# Análisis de Algoritmos
# Unidad 1: Conceptos de Análisis

Ing. Matias Valdenegro T.

Universidad Tecnológica Metropolitana

28 de diciembre de 2011

# Presentación

- Profesor: Matias Valdenegro T.
- Ingeniero Civil en Computación mención Informática (2009).
- Actualmente trabajo en el Laboratorio de Modelamiento Matemático para Geomecanica del Centro de Modelamiento Matemático, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile.
- Pueden contactarme en matias.valdenegro@gmail.com

# Contenidos del Curso

El contenido del curso esta compuesto de:

- 1.- Elementos de Analisis.
- 2.- Algoritmos y Problemas.
- 3.- Ordenamiento.
- 4.- El paradigma “divide-y-venceras”.
- 5.- Clasificacion de Problemas.

# Evaluacion

La evaluacion de la asignatura es:

- Prueba 1 con un 35 %.
- Prueba 2 con un 35 %.
- Promedio de notas de Tareas (6), con un 30 %.

# Bibliografia

## Basica


- Algoritmos Computacionales: Introduccion al Analisis y Diseño. Sara Baase, Allen Van Gelder. Pearson Educacion. ISBN 970-26-1042-8. (Disponible en Biblioteca).


## Complementaria


- Introduction to Algorithms, Second Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. MIT Press. ISBN 978-0-262-03293-3.
- **Papers y artículos del área, seleccionados por el profesor**.

# Que es el Análisis de Algoritmos?

## Análisis

Un análisis en sentido amplio es la descomposición de un todo en partes para poder estudiar su estructura y/o funciones.

## Análisis de Algoritmos

El análisis de algoritmos provee estimaciones teóricas para los recursos necesarios para ejecutar un algoritmo. Generalmente los recursos considerados son espaciales (Memoria RAM y Almacenamiento) y tiempo de ejecución.

# Porque?

- Comparar algoritmos.
- Determinar el mejor algoritmo para resolver un problema dado.
- Determinar si es posible resolver un problema dadas ciertas restricciones de recursos.

# Ejemplo

Se desea ordenar una base de datos que contiene datos de personas, usando el RUT como indicador de orden. La base de datos contiene $10^8$ registros. Entonces:

1. Que algoritmo de ordenamiento se deberia usar?
2. Que requerimientos (cantidad) de memoria se requieren para ordenar dicha cantidad de datos?
3. Es posible ejecutar ese trabajo en un computador con un procesador de 2 GHz? Cuanto tiempo tomara (aproximadamente)?

# Algoritmo

## Definición

Un algoritmo es un procedimiento paso-a-paso para resolver un problema en una cantidad de tiempo finita.

## Definición Alternativa

Un algoritmo es una función que transforma un objeto entrada en un objeto salida.

## Nota Importante

En general, los algoritmos se usan para resolver problemas.

