan el cálculo.



Cuando la definición del tiempo de ejecución de un algoritmo se realiza **en función de la cantidad de unidades de tiempo** necesarias para resolver un determinado problema, es necesario tener en cuenta las siguientes reglas

**Reglas para calcular el tiempo de ejecución de un algortimo**

* *Las declaraciones* no consumen tiempo.
* *Tiempo de ejecución de Sentencias simples*: cualquier Acción simple, de asignación, lectura o escritura consume una unidad de tiempo, éstas se conocen como operaciones elementales.
* *Las operaciones aritméticas* requieren una unidad de tiempo para su ejecución.
* El *acceso a una componente* de un arreglo involucra una unidad de tiempo.

De lo expuesto se infiere que:

tiempo de asignación a una variable de una expresión =

tiempo de evaluación de la expresión + tiempo de asignación

* *Bucles incondicionales*:

Si la cantidad de iteraciones es fija: el tiempo de ejecución se obtiene como el producto del tiempo de ejecución de las sentencias que están dentro del bloque por la cantidad de iteraciones que se debe realizar. A este tiempo debe agregarse el de inicialización de variables, testeos e incrementos de variable de control.

Tiempo ejecución bucle = tiempo del cuerpo\* número de iteraciones + tiempo de evaluar

la condición

* *Bucles condicionales*: Si la cantidad de iteraciones varía en función del valor de la variable de control, el cálculo del tiempo se expresa como una sumatoria.
* *Ciclos incondicionales anidados*: Se resuelven de adentro hacia afuera. Es decir, el tamaño de cada bloque se va multiplicando por la cantidad de iteraciones de cada ciclo. A este tiempo debe agregarse el de inicialización de variables y testeos de cada uno de los ciclos. En este caso la complejidad es polinómica.

* *Acciones alternativas : Selección doble*

Si (condición)

Entonces S1

Sino S2

Finsi

Si llamamos t1 al tiempo de ejecución de la sentencia o bloque S1 y t2 al tiempo de ejecución de la sentencia o bloque S2, el tiempo de ejecución máximo se obtiene sumando al tiempo de testeo de la condición, el tiempo máximo entre t1  y t2

T= tiempo de condición + máximo (t1, t2 ).

* *Alternativa Múltiple*

Para el caso de la alternativa múltiple, se adiciona al tiempo de evaluación de la condición el valor correspondiente al tiempo de ejecución máximo de las distintas alternativas.

Como se deduce de estas reglas, el análisis de los algoritmos se realiza desde adentro hacia afuera. Esto es, primero se determina el tiempo requerido para las instrucciones individuales y luego el que corresponde a la estructura de control que contiene a dichas instrucciones.

Con estas consideraciones, analicemos el tiempo de ejecución del siguiente segmento de código :

entero j, min

**1ut**

1. min= 0

**2n**

2. Para j desde 1 hasta n-1 1+ n + n-1 =

3. si (vector[j] < vector[min]) 3ut

**4(n-1**)

4. entonces min= j 1ut

5. Finpara

Teniendo en cuenta estas consideraciones, la estimación del tiempo de ejecución correspondiente al siguiente algoritmo es: T(N)= 1+2n+4 n-4= 6n-3

Algoritmo Envases1

Comienzo

entero Ct, Cc

real Pi, diametro, altura, sup, Vol

1 Pi= 3.14159

2 Ct=0

3 Cc=0

4 Para i Desde 1 Hasta 20000

5 Leer diametro, altura

6 radio= diametro/2

7 radioc=radio\*radio

8 sup = pi\* radioc

9 dif= sup-25

10 Si (( dif)> 0.05 ) o (( dif) < -0.05))

11 Entonces Escribir “ Tapa a eliminar”

12 Ct= Ct+1

13 Finsi

14 Vol= pi\* radioc\*altura

15 dif=Vol-50

16 Si (( dif > 0.5) o ( dif<- 0.5))

17 Entonces Escribir “ Cilindro a eliminar”

18 Cc=Cc+1

19 FinSi

20. FinPara

21 Escribir “ Detectadas ”, Ct, “ tapas defectuosas Porcentaje ”, Ct/ 200

22 Escribir “ Detectados ”, Cc, “cilindros defectuosos Porcentaje ”, Cc/ 200

Fin

Las declaraciones no consumen tiempo.

Las líneas 1 a 3 consumen una unidad de tiempo (ut) cada una.

La línea 4 correspondiente al ciclo Para, utiliza 1 unidad para la inicialización, N+1 unidades para el testeo (N es la cantidad de datos y por tanto el número de veces que se realiza el ciclo) y N unidades para el incremento de la variable del ciclo. Esto hace un total de 2\*N+2 unidades de tiempo.

Analicemos ahora las acciones que están dentro del ciclo:

La línea 5, lectura de 2 valores, utiliza 2 ut.

En las líneas 6, 7, 8 y 9 se realizan 1 operación y 1 asignación, 2ut cada una, siendo 8ut en total.

La líneas 10 a 13 correspondiente a la estructura de selección doble, requieren 6 ut.

La línea 14 realiza 2 operaciones aritméticas y 1 asignación, ocupa 3 ut .

En la línea 15 se efectúa 1 operación aritmética y 1 asignación, ocupa 2 ut.

Las líneas 16 a 19 correspondientes a la otra estructura alternativa también requieren 6 ut.

Por ello el cuerpo del ciclo requiere 27 unidades de tiempo que deberán multiplicarse por las veces en que el ciclo debe repetirse, esto es por N.

Finalmente para las Acciones 21y 22 que realizan 2 operaciones de escritura y 1 operación aritmética cada una, se necesitan en total 6ut

Resumiendo:

|  |  |  |
| --- | --- | --- |
| **Acciones** | | **Unidades de tiempo** |
| **Simples** : líneas 1 a 3 | | 3 |
| **Ciclo** | |  |
| Línea 4 | | 2\*N+2 |
| Cuerpo del ciclo: | | (27 \* N) |
|  | Líneas 5, 6, 7, 8, 9 | 10 |
| Alternativa 10 a 13 | 6 |
| Línea 14 | 3 |
| Línea 15 | 2 |
| Alternativa 16 a 19 | 6 |
| **Simples** : líneas 21 y 22 | | 6 |

*Tabla 4.3 Cálculo de unidades de tiempo*

Teniendo en cuenta que el ciclo se repite N veces, el total de unidades de tiempo para este algoritmo es

3 + (2\* N + 2) + (27 \* N) + 6 = 29\* N + 11.

