Análisis de Algoritmos y Estructura de Datos

Complejidad de algoritmos

Prof. Violeta Chang C

Semestre 2 – 2023

DEPARTAMENTO DE
INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE SANTIAGO DE CHILE

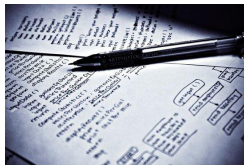
Complejidad de algoritmos

- **Contenidos**:
  - Eficacia vs eficiencia de un algoritmo
  - Análisis de complejidad
  - Complejidad de algoritmos iterativos
  - Complejidad de algoritmos recursivos
- **Objetivos**:
  - Describir conceptos de complejidad algorítmica
  - Explicar mecanismos de cálculo de complejidad de algoritmos iterativos y recursivos
  - Calcular complejidad de algoritmos iterativos y recursivos

Ruta de la sesión

- Eficiencia VS eficacia
- Complejidad algorítmica
- Cálculo de complejidad de algoritmos iterativos
- Cálculo de complejidad de algoritmos recursivos

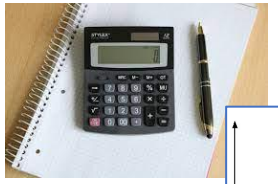
Contexto

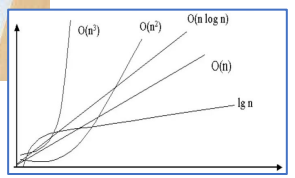


Solución 1 → seudocódigo 1 → Implementación1

Solución 2 → seudocódigo 2 → Implementación 2

Solución 3 → seudocódigo 3 → Implementación 3



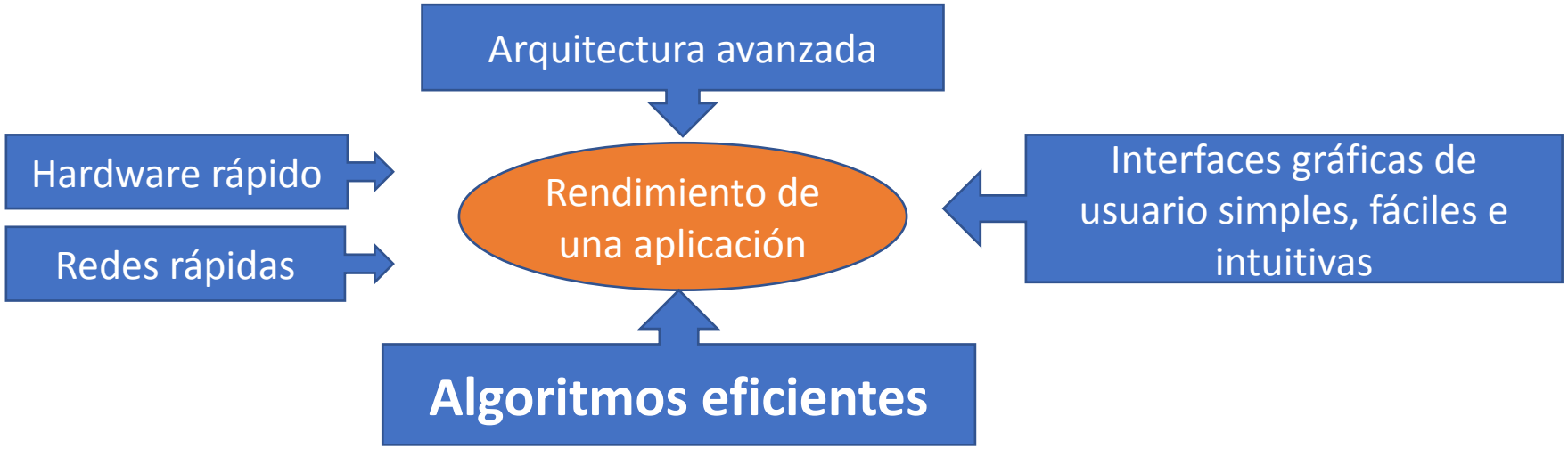


Eficiencia de un algoritmo

- Características a considerar al construir algoritmos
  - Eficacia
    - RAE:
      "Capacidad de lograr el efecto que se desea o se espera."
    - El algoritmo hace correctamente lo que debe hacer
  - Eficiencia
    - RAE:
      "Capacidad de disponer de alguien o de algo para conseguir un efecto determinado"
    - El algoritmo usa la menor cantidad de recursos para cumplir con su tarea

¿Por qué construir algoritmos eficientes?

