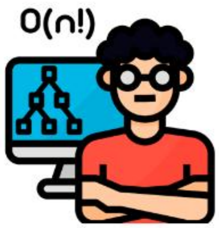
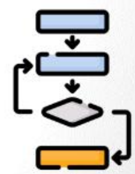
Instituto Politécnico Nacional
Escuela Superior de Cómputo



Análisis de algoritmos

Tema 02: Complejidad de los algoritmos

Prof. Edgardo Adrián Franco Martínez
http://eafranco.com
edfrancom@ipn.mx
@edfrancom edfrancom



Algoritmo

"Un algoritmo es un procedimiento para resolver un problema cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe terminar para todas las entradas"

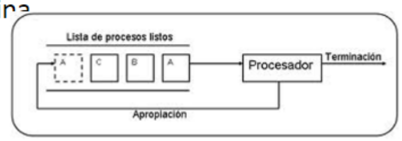
- Un **paso es NO ambiguo** cuando la acción a ejecutar está perfectamente definida:
  - $x \leftarrow log(0)$ *Ambigua*
  - $x \leftarrow log(10)+5$ *NO Ambigua*

- Una **instrucción es concreta o efectiva** cuando se puede ejecutar en un intervalo finito de tiempo
  - $x \leftarrow 2 + 8$ *Efectiva*
  - *mensaje ← Concatena('Hola','Mundo') Efectiva*
  - *x ← cardinalidad (números naturales) NO Efectiva*

Algoritmo vs. Proceso Computacional

- Si un conjunto de instrucciones a computar tiene todas las **características de un algoritmo**, **excepto ser finito en tiempo** se le denomina **proceso computacional**.

- Los sistemas operativos son el mejor ejemplo de proceso computacional, pues están diseñados para ejecutar tareas mientras las haya pendientes, y cuando éstas se terminan, el sistema operativo entra en un estado de espera, hasta que llegan más, pero nunca termina.



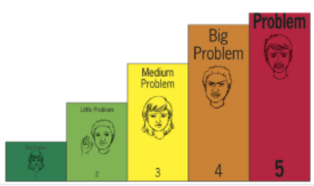
- En computación **se considera que un problema tiene solución algorítmica** si además de que el **algoritmo existe, su tiempo de ejecución es razonablemente corto**.

Tamaño de problema

"El tamaño de problema es aquella(s) característica(s) cuantificable del problema que determina la complejidad de encontrar una solución para este".

- Si el tamaño de problema aumenta este se vuelve más difícil de resolver i.e. requiere más recursos para su solución.

- **Tamaño de problema ≠ Complejidad o dificultad del problema**
- **A mayor tamaño de problema mayor complejidad o dificultad para hallar una solución; o en lo ideal complejidad constante para todos los tamaños de problema, pero nunca podría disminuir la dificultad si el tamaño de problema aumenta.**



- **Ejemplo 01:** Es posible diseñar un algoritmo para jugar ajedrez que **triunfe siempre**: *el algoritmo elige la siguiente tirada* ***examinando todas las posibles secuencias*** *de movimientos desde el tablero actual hasta uno donde sea claro el resultado y elige la tirada que le asegure el triunfo*; el pequeño inconveniente de esta algoritmo es que dicho **espacio de búsqueda se ha estimado en 1000^{40} tableros** por lo que puede tardarse años en tomar una decisión.

- Se considera **que si un problema tiene una solución que toma años en computar, dicha solución no existe.**

- **Ejemplo 02:** ordenar un conjunto de valores.

49	12	56	90	2	5	11	32	22	7	99	02	35	1

  - Si el conjunto tiene 2 elementos es más fácil resolverlo que si tiene 20, análogamente un algoritmo que resuelva el problema tardará más tiempo mientras más grande sea el conjunto y requerirá una cantidad de memoria mayor para almacenar los elementos del conjunto.

- ***"En general la cantidad de recursos que consume un algoritmo para resolver un problema se incrementa conforme crece el tamaño del problema".***

- Dependiendo del problema en particular, uno o varios de sus parámetros serán elegidos como **tamaño del problema**.

- Determinar el tamaño del problema es relativamente fácil realizando un análisis del problema si el problema ya ha sido comprendido.