Entonces se puede decir que el tiempo de ejecución de este algoritmo es de orden O(N)

Como se observa, aplicar este tipo de análisis a cada segmento de un algoritmo resulta una tarea compleja y poco práctica. Las siguientes son algunas consideraciones que pueden tenerse en cuenta, para reducir este tedioso trabajo y obtener la misma respuesta.

## Reglas para determinar el orden de complejidad

Con el fin de simplificar la tarea de calcular el orden de complejidad de los algoritmos se establecerán reglas para su determinación.

Supongamos que T1(N) y T2(N) son los tiempos de ejecución de dos segmentos o bloques de programa P1 y P2 y que T1(N) ϵ O(f(N)) y T2(n) ϵO(h(N)).

Las siguientes reglas, resultan eficaces para calcular el tiempo de ejecución de un algoritmo:

1- Regla de la Suma u Orden correspondiente a bloques consecutivos con tiempo de ejecución de distinto orden

El tiempo de ejecución de P1 seguido de P2, es decir T1(N) + T2(N), es O(max(f(N), h(N))), esto es el tiempo de ejecución total tiene un orden que coincide con el mayor de los anteriores.

Simbólicamente:

Si T1ϵO( f( N) ) y T2 ϵ O ( h( N)), entonces

|  |
| --- |
| T1 + T2 = c \* máximo ( O ( f( N) ) , O ( h( N)) ) |

*Justificación:* Esta afirmación se puede probar como sigue:

Si T1 ϵ O( f( N) ) y T2 ϵ O ( h( N) ) entonces existen las constantes c1 ,c2, N1 y N2 tales que

T1  ≤ c1 f(N) para N ≥ N1

T2 ≤ c2  h(N) para N ≥ N2

Si N0 = máximo (N1, N2) entonces T1  ≤ c1 f(N) para N ≥ N0

T2 ≤ c2 h(N) para N ≥ N0

⇒ T1  + T2 ≤ c1 f(N) + c2 h(N)para N ≥ N0

Sea c0= máximo ( c1 , c2 ) ⇒ T1  + T2 ≤ c0 f( N) ) + c0 h( N)para N ≥ N0

T1  + T2 ≤ c0 f( N) ) + c0 h( N))≤ 2c0 máximo( f( N) , h( N))

para N ≥ N0y c0= máximo ( c1 , c2 )

Llamando c a 2c0, se tiene:

T1  + T2 ≤ c máximo( f( N) , h( N))para N ≥ N0y c=2c0

***Aplicabilidad:*** La regla de la suma puede usarse para calcular el tiempo de ejecución de un algoritmo constituido por bloques. Entonces, si el algoritmo está constituido por un bloque de orden O(N), otro de orden O (N2) y un tercero de orden O(N\*logN), el tiempo completo de ejecución es de orden O(N2). El resultado está sobreestimado, por ello se habla de tiempo máximo.

O(g1(N))=**O(N),**

O(g2(N))=**O(N2),**

O(g3(N))= **O(N lgN)**

**O( max(g1(N), g2(N), g3(N) )=**O(max(N, N2, N\* lgN))= O(N2).

2 - Regla del producto O(f \* h) = O(f) \* O(h)

El orden de complejidad del producto de dos funciones es igual al producto de los órdenes de complejidad de cada una de ellas.

Justificación

Si T1 ϵO( f( N) ) y T2 ϵ O( h( N) ) entonces existen las constantes c1,c2, N1 y N2 tales que

T1  ≤ c1 f(N) para N ≥ N1

T2 ≤ c2  h(N) para N ≥N2

Entonces T1 \*T2 ≤ c1 f( N) ) \* c2 h( N))= c1 \*c2  f( N) \* h( N))

EntoncesT1 \*T2 ϵ O(f (N)\* h(N)) para N >= N0y N0= máximo (N1, N2)

**Ejemplo 4**: O((20 \* N) \* N) = O(20 \* N) \* O(N) = O(N2)

***Aplicabilidad:*** Si dos trozos de código anidados (no independientes), tienen eficiencias O(f(N)) y O(h(N)), la eficiencia del trozo completo es O(f(N)\*h(N)).

**Ejemplo 5:** dados los siguientes ciclos anidados

O(N)

O(lg N)

El orden del código completo es O(N\*lgN).

3 - O(C\*g) = O(g)

La complejidad de una función por una constante es una O de la función.

Esto es, O (2500\*N), es una O (N) , siendo C= 2500

**4 -** Los algoritmos sin lazos y sin recursión tienen complejidad constante. Se dice que son O(1) ya que representan una cantidad constante de tiempo.

**5 -** Los lazos anidados tienen complejidad polinómica.

Estas propiedades nos permiten inferir el orden de complejidad de las estructuras de control, como se muestra en la siguiente tabla:



|  |  |
| --- | --- |
| **Estructura** | **Orden** |
| Secuencial (S) | **O(1):**Si el bloque está constituido de Acciones simples de orden O(1), entonces el bloque completo será de orden O(1) por aplicación de la regla de la suma. |
| Bloques consecutivos de distinto orden S1, S2 | La **función mayor** entre **O(S1) y O(S2)** |
| Si (Condición)  entonces S1  sino S2  Finsi | La **función mayor** entre **O(S1), O(S2)** (no se considera el ordende la condiciónya que es contante) |
| Ciclo con un índice que se ejecuta n veces[S1] i = 1, n | **O(n)** |
| Ciclo con dos índices que se ejecuta n veces [S1] i = 1, nj = 1, n | **O(n2)** |

*Tabla 4.4 Orden de Complejidad para distintas estructuras de control*

**Ejemplo 6**

Analicemos el tiempo de ejecución del siguiente segmento de código

**Si** (n> m)

entonces n= n \*m

sino

Parai desde 0 hasta n-1

m= m \* n

FinPara

La sentencia que corresponde al entonces tiene orden de ejecución O(1), y el orden del sino es O(n), por tanto orden del si es O(max(1,n))= O(n)

**Ejemplo 7**

Analicemos el tiempo de ejecución de los bucles

1. Para i desde 1 hasta n

2. Para j desde 1 hasta n

3. a[i,j] = 0

4. FinPara

5. FinPara

Como se observa, los límites de los lazos indican la cota superior, ya que se refieren al número de veces exacto que se repite el ciclo.