- El tiempo de ejecución y el espacio en memoria son recursos limitados.



Eficiencia de un algoritmo

- ¿Cómo se puede medir la eficiencia de un algoritmo?
  - Tiempo
  - Número de instrucciones
  - Cantidad de memoria usada
  - “Dificultad de un conjunto de instrucciones”
- ¿Es el tiempo una buena medida para medir la eficiencia de un algoritmo? ¿Qué factores influyen en este tiempo?

Eficiencia de un algoritmo

¿Se pueden comparar?

- Dados dos algoritmos A y B que resuelven un mismo problema
  - **Ambos son eficaces**
  - Los algoritmos fueron ejecutados en **dos máquinas distintas** y A fue más rápido que B.
  - Los algoritmos fueron ejecutados en la misma máquina y A (**en C**) fue más rápido que B (**en JAVA**)
  - Los algoritmos fueron ejecutados en la misma máquina y en el mismo lenguaje, pero por dos **programadores distintos**, y A fue más rápido que B
  - Los algoritmos fueron ejecutados en la misma máquina, en el mismo lenguaje, por el mismo programador, pero usando dos **compiladores distintos** y A fue más rápido que B.
  - Todo lo anterior igual (máquina, lenguaje, programador, compilador, etc, etc, etc) pero usando dos **instancias del problema distintas**, y A fue más rápido que B.

Eficiencia de un algoritmo

- Contabilizaremos la cantidad de instrucciones/pasos que realiza un algoritmo para dar una solución a una instancia del problema
- Para n = 5, ¿cuántas instrucciones se realizan en cada algoritmo?
- ¿y si n aumenta?

```
Alg1(num n)
acum ← 0
PARA i ← 0 HASTA n
    acum ← acum + i
ESCRIBIR(acum)
```

```
Alg3(num n)
acum ← (n*(n+1))/2
ESCRIBIR(acum)
```

Contando instrucciones: Ejemplo 1

- Sumar los elementos de un arreglo A de n posiciones

```
sumaArreglo(arreglo A): num
1. acum ← 0
2. n ← largoArreglo(A)
3. para i← 1 hasta n
4.    acum ← acum + A[i]
5. devolver(acum)
```

Contando instrucciones: Ejemplo 1

- Sumar los elementos de un arreglo A de n posiciones

```
sumaArreglo(arreglo A): num
1. acum ← 0                          1 instrucción
2. n ← largoArreglo(A)               1 instrucción
3. para i← 1 hasta n
4.    acum ← acum+A[i]               2 instrucciones
5. devolver(acum)                    1 instrucción
```

```
Total instrucciones = 2 + n*2 + 1
                    = 3 + 2*n
```

Contando instrucciones: Ejemplo 2

- Multiplicar dos matrices cuadradas de orden n

```
multMat(matriz A, matriz B): matriz
1. acum ← 0
2. n ← numFilas(A) ...numColumnas(A) o numFilas(B) o numColumnas(B)
3. para i← 1 hasta n
4.     para j← 1 hasta n
5.         acum ← 0
6.         para k ← 1 hasta n
7.             acum ← acum + A[i][k]*B[k][j]
8.         C[i][j] ← acum
9. devolver( C )
```

Contando instrucciones: Ejemplo 2

- Multiplicar dos matrices de orden n

```
multMat(matriz A, matriz B): matriz
1.  acum ← 0                                          1 instrucción
2.  n ← numFilas(A)                                   1 instrucción
3.  para i← 1 hasta n
4.      para j← 1 hasta n
5.          acum ← 0                                  1 instrucción
6.          para k← 1 hasta n
7.              acum ← acum + A[i][k]*B[k][j]         3 instrucciones
8.          C[i][j] ← acum                            1 instrucción
9.  devolver( C )                                     1 instrucción
```