PROBLEMA	TAMAÑO DEL PROBLEMA
Búsqueda de un elemento en un conjunto	**Número de elementos en el conjunto**
Multiplicar dos matrices	**Dimensión de las matrices**
Recorrer un árbol binario de búsqueda	**Número de nodos en el árbol**
Resolver un sistema de ecuaciones lineales	**Número de ecuaciones y/o incógnitas**
Ordenar un conjunto de valores	**Número de elementos en el conjunto**
Cálculo de la sumatoria $\sum_{i=m}^{n} a_i$	**Tamaño del intervalo (m,n)**
Encontrar un elemento en una Tabla Hash Abierta	**Número de elementos en la Tabla**

Función complejidad

- La *función complejidad, f(n)*; donde n **es el tamaño del problema**, da una **medida de la cantidad de recursos** que un algoritmo necesitará al implantarse y ejecutarse en alguna computadora.

- La cantidad de recursos que consume un **algoritmo crece conforme el tamaño del problema** se incrementa, la **función complejidad es** *monótona creciente* $f(n) \geq f(m)$ *si* $n > m$ con respecto al tamaño del problema.

- La **memoria y el tiempo de procesador** son los recursos sobre los cuales se concentra todo el interés en el análisis de un algoritmo, así pues se distinguen dos clases de función complejidad:

1. **Función complejidad espacial.** Mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño n: $f_e(n)$.

2. **Función complejidad temporal.** Indica la cantidad de tiempo que requiere un algoritmo para resolver un problema de tamaño n; viene a ser una medida de la cantidad de instrucciones de CPU que requiere el algoritmo para resolver un problema de tamaño n: $f_t(n)$.

- La **cantidad de memoria** que utiliza un algoritmo depende de la implementación, no obstante, es posible obtener una **medida del espacio** necesario con la sola inspección del algoritmo.

- Para obtener esta cantidad es necesario sumar todas las celdas de memoria que utiliza. En general se requerirán dos tipos de celdas de memoria:

1. **Celdas estáticas.** Son las que se utilizan en todo el tiempo que dura la ejecución del programa, p.g., las variables globales.

2. **Celdas dinámicas.** Se emplean sólo durante un momento de la ejecución, y por tanto pueden ser asignadas y devueltas conforme se ejecuta el algoritmo, p.g., el espacio de la pila utilizado por las llamadas recursivas.

Análisis de algoritmos
02 Complejidad de los algoritmos
Prof. Edgardo Adrián Franco Martínez

- **El tiempo** que emplea un algoritmo en ejecutarse refleja la cantidad de trabajo realizado, así, la **complejidad temporal** da una medida de la cantidad de tiempo que requerirá la implementación de un algoritmo para resolver el problema, por lo que **se le puede determinar en forma experimental**.

- Para encontrar el valor de la función complejidad de un algoritmo A que se codifica un lenguaje de programación L; se compila utilizando el compilador C; se ejecuta en la máquina M y se alimenta con un conjunto de casos S. Se deberá de medir el tiempo que emplea para resolver los casos (**análisis a posteriori**).

Análisis Temporal

- Medir la complejidad temporal de manera experimental presenta, entre otros, el inconveniente de que los resultados obtenidos dependen de:
  - Las entradas proporcionadas,
  - La calidad del código generado por el compilador utilizado
  - La máquina en que se hagan las pruebas

- Cada **operación** requiere cierta **cantidad constante de tiempo** para ser ejecutada, por esta razón si se cuenta el número de operaciones realizadas por el algoritmo se obtiene una **estimación del tiempo** que le tomará resolver el problema.

- Dado un algoritmo, se puede determinar que tipos de operaciones utiliza y cuantas veces las ejecuta para una entrada especifica (**análisis a priori**).

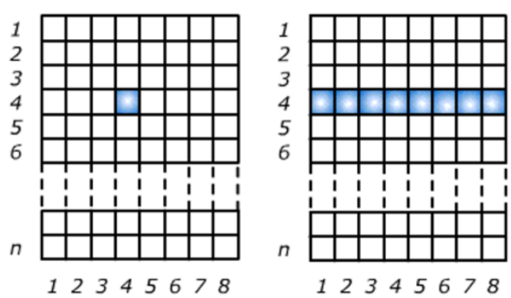
- Para evitar que **factores prácticos** se reflejen en el **cálculo de la función complejidad**, el **análisis temporal y el espacial a priori** se realiza únicamente **con base al algoritmo escrito en pseudocódigo**.

- Como el pseudocódigo no se puede ejecutar para medir la cantidad de tiempo que consume, la complejidad temporal no se expresará en unidades de tiempo, sino en términos de la **cantidad de operaciones que realiza**.

```
inserción(t)
for i=1 to n
    x= t[i]
    j=i-1
    while j > 0 and x<t[j]
       t[j+1]=t[j]
       j=j-1
    t[j+1]=x
```


Análisis Espacial

- Los casos en la función complejidad espacial, se pueden definir análogamente, considerando ahora el conjunto *C(n)*; como el conjunto formado por el número de celdas de memoria utilizadas por el algoritmo para resolver cada instancia del problema.