Procediendo desde adentro hacia afuera, la línea 3 consume un tiempo del orden O(1) (3 unidades de tiempo). El lazo de la línea 2 se realiza en n ocasiones. Como su cuerpo consume un tiempo O(1) y podemos despreciar los tiempos O(1) que resulta incrementar y comparar la variable de control j, entonces el tiempo de ejecución de las líneas 2 y 3 es O(n). Un análisis idéntico es válido para el ciclo exterior que, como se realiza también n veces, nos proporciona en forma conjunta un tiempo de O(n2) para este segmento de algoritmo.

**Ejemplo 8**

Para el caso del siguiente segmento de código:

1. Para j desde **i+1** Hasta n

2. Si (a[j] > maxi)

3. Entonces maxi= a[j]

4. FinSi

5. FinPara

Como se observa, la línea 2 consume un tiempo O(1) para realizar el test y la línea 3, en caso de ejecutarse, también es de orden O(1). Por ello, como tanto el incremento, el testeo de la variable de control y el tiempo para ejecutar el cuerpo del ciclo son O(1), el tiempo total de una sola iteración es O(1).

Veamos entonces:

**el número de veces que se realiza el ciclo está dado por la fórmula: Límite superior menos límite inferior más uno**

Por ello, para el ejemplo que estamos analizando, el número de veces que se realiza el ciclo es, n - (i+1) + 1=n-i.

Por lo tanto el tiempo de ejecución de este ciclo es (n-i) \* O(1), es decir de orden O(n-i).

**Ejemplo 9**

Calculemos el orden de eficiencia del siguiente segmento de algoritmo:

1. Si ( a<b)

2. Entonces Para m desde 1 Hasta k

3. Para n desde 1 Hasta k

4. p= m\*n

5. FinPara

6. FinPara

7. Sino Para m Desde 1 Hasta k

8. s=m\*m

9. FinPara

10 .FinSi

El tiempo de ejecución de la línea 1 es O(1), el de las líneas 2 a 6 es de orden O(k2), y el de las líneas 7 a 9 es O(k). Como el peor caso se produce cuando las condición de la línea 1 es verdadera, se dice que el orden de este bloque de algoritmo es O(k2 ).

Para resolver el siguiente ejemplo, se utilizan algunas propiedades de las series cuyas demostraciones se han desarrollado en el Anexo I.

**Ejemplo 10**

Calculemos el orden de eficiencia del siguiente segmento de algoritmo que posee dos iteraciones anidadas.



Comienzo

Constante N=10

entero i, j, may, aux, A[N]

Para **i** Desde 1 Hasta N-1

may = i

Para **j** Desde **i + 1** Hasta N

Si (A[j] > A[may])

Entonces

may = j

aux = A[may]

A[may] = A[i]

A[i] =aux

FinSi

FinPara

FinPara

Fin

**Solución**

Se debe observar que si bien el bucle externo se realiza una cantidad determinada de veces(N-1), la cantidad de veces que se ejecuta el bucle interno no es constante, varía para cada valor de i. Por lo cual la cantidad de veces que se ejecutan ambos ciclos resulta de aplicar la sumatoria de la siguiente serie aritmética:

Cuando i =1 el bucle se ejecuta (N-(1+1)+1) = N-1 veces

Para i =2 el bucle se ejecuta N-2 veces

:

Para i =N -1 el bucle se ejecuta 1 vez

Entonces Cantidad total = (N-1)+(N-2)+…+1

Aplicando la propiedad de **suma de series aritméticas**, dada por la ecuación (2) del anexo 1, que expresa que la sumatoria de una serie aritmética *es igual a la suma del primer elemento más el último, multiplicado por la mitad de la cantidad de elementos de la serie,* resulta:

i = (1+(N-1) )\*(N-1)/2= N(N-1)/2

Por lo tanto, la eficiencia de este algoritmo O(N2)

**Ejemplo 11**

Analicemos la eficiencia del siguiente segmento de código que presenta un bucle donde la evolución de la variable de control es ascendente no lineal.

Constante n=100

entero c = 1, vector[n], aux

Mientras(c < n) **O(log n)**

Si(vector[c] < vector[n])

entonces aux = vector[n]

vector[n] = vector[c]

vector[c] = aux

Finsi

c = c \* 2

FinMientras

Observemos los valores de la variable c en las distintas iteraciones del ciclo:

|  |  |
| --- | --- |
| **c** |  |
| 1 | Inicio |
| 2 | Finalizar iteración 1 - 21 |
| 4 | Finalizar iteración 2- 22 |
| 8 | Finalizar iteración 3 23 |
| : |  |
| x | Finalizar iteración k 2x |

Como se observa, el valor inicial de c es 1, al finalizar la primera iteración c=21, al cabo de x iteraciones c= 2x

Si c <n luego 2*x* < n .

Aplicando log2 n a esta desigualdad se obtiene, log2 2*x* < log2 n .

Por propiedades de logaritmo: x log2 2 < log2 n

Por tanto la cantidad máxima de iteraciones x = log2 n .

Si el orden de complejidad del cuerpo del ciclo es O(1) entonces el orden de complejidad del bucle es:

O(log n) \* O(1) = **O(log n),** complejidad logarítmica.

**Actividad 1**

Calcular el orden de eficiencia del siguiente trozo de algoritmo que posee tres iteraciones anidadas. Para mayor simplicidad, no se debe considerar el tiempo de las Acciones de orden O(1).

Algoritmo Calculo

Comienzo

entero i, j, k ,c

c=0

Para i Desde 1 Hasta N-1

Para j Desde i+1 Hasta N + 1

Para k Desde 1 Hasta j

Escribir i\* j\*k

c=c+1

FinPara

FinPara

FinPara

Escribir c

Fin

**Actividad 2**

Analizar el orden de complejidad del siguiente segmento de código formado por un bucle donde la evolución de la variable de control no es lineal, junto a un bucle con evolución de variable lineal.

entero c, x, i

Para i desde 0 Hasta N-1

c = i

Mientras(c > 0)

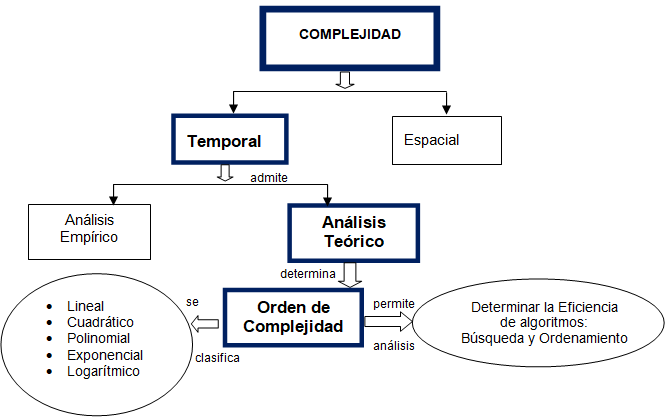
x = x / c

c = c / 2

Fin Mientras

x = x + 2

Fin Para





# 3 Eficiencia de Algoritmos de búsqueda

Los algoritmos de búsqueda son aquellos que permiten ubicar un dato en un conjunto determinado. En la unidad anterior se analizaron distintos algoritmos de búsqueda secuencial en arreglos. También se observó que el algoritmo a utilizar tiene dependencia directa de la forma de organización de los datos y que para el caso de arreglos ordenados es conveniente el uso de la búsqueda binaria.