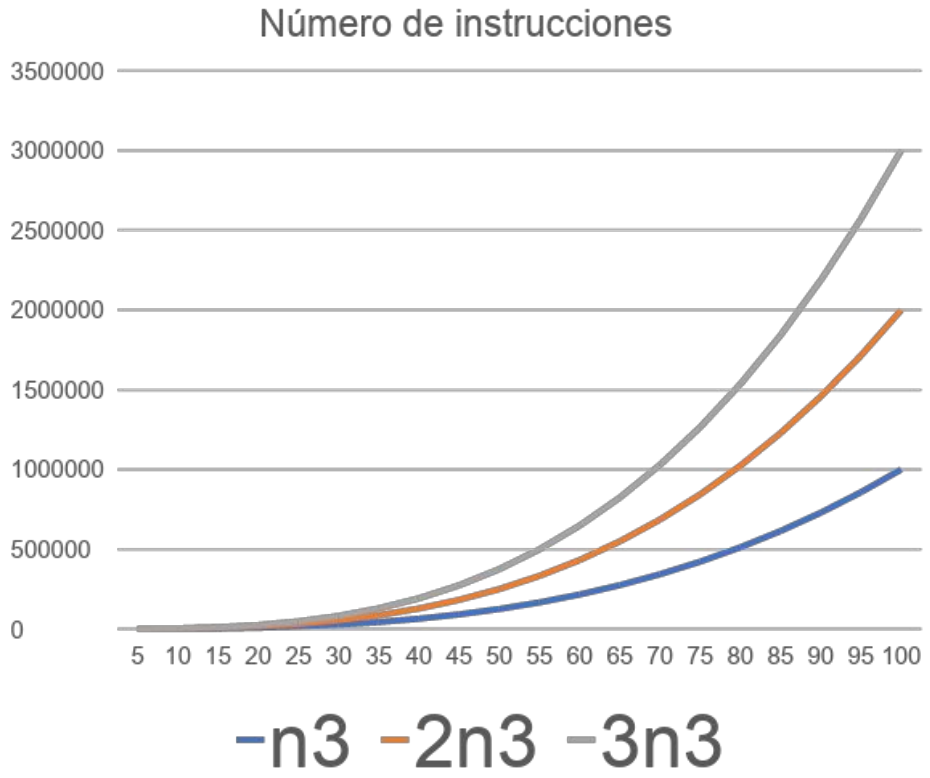
Multiplicación, suma y asignación

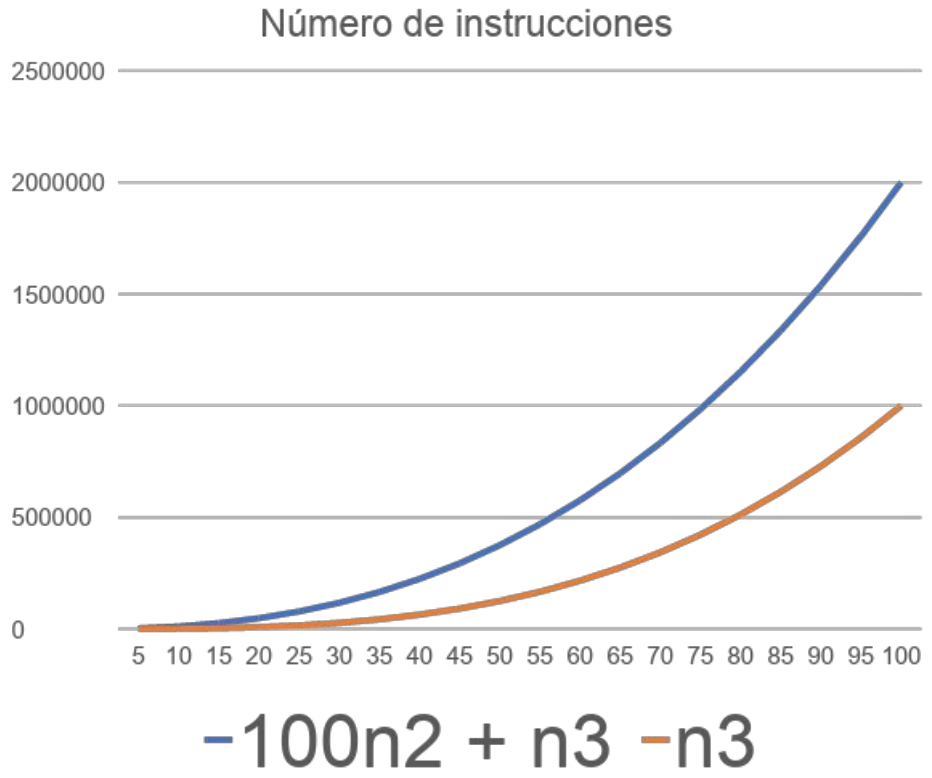
$$\text{Total instrucciones} = 2 + n*(n*(1 + n*3 + 1)) + 1$$
$$= 3 + 2n^2 + 3n^3$$