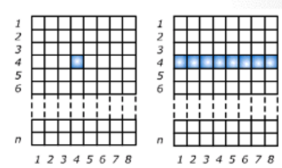
Medición del tiempo de ejecución

- **Medir**
  - **Cantidad de instrucciones** básicas (o elementales) que se ejecutan.
    - Ejemplos de instrucciones básicas:
      1. Asignación de variables
      2. Lectura o escritura de variables
      3. Saltos (goto's) implícitos o explícitos.
      4. Operaciones aritméticas
      5. Evaluación de condiciones
      6. Llamada a sentencias simples
      7. Llamadas y retornos de función*

**Las funciones tienen un costo de tiempo que hay que considerar evaluando particularmente cada una si no se conocen.*

Medición de la memoria requerida

- **Medir**
  - **Cantidad de celdas de memoria** (o elementales) que se requieren.
    - Ejemplos de celdas de memoria:
      1. Variables del algoritmo
      2. Numero de objetos instanciados requeridos
      3. Tamaño de las estructuras de datos empleadas
      4. Memoria de Entrada/Salida requerida
      5. Tamaño de arreglo, matrices u otro tipo de memoria continua estática o dinámica empleada por el algoritmo.



Ejemplo 1: Cantidad de instrucciones

Considerando el conteo de ASIGNACIONES, ARITMETICAS, SALTOS y CONDICIONALES

```
cont = 1;                →1 asignación
do {
   x = x + a[cont];      →n asignaciones + n operaciones aritméticas
   x = x + b[cont];      →n asignaciones + n operaciones aritméticas
   cont = cont + 1;      →n asignaciones + n operaciones aritméticas
}
while (cont <= n)        →n (comparaciones) + n-1 (goto implícito *true)
                         →1 (goto implícito *falso)
```

TOTAL: **8n + 1 instrucciones**

Función de complejidad temporal
f(n)=8n+1

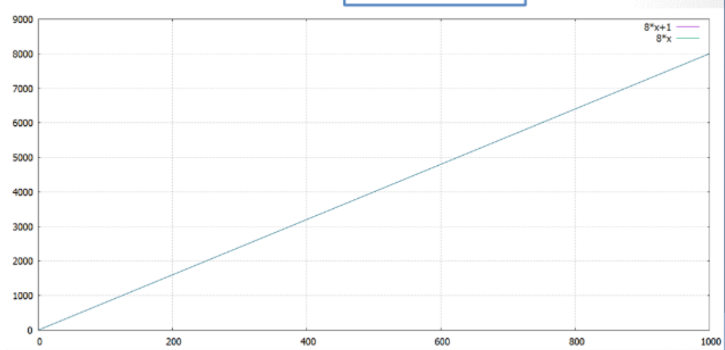
Función de complejidad temporal

f(n)= 8n + 1

n	Instrucciones
0	1
1	9
10	81
100	801
500	4001
1000	8001

```
gnuplot> set grid
gnuplot> set autoscale
gnuplot> plot 8*x+1,8*x
gnuplot> plot [0:100] 8*x+1,8*x
gnuplot> plot [0:1000] 8*x+1,8*x
```

```
cont = 1;
do {
  x = x + a[cont];
  x = x + b[cont];
  cont = cont + 1;
}
while (cont <= n)
```



Ejemplo 1: Cantidad de celdas de memoria

```
cont = 1;              →1 variable “cont”
do {
   x = x + a[cont];    →1 variable “x”
   x = x + b[cont];    →n variables del arreglo “a”
   cont = cont + 1;    →n variables del arreglo “b”
}
while (cont <= n)
```

TOTAL: 2n + 2

Función de complejidad espacial
$f(n)=2n+2$

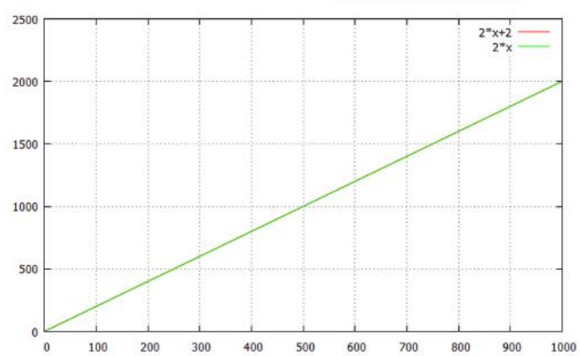
****No se cuenta a n como VARIABLE, estrictamente debería ser 2n+3***