## Eficiencia de la búsqueda lineal: peor caso

El análisis teórico debe estar basado en contabilizar la cantidad de veces que se realiza alguna acción que resulte esencial en el algoritmo. Para el caso de la búsqueda lineal, el algoritmo consiste en comparar elementos, por lo que el análisis se realizará teniendo en cuenta la relación entre el número de elementos del arreglo y la cantidad de comparaciones necesarias para encontrar un elemento dentro de él. Esto significa que se debe buscar una función que relacione las variables: número de comparaciones, (variable dependiente) y cantidad de componentes del arreglo (variable independiente).

El método de búsqueda secuencial requiere, en el peor de los casos en que el elemento buscado está al final del arreglo o no está en él, consultar los N (o N+1) elementos que lo constituyen. En este caso el tiempo de búsqueda es directamente proporcional al número de elementos del arreglo, se dice entonces que el tiempo es de orden N, lo que se simboliza TϵO(N).

Si la función de eficiencia de un algoritmo se representa como f(n) y teniendo en cuenta que el tiempo de búsqueda es directamente proporcional al número de elementos, se puede expresar

T= k \* N , siendo o < k <= 1

Por tanto el tiempo máximo de búsqueda para arreglos de N elementos, responde a la función lineal:

T O( N)

## Análisis comparativo de las tres funciones de búsqueda secuencial

Recordando que:

La Búsqueda Secuencial o Lineal consiste en recorrer y examinar cada uno de los elementos del arreglo, desde el primero, hasta encontrar el elemento buscado o hasta que se hayan examinado todos los elementos del arreglo sin éxito.

**T(N) representa** unidades de tiempo (segundos, milisegundos,...) que un algoritmo tardaría en ejecutarse con unos datos de entrada de tamaño N.

T(N) representa el tiempo de ejecución del algoritmo en un ordenador idealizado, donde cada una de las instrucciones simples consume una unidad de tiempo.

Hallemos la función T(N) para cada una de las formas de búsqueda secuencial vistas:

**Forma 1:** utilizando una bandera lógica

booleano **Bandera** (arreglo arre, entero elem)

Comienzo

entero i

booleano está

i=0 **1ut**

esta=falso **1ut**

Mientras ((i<N) y ( esta ==falso)) **3ut (***evaluación de expresión lógica***)**

Si (arre[i] ==elem) **2ut**

Entonces esta= verdadero **1ut 4ut**

Sino i=i+1 **2ut**

FinSi

FinMientras

retorna(esta) **1ut**

Fin

* En el mejor de los casos, cuando el elemento se encuentra en la primera componente del arreglo, **T(N)= 12 ut**
* En el peor de los casos, cuando el elemento no está en el arreglo, la sentencia iterativa **Mientras** se ejecuta N veces; entonces**.**

**Cuerpo del ciclo** 4\*N

**Condición** 3\*N +1 **Total** 7\*N+1

Si sumamos las asignaciones que están fuera del ciclo obtenemos: **T(N)= 7 \*N+4**

Por tanto podemos decir que el tiempo de ejecución para este algoritmo varía entre

**12 ut ≤ T(N)** ≤ **7\*N+4**  **ut**

.

El mejor caso es la cota inferior y el peor caso la cota superior

Analicemos ahora el tiempo para la forma 2 estudiada:

**Forma 2:** sin utilizar bandera lógica

entero **SinBandera** (arreglo arre, entero elem)

Comienzo

entero i

i=0 **1ut**

Mientras ((i<N) y (arre[ i] <> elem)) **4ut** *(evaluación de expresión lógic*a)

i=i+1 **2ut**

FinMientras

retorna( i ) **1ut**

Fin

* En el mejor de los casos, cuando el elemento se encuentra en la primera componente del arreglo, **T(N)= 6 ut**
* En el peor de los casos, cuando el elemento no esté en el arreglo, la sentencia iterativa

**Mientras** se ejecuta N veces; entonces:

**tiempo de ejecución del cuerpo** 2\*N

**tiempo de ejecución de condición 4\*N ut** más **1ut** de evaluar por última vez la

expresión relacional i<N; lo que da un total de  **6\*N+1 ut.**

Si sumamos todas las unidades de tiempo detalladas obtenemos: **T(N)=6\*N+3**

Por tanto podemos decir que el tiempo de ejecución para este algoritmo varía entre

6 **ut** **≤ T(N)** ≤ **6\*N+3**  **ut**

De esta manera hemos probado que la forma 2 es más eficiente que la forma 1, como habíamos inferido en la unidad anterior.

**Forma 3:** utilizando un elemento centinela.

entero **Centinela** (arreglo1 arre, entero elem)

Comienzo

entero i

i=0 **1ut**

Mientras (arre[i]<>elem) **2ut (***evaluación de expresión relacional)*

i=i+1 **2ut**

FinMientras

retorna(i) **1ut**

Fin

* En el mejor de los casos, cuando el elemento se encuentra en la primera componente del arreglo, **T(N)= 4ut**
* En el peor de los casos, cuando el elemento sea la N+1 ésima componente del arreglo, que está en la posición N , la sentencia iterativa **Mientras** se ejecuta N veces; por tanto el tiempo de ejecución es**: 4**\***N\* ut** más 2**ut** de evaluar por última vez la expresión relacional arre[i]<>elem; lo que da un total de **4\*N+2 ut.**

Si sumamos todas las unidades de tiempo detalladas obtenemos: **T(N)=4\*N+4**

Por tanto podemos decir que el tiempo de ejecución para este algoritmo varía entre

**4 ut** **≤ T(N)** ≤ **4\*N+4**  **ut**

Hemos probado que la forma 3 es más eficiente que las 2 formas anteriores.