Contando instrucciones

Cantidad de operaciones que se deben realizar, dependiendo de algún parámetro de entrada

Crecimiento del tiempo de ejecución de un algoritmo con respecto a alguna variable de entrada





La unidad de T(n) (“tiempo” de un algoritmo) puede ser pensada en términos de número de pasos a ser ejecutados en un computador ideal

Complejidad : Computador ideal

- El análisis de realiza sobre una máquina hipotética ideal (RAM: Random Access Machine)
  - Las instrucciones se ejecutan de manera secuencial
  - No hay operaciones concurrentes
  - Cada operación tiene un costo asociado
- Esta definición simplifica el análisis y permite estimar adecuadamente la complejidad de tiempo de un algoritmo

Complejidad algorítmica

- Se entiende como la cantidad de recursos necesarios para ejecutar un algoritmo
  - Tiempo (pasos): **complejidad de tiempo**
  - Memoria: complejidad de espacio
- Complejidad de tiempo:
  - cómo crece el tiempo de ejecución del algoritmo con respecto tamaño n de la entrada: **función en términos del tamaño de la entrada n -> f(n)**
  - **Tiempo de un algoritmo**: **T(n)** representa el número de pasos a ser ejecutados en un computador ideal
  - **Orden de complejidad de un algoritmo**: **O()** representa la cota superior de T(n)

Orden de complejidad

- Análisis del peor caso:
  - Se considera a *T(n)* como el peor caso de una ejecución, el tiempo máximo que puede alcanzar para una entrada *n*
  - Entrega una función del tiempo que puede ser muy pesimista
- Análisis del caso promedio:
  - $T_{avg}(n)$ entrega el tiempo promedio sobre todos los valores de n
  - Difícil de calcular
  - Se puede estimar sobre una muestra de datos, pero puede ser costoso
- El análisis se realizará siempre pensando en el peor caso, salvo que se indique algo diferente

Cálculo de complejidad

- Todo algoritmo posee un nombre, entradas (y salidas)

  NOMBRE_ALGORITMO(entradas**)**

  NOMBRE_ALGORITMO(entradas**): SALIDA**

  ⟶ $T_{alg}(n)$

- Tipos de instrucciones
  - Asignación: ←
  - Comparación: **=**, **<>**, **>**, **<**, **>=**, **<=**
  - Comentarios: **…**
  - Aritméticas: **+**, **-**, ***, **/** , **MÓDULO**
  - Lógicas: **Y** , **O** , **NO**
  - Input/Output:

    LEER(valor**)**

    ESCRIBIR(valor**)**

  ⟶ c

Cálculo de complejidad

- Instrucciones de control

SI condición **ENTONCES**

InstruccionesV

SINO

InstruccionesF

$$T_{IF} = T_{cond} + \max(T_V, T_F)$$

Cálculo de complejidad

- Instrucciones de control

PARA varControl← valorInicio **HASTA** valorFin
    Instrucciones

MIENTRAS condición **HACER**
    Instrucciones

$$\longrightarrow \quad T_{CICLO} = \sum_{inicio}^{fin} (Tco_{nd} + Tin_s)$$

Cálculo de complejidad

- **Retorno**:

  DEVOLVER(valor**)** ⟶ c

- **Llamada a procedimiento:**

  [variable ←] **algoritmoX (**ent1**,** ent2**, …)** ⟶ T_{algX}

Cálculo de complejidad: ejemplo

- Tomando el ejemplo particular de la suma de elementos de un arreglo

```
sumaArreglo(arreglo A): num
        acum ← 0
        n← largoArreglo(A)
        para i← 1 hasta n
                acum← acum+A[i]
        devolver(acum)
```