Función de complejidad espacial

$f(n)= 2n + 2$

```
cont = 1;
do {
   x = x + a[cont];
   x = x + b[cont];
   cont = cont + 1;
}
while (cont <= n)
```

n	Celdas de memoria
0	2
1	4
10	22
100	202
500	1002
1000	2002



Análisis de algoritmos
02 Complejidad de los algoritmos
Prof. Edgardo Adrián Franco Martínez

Ejemplo 2: Cantidad de instrucciones

Considerando el conteo de ASIGNACIONES, ARITMETICAS, SALTOS y CONDICIONALES

```
z = 0;
for (int x=1; x<=n; x++)
  for (int y=1; y<=n; y++)
    z = z + a[x,y];
```

→ 1 = 1

→ 1+ (n+1) (asignación + comparaciones) + n (aritméticas)= $2+2n$

→ (1+(n+1) + n)*n (veces de ejecución del for interno) = $2n^2 + 2n$

→ (n+n)(asignación + aritméticas) *n = $2n^2$

→ n (goto implícito en falso for y) = n

→ n*n (goto implícito en true for y) = n^2

→ 1 (goto implícito en falso for x) = 1

→ n (goto implícito en true for x) = n

→ = $1+2+2n+2n^2+2n+2n^2+ n+ n^2+1+n$

TOTAL: $5n^2 + 6n + 4$

Función de complejidad temporal

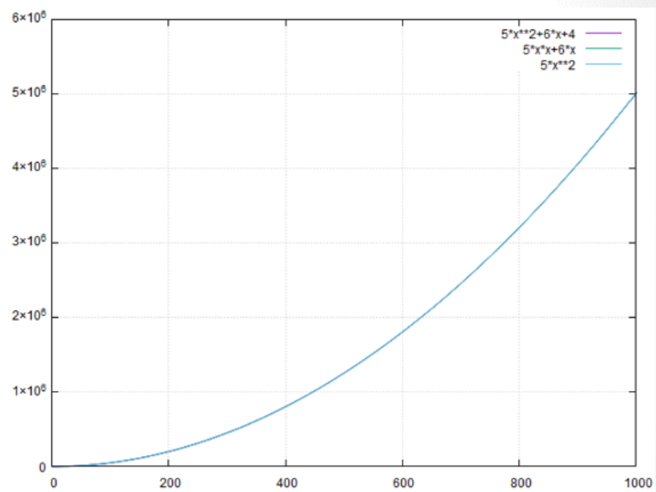
$f(n)=5n^2 + 6n + 4$

Función de complejidad temporal

```
z = 0;
for (int x=1; x<=n; x++)
  for (int y=1; y<=n; y++)
    z = z + a[x,y];
```

$f(n)= 5n^2 + 6n + 4$

n	Instrucciones
0	5
1	15
10	1,104
100	51,704
500	1,251,304
1000	5,006,004



Ejemplo 2: Cantidad de celdas de memoria

```
z = 0;                          → 1 (variable "z")
for (int x=1; x<=n; x++)        → 1 (variable "x")
 for (int y=1; y<=n; y++)       → 1 (variable "y")
    z = z + a[x,y];             → n*n (variables de la matriz "a")
```

TOTAL: $n^2 + 3$

Función de complejidad temporal
$f(n)=n^2 + 3$

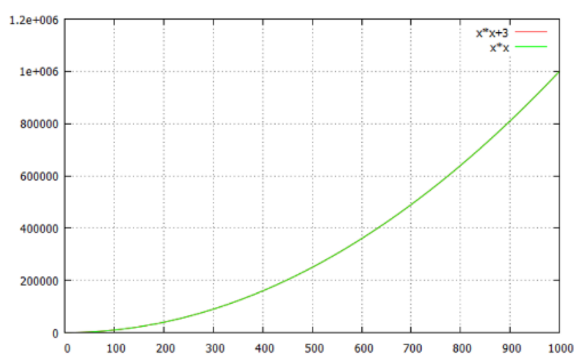
***No se cuenta a n como VARIABLE, estrictamente debería ser n^2+4**

Función de complejidad espacial

```
z = 0;
for (int x=1; x<=n; x++)
 for (int y=1; y<=n; y++)
    z = z + a[x,y];
```

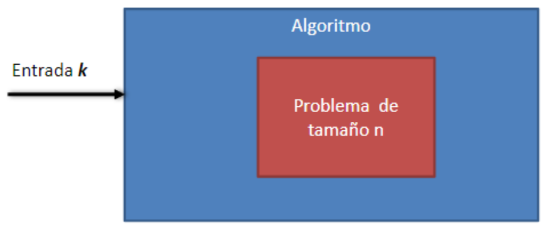
$f(n)=n^2 + 3$

n	Celdas de memoria
0	3
1	4
10	103
100	10,003
500	250,003
1000	1,000,003



- Los algoritmos de los ejemplos 1 y 2 tienen un tamaño de problema directamente asociados a ***n i.e. solo existe un caso (instancia) del problema***.