En el siguiente cuadro comparativo analizamos las funciones T(N) obtenidas:

|  |  |  |
| --- | --- | --- |
|  | **Mejor caso** | **Peor caso** |
| **Función Bandera** | 12 ut | 7\*N+4 |
| **Función SinBandera** | 6 ut | 6\*N+3 |
| **Función Centinela** | 4 ut | 4\*N+4 |



*Tabla 4.5 Comparativo funciones T(N) Búsqueda Secuencial*

**Conclusión:** los valores del cuadro muestran que tanto en el peor como en el mejor de los casos, la función correspondiente a la búsqueda secuencial Centinela es la que proporciona menor tiempo de ejecución, para N>=2.

## Eficiencia de la Búsqueda binaria

Como se ha dicho, para arreglos de gran dimensión que estén ordenados resulta más eficiente la Búsqueda Binaria. Se determinará ahora la función matemática que muestre que este algoritmo de búsqueda es más eficiente que el de la búsqueda lineal.

Algoritmo BB

constante n=100

void **carga\_arreglo**( entero xa[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Leer xa[i]

FinPara

Fin

entero **Búsqueda\_Binaria**(entero xa[n], entero xnro)

comienzo

entero li, mi,ls

li=0

ls=n-1

mi=( li+ls) div 2

Mientras((li<=ls) y (xnro <> a[mi]))

Si ( xnro < a[mi] )

Entonces ls = mi -1

Sino li = mi+1

FinSi

mi =( li+ ls) div 2,

FinMientras

Si (li > ls)

Entonces retorna(-1)

Sino retorna(mi)

Finsi

Fin

// **Algoritmo Principal**

Comienzo

entero a[n], nro,pos

**carga\_arreglo**(a)

Leer nro

pos=**Búsqueda\_Binaria**(a, nro)

si (pos==-1)

entonces Escribir “El número ”, nro , “ no se encontró en el arreglo “

sino Escribir “El numero “, nro , “ se encontró en la posición “, pos

finsi

fin

***Análisis del peor de los casos***

Recordemos que el análisis teórico de la eficiencia de un algoritmo debe estar basado en contabilizar la cantidad de veces que se realiza alguna acción que resulte esencial en el mismo. En el caso de los algoritmos de búsqueda, la acción que se realiza es la comparación del elemento a buscar, con las distintas componentes del arreglo. Por esto, para determinar la eficiencia se analiza el número de comparaciones expresado como una función del número de elementos del arreglo.

En este algoritmo de búsqueda, con cada comparación se divide en 2 mitades el arreglo. Por lo tanto, si el tamaño del arreglo es N, los tamaños sucesivos de los subarreglos son:

N / 2 , N / 4 , N / 8 , N/16 ...

ó

N / 2 1  , N / 2 2 , N / 2 3 N/ 2 4 .........

1ª 2ª 3 ª 4ª ......

comparación comparación comparación comparación

Por lo tanto al terminar el proceso, el tamaño del subarreglo será mayor o igual a 1, se encuentre o no el elemento. Si se llama K al número de comparaciones, se verifica:

N / 2 k ≥ 1  N ≥ 2 k

Aplicando logaritmo en base 2 a ambos miembros:

log 2 N ≥ K . log 2 2  K ≤ log 2 N

Esto indica que el número de comparaciones que se realizarán para buscar un elemento en un arreglo de N componentes, hasta llegar a obtener el subarreglo de menor tamaño (1 elemento) será **a lo sumo** log 2 N.

Por lo tanto, la eficiencia de la búsqueda binaria es de orden:

|  |
| --- |
| TO( log2 N) |

## Comparación de eficiencia de los métodos de Búsqueda Secuencial y Binaria para el peor de los casos.

El siguiente gráfico permite comparar la eficiencia de ambos métodos de búsqueda estudiados.

Número de comparaciones

Tamaño del arreglo

***Búsqueda Secuencial***

**y = N**

***Búsqueda Binaria*** **y = log2 N**

*Figura 4.5 Eficiencia algoritmos de Búsqueda Secuencial Vs. Búsqueda Binaria*

Sobre el eje x se considera el tamaño del arreglo y sobre el eje y la cantidad máxima de comparaciones, para los dos métodos.

Se observa que a mayor número de componentes, más eficiente es el método de búsqueda binaria.

Así, si un arreglo tiene 1.024 elementos, la búsqueda lineal en el peor de los casos re­quiere 1.024 comparaciones, mientras que la búsqueda binaria no requerirá más de log2 1.024 = 10, es decir alrededor de 10 compa­raciones. A este tiempo deberá agregarse, si el arreglo está desor­denado, el tiempo empleado en ordenarlo.

La siguiente tabla muestra la cantidad máxima de comparaciones de ambos métodos de búsqueda para arreglos de distintos tamaños.



|  |  |  |
| --- | --- | --- |
| **N** | **B. Secuencial** | **B. Binaria** |
| 16 | 16 | 4 |
| 64 | 64 | 6 |
| 1024 | 1024 | 10 |

*Tabla 4.6 Cuadro Comparativo Eficiencia de Algoritmos de Búsqueda*

# 4 Eficiencia de los métodos de Ordenamiento

Como se dijo, ordenar un vector es una operación que permite disponer sus elementos en un orden secuencial de acuerdo con un criterio determinado, en orden ascendente o descendente.

Existen distintas técnicas de ordenación de vectores, la eficiencia de un método se determina en función de 2 criterios:

**Uso eficiente de memoria**: algunos métodos para ordenar vectores utilizan un arreglo auxiliar. Sin embargo, los que se estudiarán aquí son métodos que sólo utilizan el arreglo original, donde se intercambian los elementos que están desordenados, de modo de optimizar el almacenamiento.

**Eficiencia o economía de tiempo**: la rapidez del método depende fundamentalmente de:

* Número de comparaciones entre componentes.
* Número de movimientos o intercambios entre componentes.

En la siguiente sección se analizará la eficiencia de algunos de los algoritmos de ordenamiento más utilizados.

## Eficiencia de los Métodos de Ordenamiento por Intercambio

**Método de la burbuja mejorado ("buble sort")**

Este método de ordenamiento, que fue analizado en la unidad anterior, tiene características que se resumen como sigue.

Si A es un arreglo de dimensión N, cuyas componentes deben ser ordenadas ascendentemente, el método consiste básicamente en:

Realizar un proceso iterativo mediante una serie de pasadas, en cada una de las cuales se comparan pares de elementos adyacentes, ordenándolos si están desordenados.

Al finalizar la primera pasada el elemento mayor queda ubicado en la última posición, y en las distintas pasadas los elementos más grandes tienden a desplazarse hacia la derecha.