Cálculo de complejidad: ejemplo

- Tomando el ejemplo particular de la suma de elementos de un arreglo

```
sumaArreglo(arreglo A): num
    acum ← 0                          c
    n← largoArreglo(A)                c
    para i← 1 hasta n                 3c (suma, asignación, comparación)
        acum← acum+A[i]                   2c (suma, asignación)
    devolver(acum)                    c
```

} *n* iteraciones

Cálculo de complejidad: ejemplo

- Tomando el ejemplo particular de la suma de elementos de un arreglo

```
sumaArreglo(arreglo A): num
    acum ← 0                          c
    n← largoArreglo(A)                c
    para i← 1 hasta n                 3c (suma, asignación, comparación)
        acum← acum+A[i]                   2c (suma, asignación)   } n iteraciones
    devolver(acum)                    c
```

$$T(n) = 2c + \sum(3c + 2c) + c$$

$$T(n) = 3c + 5cn$$

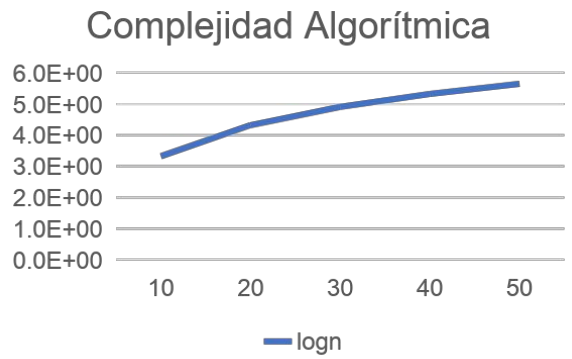
$$T(n) \leq 8cn \ldots \text{por lo tanto } O(n)$$

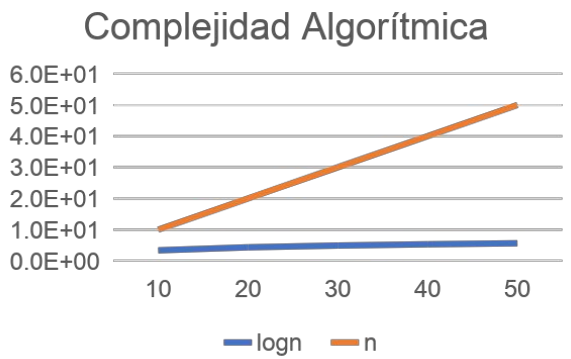
Tipos de orden de complejidad

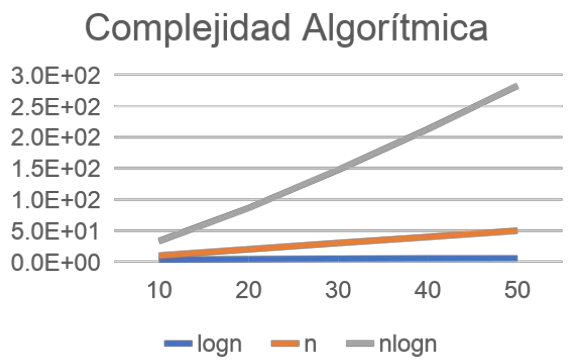
La función de O(f(n)) indica qué tan bueno es un algoritmo para distintos tamaños de entrada

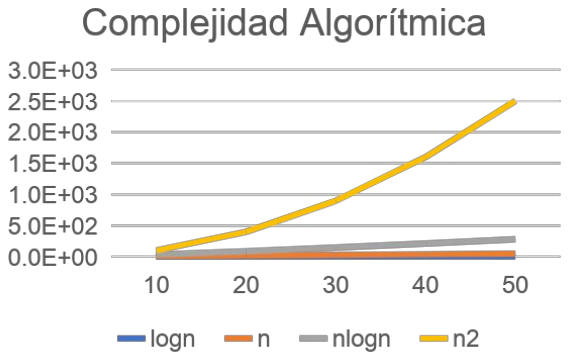
Tiempo	Orden	Tipo
$T(n)=c$	O(1)	Constante
$T(n)=c \log_2 n$	$O(\log_2 n)$	Orden logarítmico
$T(n)=c\ n$	$O(n)$	Orden lineal
$T(n)=c\ n^2$	$O(n^2)$	Orden cuadrático
$T(n)=c\ n^3$	$O(n^3)$	Orden cúbico
$T(n)=c\ 2^n$	$O(2^n)$	Orden exponencial
$T(n)=c\ n!$	$O(n!)$	Orden factorial