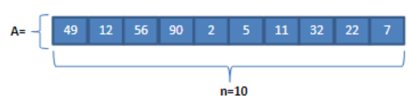
- En la mayoría de los algoritmos también se deberá de considera que el número de operaciones y celdas memoria dependerá de los **casos de entrada** por lo que debe de realizarse el análisis considerando cada caso ***K (instancia del problema)*** con el tamaño de problema ***n.***
  - i.e. en un algoritmo, se debe determinar que tipos de operaciones utiliza, cuantas veces las ejecuta y cuanta memoria requiere para cada entrada especifica ***k***.



Ejemplo 3: Cantidad de instrucciones

Considerando el conteo de ASIGNACIONES, ARITMETICAS, ACCESOS A A[] y CONDICIONALES

```
func BusquedaLineal(Valor,A,n)
{
        i=1;
        while(i<=n&&A[i]!=Valor)
        {
                i=i+1;
        }
        return i;
}
```



Caso 0: Valor=11, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor es 11 $\Rightarrow$ se entra al ciclo 6 veces $\Rightarrow$ k=6

Si el ciclo se ejecutará k veces (k puede tomar valor de 1 hasta n)

→k sumas (una por cada iteración).

→k + 2 asignaciones (las del ciclo y las realizadas fuera del ciclo).

→k + 1 operaciones lógicas (la condición se debe probar k + 1 veces, la última es para saber que el ciclo no debe volver a ejecutarse).

→k + 1 comparaciones con el índice.

→k + 1 comparaciones con elementos de A:

→k + 1 accesos a elementos de A:

→6k + 6 operaciones en total.

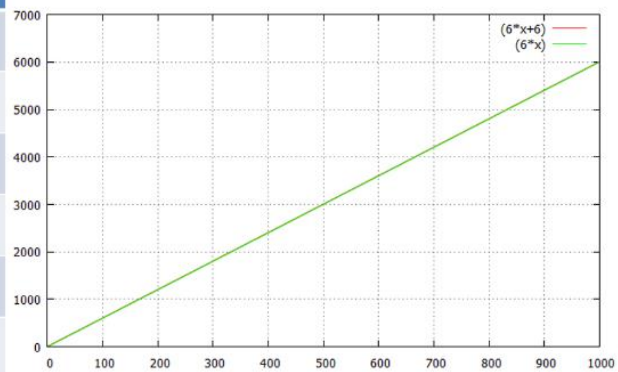
Función de complejidad temporal

→f(k)=6k + 6

→k=Numero de veces que se entra al ciclo

```
func BusquedaLineal(Valor,A,n)
{
        i=1;
        while(i<=n&&A[i]!=Valor)
        {
                i=i+1;
        }
        return i;
}
```

k	Instrucciones
0	6
1	12
10	66
100	606
500	3006
1000	6006



Ejemplo 3: Cantidad de celdas de memoria

A= | 49 | 12 | 56 | 90 | 2 | 5 | 11 | 32 | 22 | 7 |

n=10

```
func BusquedaLineal(Valor,A,n)
{
        i=1;
        while(i<=n&&A[i]!=Valor)
        {
                i=i+1;
        }
        return i;
}
```

Caso 0: Valor=11, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor=11 $\Rightarrow$ *se requieren* $10 + 3$ *variables*

Caso 1: Valor=12, A={49,12,56,90,2,5,11,32,22,7}, n=10

Si el Valor=12 $\Rightarrow$ *se requieren* $10 + 3$ *variables*

Si se analizaran todos los casos

→Para este algoritmo la complejidad espacial no varia

→n + 3 celdas de memoria en total.

Función de complejidad espacial constante para todos los casos

→f(n)= n + 3

k	Celdas de memoria
0	3
1	4
10	13
100	103
500	503
1000	1003

```
func BusquedaLineal(Valor,A,n)
{
        i=1;
        while(i<=n&&A[i]!=Valor)
        {
                i=i+1;
        }
        return i;
}
```