En cada pasada debe detectarse la posición del último cambio realizado, para que en la siguiente las comparaciones, que siempre comienzan desde el primer elemento, se realicen hasta una posición anterior al último intercambio realizado.

Si durante alguna pasada no se realizan intercambios, el algoritmo finaliza por haber quedado el arreglo ordenado.

**NOTA:** Recordar que para realizar correctamente un intercambio se debe utilizar una variable auxiliar que permita resguardar el valor de una de las variables que se compara, para finalizar produciendo el cambio requerido.

El algoritmo que responde a estos requerimientos se muestra a continuación:

Algoritmo OrdenaBurbuja\_M

constante n=10

void carga\_arreglo( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Leer xA[i]

FinPara

retorna()

Fin

void **Ordena\_Burbuja\_Mejorado**(entero xA[n])

comienzo

entero k, i , aux, cota

cota= n-1

k= 1

Mientras (k <> -1)

k= -1

Para i desde 0 hasta cota-1

Si **(xA[i] > xA [i+1])**

entonces **aux = xA [i]**

**xA [i] = xA [i+1]**

**xA [i+1] =aux**

K =i

FinSi

FinPara

cota =k

FinMientras

retorna()

Fin

void muestra\_arreglo( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Escribir xA[i]

FinPara

retorna()

Fin

// **Algoritmo Principal**

Comienzo

entero A[n]

carga\_arreglo(A)

**Ordena\_Burbuja\_Mejorado**(A)

muestra\_arreglo(A)

Fin

Se analizará la eficiencia de este método de ordenamiento para el peor de los casos, es decir cuando el arreglo esté invertido, para lo cual se considerarán los dos criterios mencionados: el número de comparaciones y el número de intercambios entre componentes.

1. **Número de comparaciones** 

|  |  |
| --- | --- |
| **Pasada** | **Comparaciones** |
| 1 | N-1 |
| 2 | N-2 |
| 3 | N-3 |
| : | : |
| N -1 | 1 |

*Tabla 4.7. Cantidad de comparaciones por pasada*

*(peor caso del método de la burbuja)*

El número total de comparaciones es entonces (N-1) + (N-2) +(N-3) + … 3 +2 +1

**Por suma de series aritméticas.( demostración en Anexo I )**

(N-1) + (N-2) +(N-3) + … 3 +2 +1= ( (N -1)+1 ) \* (N-1) /2 =N \* (N-1)/2= 1/2 (N² - N)

Su eficiencia es una función cuadrática de N, O( N² ) , lo que significa que si bien es muy sencillo es poco eficiente.

Este es el *peor de los casos*, que no es el caso más frecuente que se da en la práctica, en general los arreglos quedarán ordenados en pasadas intermedias, reduciendo de esta manera el número de comparaciones.

Para el *mejor de los casos*, cuando el arreglo esta ordenado inicialmente, este algoritmo debe realizar sólo una pasada y por tanto N -1 comparaciones.

|  |
| --- |
| (N -1)<= Comparaciones < = N\* (N -1)/2 |

**b- Número de movimientos o transferencias entre componentes**

Cuando el arreglo está ordenado, este método no realiza intercambios. En el peor de los casos, en cada comparación se intercambian dos elementos del vector, es decir se realizan tres movimientos.

Tendremos que la cantidad de movimientos es:

|  |
| --- |
| 0<= Intercambios< = 3\* N (N -1)/2 |

## Método de Selección

Para ordenar el arreglo A de N elementos en forma ascendente, primero busca el menor elemento entre los N, y lo coloca en la primera posición, busca nuevamente el menor entre los N-1 elementos restantes y los coloca en la segunda posición. Repite N –1 veces este proceso hasta colocar todos los elementos en la posición que les corresponde.

Supongamos que se quiere ordenar el siguiente arreglo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |

**Pasada 1**: El menor elemento de los 6 del arreglo es el 4 que se coloca en la primera posición

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 21 | 40 | 9 | 10 | 35 | Se intercambia a[0] con a[2] |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 2**: El menor elemento de los 5 restantes del arreglo es el 9 que se coloca en la segunda posición

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 40 | 21 | 10 | 35 | Se intercambia a[1] con a[3] |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 3**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 40 | 35 | Se intercambia a[4] con a[2] |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 4**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 40 | 35 | Se intercambia a[4] con a[5] |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

El algoritmo que trabaja el método de Selección es el que sigue:

Algoritmo Selección

Constante n=10

void **carga\_arreglo**( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Leer xA[i]

FinPara

Fin

void **Ordena\_Selección**(entero xA[n])

comienzo

entero i ,j, min, aux

Para i Desde 0 Hasta n-2

min=i

Para j Desde i+1 hasta n -1

Si (xA [ j ] <x A [min]) /\* busca el mínimo entre los elementos a[i] .....a[n-1] \*/

Entonces min=j

FinSi

FinPara

aux= xA [ i ] /\* intercambia los valores \*/

xA [ i ] = xA [min]

xA [ min] = aux

FinPara

Fin

void **muestra\_arreglo**( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Escribir xA[i]

FinPara

Fin

**// Algoritmo Principal**

Comienzo

Entero A[n]

Carga\_arreglo(A)

**Ordena\_Selección**(A)

Muestra\_arreglo(A)

Fin

Se observa que para ordenar el vector son necesarias N -1 pasadas, ya que en la última se ubican los el elementos N-1 y N-ésimo.

Se analizará la eficiencia de este método de ordenamiento para el peor de los casos, es decir cuando el arreglo esté invertido, para lo cual se considerarán los dos criterios mencionados: el número de comparaciones y el número de intercambios entre componentes.

Recordemos que la acción estructurada iterativa Para-finpara **siempre** se ejecuta la siguiente cantidad de veces: **lím sup – lím inf + 1**

**ANÁLISIS DE LA CANTIDAD DE COMPARACIONES**

Observamos que en el código existe una única comparación: (**a[j] < a[min]**), la cual se encuentra ubicada en el para-finpara de adentro.