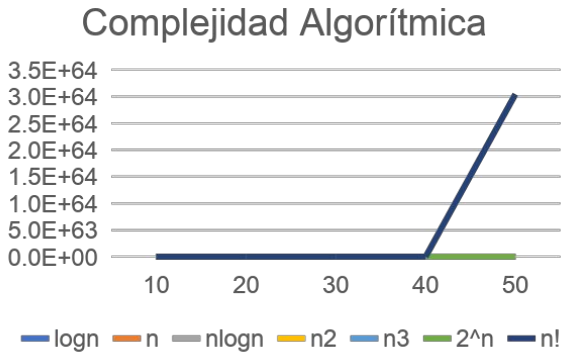
Análisis de algoritmos

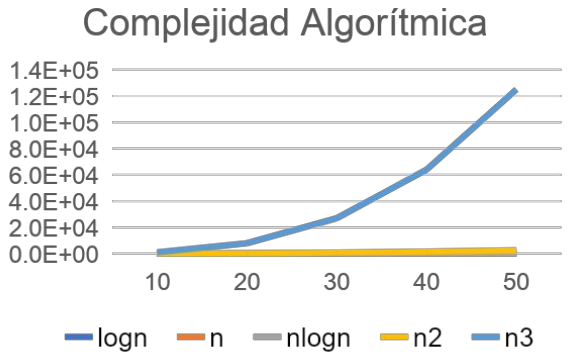


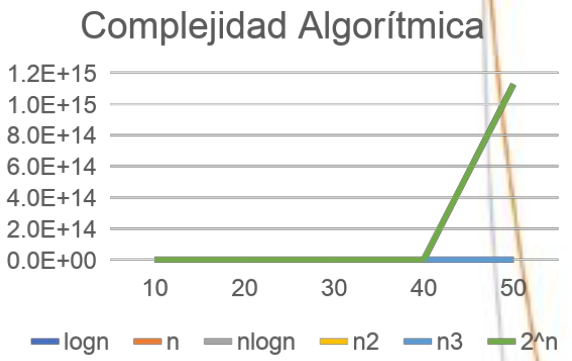












Complejidad de algoritmos iterativos

- Se debe conocer el tiempo de ejecución teórico de cada instrucción (modelo RAM teórico)
- Anotar la suma de todos los tiempos (T(n))
- Resolver y acotar la función (f(n))
- Determinar orden de complejidad O(*f(n)*)

Complejidad de algoritmos recursivos

- Un algoritmo recursivo posee la siguiente estructura general

```
algRec(entrada n): salida
...
si condición(n) entonces
    devolver(valor)
...
valor ⟵ algRec(entrada modificada)
devolver(valor)
```

→ Condición de término

→ Llamada recursiva

Complejidad de algoritmos recursivos

- La complejidad de estos algoritmos se expresa como una ecuación de recurrencia de la forma:

$$T(n) = \begin{cases} c & cond\ término \\ T(n\ modificado) + c, & otro\ caso \end{cases}$$

Complejidad de algoritmos recursivos

- La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & n > 1 \\ 1 & n = 0 \end{cases}$$

Complejidad de algoritmos recursivos

- La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

Complejidad de algoritmos recursivos

- La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

- La ecuación de recurrencia para el tiempo de ejecución es:

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

- La ecuación de recurrencia para el tiempo de ejecución es:

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

- La ecuación de recurrencia para el tiempo de ejecución es:

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

Complejidad algoritmos recursivos

- Solución por reducción:
  - Reducción por sustracción:
    - se realizan ***a*** llamadas recursivas
    - tamaño del problema se reduce una cantidad constante **b** en cada llamada
    - Operaciones parte no recursiva tienen ***O(n^k)***

- $T(n) = aT(n-b) + O(n^k), si\ n \geq b$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