Observamos también que **en el mejor caso** (cuando el arreglo ya está ordenado) y **en el peor caso** (cuando el arreglo está ordenado de manera inversa) **coincidirán la cantidad de comparaciones.**

Hallemos esa cantidad:

El Para-finpara de afuera se ejecuta: N-2 -0 + 1= **N-1 veces**

Veamos cuantas veces se ejecuta el Para-finpara de adentro:

Cuando i=0, j varía desde 1 hasta N-1,

entonces el para-finpara se ejecuta: N-1 -1 + 1= **N-1 veces**

Cuando i=1, j varía desde 2 hasta N-1,

entonces el para-finpara se ejecuta: N-1 -2 + 1= **N-2 veces**

Cuando i=2, j varía desde 3 hasta N-1,

entonces el para-finpara se ejecuta: N-1 -3 + 1= **N-3 veces**

……

Cuando i=N-2, j varía desde N-1 hasta N-1,

entonces el para-finpara se ejecuta: N-1 –(N-1) + 1= **1 vez**

Si **sumamos las veces** que se ejecuta el para de adentro obtenemos:

(N-1) + ( N-2) + ( N-3) + . . . + 1 = ( (N-1) + 1 ) \* (N-1)/2

Esta última igualdad se obtiene aplicando la propiedad de series aritméticas (que es el primer miembro de esta igualdad): se suman el primer y el último término (N-1) y 1; se lo multiplica por la cantidad de términos que tiene esa serie aritmética (N-1 ) que es la cantidad de veces que se ejecuta el para-finpara de afuera, y se lo divide en dos. Quedando así, la siguiente fórmula:  **N\*(N-1)/2**

Entonces , **la cantidad de Comparaciones** que se realizan **en el mejor** y **en el peor caso** es: **N\*(N-1)/2**

**ANÁLISIS DE LA CANTIDAD DE INTERCAMBIOS**

Observamos que en el código existen tres asignaciones, correspondientes a los intercambios; los cuales están ubicados en el para-finpara de afuera. Como hemos dicho que esta estructura **siempre** se ejecuta N-1 veces, entonces siempre se van a realizar **3\* (N-1) intercambios, en el mejor y en el peor caso.**

## Método de inserción directa

Sea **A** un vector de N componentes que debe ordenarse en forma ascendente. Este método consiste en insertar un elemento A[i], en el lugar que le corresponde entre los anteriores A[0]....A[i-1], que ya están ordenados .

Para ello se comienza con el segundo elemento del arreglo, si los dos primeros elementos están desordenados, los intercambia. Toma ahora el tercer elemento y busca la ubicación que le corresponde respecto de los 2 anteriores. Si A[2] es mayor o igual que A[1] significa que está en la posición correcta. En caso contrario, debe comparar A[2] con A[0]. Si A[2] es mayor o igual que A[0] lo inserta entre A[0] y A[1], pero si es menor que A[0], debe colocarlo en la posición A[0] y correr A[1] y A[2] una posición a la derecha.

En general, para insertar el elemento que está en la i-ésima posición debe compararlo con los i-1 elementos anteriores y ubicarlo en la posición que le corresponde.

Este proceso se repite N-1 veces, hasta que quedan todos en su posición correcta.

Algoritmo Inserción\_Directa

Constante n=10

void carga\_arreglo( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Leer xA[i]

FinPara

Fin

void **Ordena\_Inserción\_Directa**(entero xA[n])

comienzo

entero i ,j ,valor

Para i Desde 1 Hasta n-1

valor = xA[i]

j= i-1

Mientras (( j >= 0 ) y ( valor < xA [ j ] ))

xA [ j+1]= xA [j] /\* los valores se corren un lugar a la derecha \*/

j=j-1

FinMientras

xA[j+1] = valor

FinPara

fin

void muestra\_arreglo( entero xA[n])

comienzo

entero i

Para i Desde 0 Hasta n-1

Escribir xA[i]

FinPara

Fin

// **Algoritmo Principal**

Comienzo

entero A[n]

carga\_arreglo(A)

**Ordena\_Inserción\_Directa**(A)

muestra\_arreglo(A)

Fin

Para ordenar en forma ascendente el arreglo A:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 21 | 4 | 9 | 10 | 35 |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |

**Pasada 1**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 21 | 40 | 4 | 9 | 10 | 35 |  |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 2**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 21 | 40 | 9 | 10 | 35 |  |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 3**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 21 | 40 | 10 | 35 |  |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 4**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 40 | 35 |  |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

**Pasada 5**:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 9 | 10 | 21 | 35 | 40 |  |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |  |

Se analizará la eficiencia de este método de ordenamiento para el peor de los casos, es decir cuando el arreglo esté invertido, para lo cual se considerarán los dos criterios mencionados: el número de comparaciones y el número de intercambios entre componentes.

**ANÁLISIS DE LA CANTIDAD DE COMPARACIONES**

El Para-finpara se ejecuta: N-1 -1 + 1= **N-1 veces**

Observamos que en el código existe una única comparación: **valor < a[j]**  , la cual se encuentra ubicada en el mientras-finmientras.

**En el mejor caso** (cuando el arreglo ya está ordenado), esa comparación se realiza **N-1 veces**, aunque nunca se ejecute el cuerpo del mientras, pues siempre esa comparación da Falso.

Por lo tanto **la cantidad mínima de comparaciones es: N-1**

**En el peor caso** (cuando el arreglo está ordenado de manera inversa) debemos analizar la cantidad de veces que se ejecuta el Mientras-finmientras, para hallar la cantidad máxima de comparaciones:

Hallemos esa cantidad:

El Para-finpara de afuera se ejecuta: N-1 -1 + 1= **N-1 veces**

Cuando i=1, j varía desde 0 hasta 0, entonces el mientras-finmientras se ejecuta: **1 vez**

Cuando i=2, j varía desde 1 hasta 0, entonces el mientras-finmientras se ejecuta: **2 veces**

Cuando i=3, j varía desde 2 hasta 0, entonces el mientras-finmientras se ejecuta: **3 veces**

……

Cuando i=N-1, j varía desde N-2 hasta 0, entonces el mientras-finmientras se ejecuta: **N-1** **veces**

Si **sumamos las veces** que se ejecuta el mientras finmientras obtenemos:

1 + 2 + 3 + . . . + (N-1) = ( 1 + (N-1) ) \* (N-1)/2

Esta última igualdad se obtiene aplicando la propiedad de series aritméticas (que es el primer miembro de esta igualdad): se suman el primer y el último término 1 y (N-1); se lo multiplica por la cantidad de términos que tiene esa serie aritmética (N-1 ) que es la cantidad de veces que se ejecuta el para-finpara , y se lo divide en dos. Quedando así, la siguiente fórmula:  **N\*(N-1)/2**

Entonces, **la cantidad de Comparaciones** que se realizan **en el peor caso** es: **N\*(N-1)/2**

Concluimos en que:

**(N-1) <= Cantidad de comparaciones <= N\* (N-1)/2**

**ANÁLISIS DE LA CANTIDAD DE INTERCAMBIOS**

Observamos que en el código existen **tres** **asignaciones,** correspondientes a los intercambios:

Dos están ubicadas en el para-finpara: **Valor = a[i]** y **a[ j + 1] = valor**

Y otra ubicada dentro del mientras-finmientras: a[ j + 1] = a[ j ]

En el **mejor de los casos**, cuando el arreglo está ordenado, el mientras-finmientras **nunca** se ejecuta; por lo tanto, solo se realizan los **dos** intercambios o asignaciones ubicados en el para-finpara , estructura que se ejecuta N-1 veces. Como vemos, se realizan **2\*(N-1) intercambios.**

En el **peor de los casos**, cuando el arreglo está ordenado de manera inversa; haciendo la misma deducción de la cantidad de veces que se ejecuta el mientras-finmientras y recordando la cantidad de veces que se ejecuta el para-finpara, obtenemos la siguiente cantidad de intercambios:

( 1+2 ) + ( 2 + 2) + (3 + 2)+ . . . + ( N-1 + 2 )  **(1)**

Observamos que en cada término siempre sumamos 2, pues son dos los intercambios que se hacen en el para-finpara; y vamos sumando en cada término: 1, 2, 3, . . ., N-1 correspondientes a la cantidad de intercambios, cada vez que se ejecuta el mientras-finmientras

Volviendo a la fórmula **(1),** obtenemos: 3 + 4 + 5 + . . . + ( N+1) una serie aritmética, cuyo primer término es 3, el último término es N+1. Aplicando la misma propiedad de series aritméticas y teniendo en cuenta que la serie tiene N-1 términos (las veces que se ejecuta el para-finpara), obtenemos:

( **3+ (N+1) )\* (N-1) / 2 = (4+N )\* ( N- 1) /2**

Concluimos en que: **2\*(N-1) < = cantidad de intercambios < = (4+N )\* ( N- 1) /2**

La siguiente tabla muestra la eficiencia de cada uno de los métodos de ordenamiento analizados, para el caso de un arreglo de N componentes. Se muestran los dos criterios de eficiencia a tener en cuenta: número de comparaciones y cantidad de movimientos o intercambios realizados; para el mejor caso (el arreglo está ordenado) y el peor caso (cuando el arreglo está totalmente invertido).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Métodos** | **Cantidad de Comparaciones** | | **Cantidad de intercambios** | |
| *Mejor Caso* | *Peor caso* | *Mejor Caso* | *Peor Caso* |
| **Burbuja Mejorado** | N -1 | 1/2 (N² - N) | 0 | 3 N (N -1)/2 |
| **Selección** | 1/2 (N² - N) | 1/2 (N² - N) | 3( N -1) | 3( N -1) |
| **Inserción** | N -1 | 1/2 (N² - N) | 2( N -1) | (4+ N)(N -1) /2 |

*Tabla 4.8 Comparación teórica de eficiencia de Algoritmos de Ordenamiento*

Teniendo en cuenta el número de comparaciones, cuando el arreglo está ordenado, los algoritmos que funcionan de manera más óptima son los métodos de la burbuja y de inserción. El primero realiza un única pasada, N -1 comparaciones, detecta que el arreglo está ordenado y termina la ejecución. El método de inserción compara cada uno de los elementos con el anterior y determina que el elemento debe quedar en el lugar en que está.

El método de selección por su parte es independiente del orden de los elementos del arreglo, la cantidad de comparaciones es siempre la misma.

Atendiendo a la cantidad de movimientos, para el caso de un arreglo ordenado el método de la burbuja no realiza cambios, sin embargo los otros dos métodos sí.

El análisis realizado, muestra la eficiencia de los algoritmos para los casos extremos, que no son los más comunes en los casos reales. En la mayoría de las aplicaciones los vectores estarán organizados en forma aleatoria, para estos casos, el método de la burbuja tiene la ventaja de detener la ejecución si detecta que el arreglo ha quedado ordenado en pasadas intermedias.

**Actividad 5**:

Como se observa en la tabla anterior, en el peor caso, los métodos de inserción y de la burbuja son cuadráticos. ¿Podrías probar cual es el más óptimo e indicar para que tamaño de arreglo?

# 5 Bibliografía

* Aho Alfred V , John E. Hopcroft, Jefrey D. Ullman (1988)Estructura de datos y algoritmos versión en español de Américo Vargas y Jorge LozanoPUBLICACIÓN México, DF : Addison-Wesley. ISBN 968-6048-19-7I Iberoamericana: Sistemas Técnicos de Edición,
* Braunstein Silvia y A. Gioia (1987) Introducción a la Programación y a La Estructura de Datos. Eudeba
* Casanova Faus, A(1993).Programación. Escuela Universitaria de Informática. Universidad Politécnica de Valencia. ISBN 84-7721-233-3.
* De Giusti, Armando E, Madoz Maria y otros (2001). Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci. LIDI. Facultad de Informática Universidad Nacional de la Plata. Prentice Hall.

Páginas web consultadas

* Berzal Fernado, Análisis de algoritmos *http://elvex.ugr.es/decsai/c/apuntes/algoritmos.pdf*
* Raiger Murillo Moreno. Algoritmos de Ordenamiento:

*http://www.monografias.com/trabajos/algordenam/algordenam.sht*

* Tiempo de ejecución. Notaciones para la Eficiencia de los Algoritmos

*http://decsai.ugr.es/~jmfluna/docencia/c0304/eedd/teoria/eficiencia.pdf*

* Algoritmos computacionales *http://www.monografias.com/trabajos11/alcom/alcom.shtml*

1. Diseño y Análisis de Algoritmos Tiempo de Ejecución de un Programa. Aho Alfred Pág 16 [↑](#footnote-ref-1)
2. Extraído y adaptado de Análisis de algoritmos *http://www.ub.edu.ar/catedras/ingenieria/sist\_datos/* Fuente original D. Riley en [Ril-87]. [↑](#footnote-ref-2)
3. Extraída de Diseño y Análisis de Algoritmos. Cap 2 La eficiencia de los algoritmos. Dpto Sistemas Informáticos. Universidad de Alicante [↑](#footnote-ref-3)
4. Extraído de Estructura de datos y algoritmos Aho Alfred Pág 20 [↑](#footnote-ref-4)
5. Aho Alfred .Estructura de datos y algoritmos Pág. 20 [↑](#footnote-ref-5)