Complejidad algoritmos recursivos

- Solución por reducción:
  - Reducción por división:
    - se realizan ***a*** llamadas recursivas
    - tamaño del problema se reduce en una proporción b (n/b) en cada llamada
    - Operaciones parte no recursiva tienen $O(n^k)$
    - $T(n) = aT\left(\frac{n}{b}\right) + O(n^k), si\ n \geq b$

$$O(\ ) = \begin{cases} n^k & si\ a < b^k \\ n^k \log n & si\ a = b^k \\ n^{\log_b a} & si\ a > b^k \end{cases}$$

b

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

T(n)= 2c n=1
4c+T(n-1) otro caso

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

T(n)= 2c n=1
4c+T(n-1) otro caso

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

T(n)= 2c n=1
4c+T(n-1) otro caso

*T(n)= 1*T(n-1)+O(1)*

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\ \ ) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

T(n)= 2c n=1
4c+T(n-1) otro caso

*T(n)= 1*T(n-1)+O(1)*

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\ \ ) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

a=1
b=1
k=0

Complejidad de algoritmos recursivos

- Factorial recursivo:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & si\ n > 1 \\ 1 & si\ n \leq 1 \end{cases}$$

```
factorial(num n):num
si n = 1 entonces
   devolver(1)
valor← n*factorial(n-1)
devolver(valor)
```

$$T(n) = \begin{cases} c_1, & si\ n \leq 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

T(n)= 2c n=1
4c+T(n-1) otro caso

*T(n)= 1*T(n-1)+O(1)*

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\ \ ) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

a=1
b=1
k=0

∴

$O(n^{k+1}) = O(n^{0+1}) = O(n)$

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + T(n-2) & n > 1 \\ c & n \leq 1 \end{cases}$$

$T(n) = \begin{cases} 4c & n=0,1 \\ 6c+T(n-1)+T(n-2) & \text{otro caso} \end{cases}$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + T(n-2) & n > 1 \\ c & n \leq 1 \end{cases}$$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

T(n)= 4c n=0,1
6c+T(n-1)+T(n-2) otro caso

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + T(n-2) & n > 1 \\ c & n \leq 1 \end{cases}$$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

T(n)= 4c n=0,1
6c+T(n-1)+T(n-2) otro caso

*T(n)≤ 2*T(n-1)+O(1)*

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + T(n-2) & n > 1 \\ c & n \leq 1 \end{cases}$$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

T(n)= 4c n=0,1
6c+T(n-1)+T(n-2) otro caso

*T(n)≤ 2*T(n-1)+O(1)*

$$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

a=2
b=1
k=0

Complejidad de algoritmos recursivos

- Fibonacci recursivo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + T(n-2) & n > 1 \\ c & n \leq 1 \end{cases}$$

```
fib(num n):num
si n=1 O n=0 entonces
   devolver(n)
valor←fib(n-1)+fib(n-2)
devolver(valor)
```

$T(n) = aT(n-b) + O(n^k) \text{ si } n \geq b$

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

a=2
b=1
k=0

T(n)= 4c n=0,1
6c+T(n-1)+T(n-2) otro caso

*T(n)≤ 2*T(n-1)+O(1)*

$\therefore O(a^{n/b}) = O(2^{n/1}) = O(2^n)$

Clases de problemas

- Un problema **P** es aquel que tiene un algoritmo de orden polinómico que lo resuelve.
- Un problema **NP** tiene un algoritmo de orden polinómico que lo verifica.
- Un problema es **NP-Completo** si es imposible encontrar un algoritmo eficiente para encontrar una solución óptima … hasta ahora…
- Los **algoritmos de fuerza bruta** son capaces de encontrar la solución a cualquier problema por complicado que sea. Prueban todas las posibles combinaciones hasta dar con la situación que es igual que la solución… no son eficientes

Resumen

- En análisis de complejidad en tiempo de un algoritmo entrega la tasa de crecimiento del tiempo de respuesta a medida que aumenta el tamaño de la entrada
- Se “ejecuta” sobre una “máquina ideal”
- Se debe definir el tiempo de cada tipo de instrucción
- Técnicas de cálculo para algoritmos iterativos y recursivos
- Algoritmos recursivos
  - Reducción por sustracción
  - Reducción por división

Actividad de cierre

- Ir a menti.com e ingresar código 8702 9916

DEPARTAMENTO DE
INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE SANTIAGO DE CHILE

Próximas fechas...

- Resumen de la semana:
  - Complejidad de algoritmos iterativos
  - Complejidad de algoritmos recursivos

~~cátedra~~ – refuerzo – laboratorio

- Próxima semana:
  - Algoritmos de búsqueda y ordenamiento

U1 - S2