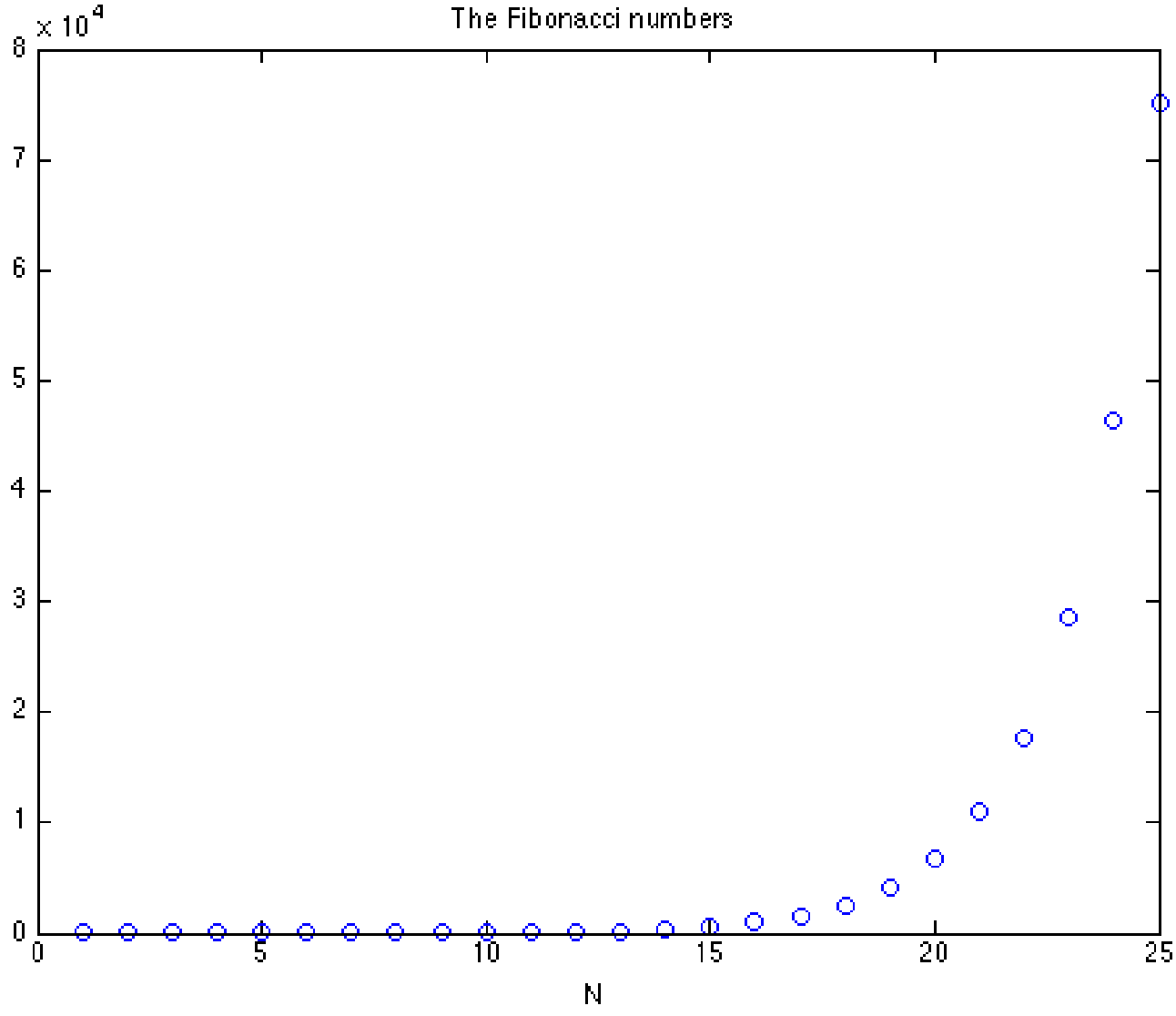
# Ejemplo simple

Dado el siguiente algoritmo:

```
int f(int n)
{
    if(n == 0 || n == 1) {
        return 1;
    } else {
        return f(n - 1) + f(n - 1);
    }
}
```

Es posible determinar (estimar) cuanto tiempo de computo tomara f(10)? f(100)? f(n)?

# Ejemplo simple: Gráfico



# Caracterización del Tiempo de Ejecución

La pregunta es: De que depende el Tiempo de Ejecución de un Algoritmo?

- Cuanto tiempo toma ordenar un arreglo de 10, 100 y 1000 elementos?
- Cuanto tiempo toma ordenar un arreglo con elementos aleatorios? Cuanto tiempo toma ordenar un arreglo ya ordenado?

# Caracterización del Tiempo de Ejecución

La características que mas influyen en el Tiempo de Ejecución son:

- Del tamaño de la entrada (N) del algoritmo.
  - Largo del arreglo de entrada
  - Numero de nodos de una lista
  - Altura del árbol de entrada.
- De características de los valores de la entrada.
  - Datos ordenados.
  - Datos con patrones no deseados.

# Caracterización del Tiempo de Ejecución

El tiempo de ejecución crece con el tamaño de la entrada.
Existen varios casos para el tiempo de ejecución:

- Mejor caso (Best case).
- Caso promedio. (Average case).
- Peor caso. (Worst case).

# Complejidad Algorítmica

## Definición

Es una medida de los recursos necesarios usados por la ejecución de un algoritmo.

## Recursos

- Tiempo (de computo, de CPU).
- Espacio (Cantidad de memoria RAM, etc).

# Complejidad Empírica o Experimental

## Definición

La complejidad experimental es la que se obtiene al realizar experimentos y medir el tiempo de ejecución del programa.

## Procedimiento

- Escribir una implementación del algoritmo.
- Ejecutar el programa, variando el tamaño de la entrada.
- Medir los tiempos de ejecución. (gettimeofday(), time(), etc).
- Presentar los resultados.

# Limitaciones

- Es necesario implementar el algoritmo, lo que puede ser difícil.
- Si el algoritmo es teórico, se dificulta su análisis.
- Los resultados pueden ser no representativos.
- Es posible usar datos que no representen la realidad.
- Para comparar algoritmos, se debe usar el mismo entorno de HW y SW.

# Análisis de Código

Otra forma de obtener una estimación del tiempo de ejecución es contar el numero de operaciones primitivas que realiza un algoritmo.

## Operaciones

- Operaciones básicas y/o primitivas realizadas por el algoritmo.
- Independientes del lenguaje de programación.
- Identificables en el código fuente.

El tiempo de ejecución queda expresado en el numero de operaciones.

# Análisis de Código

## Operaciones Primitivas

- Asignación de variables, Acceso a arreglos.
- Operaciones aritméticas, Comparación.
- Llamadas a funciones.
- Operaciones de ramas (If, Else, etc).
- Desreferenciar punteros, acceder a miembros.

# Costo de Operaciones Primitivas

## Operaciones de Costo 1

Asignación, Indexado de Arreglos, Operaciones aritméticas, Comparación, Desreferenciar punteros, acceso a miembros.

## For, While, Do-While

Cuestan el producto del numero de veces que se ejecutan con el costo del código al interior de estos, mas el costo de evaluar la condición.

## If, Else

Cuestan el máximo entre el costo de cada rama, mas el costo de evaluar la condición.

## Llamadas a Funciones

Cuestan el costo de la función por cada llamada/invocación. Pero si es recursiva, se deben usar recurrencias.

# Ejemplos

```
A = 2 + 3;
B = A - 3;
C = A / B;
D = sqrt(C);
```

# Ejemplos

```
for(int i = 0; i < n; i++) {
    A[i] = i + 1;
}
```

# Ejemplos

```
for(int i = 0; i < n; i++) {
    if(A[i] > A[mayor]) {
        mayor = i;
    }
}
```

# Ejemplos

```
for(int i = 0; i < n; i++) {
    for(int j = i; j < n; j++) {
        B[i] += A[i] + A[j];
    }
}
```

# Ejemplos

```
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < n; k++) {
            A[i][j] += A[i][k] * A[k][j];
        }
    }
}
```

# Ejemplos

```
for(int i = 0; i < n; i++) {
    if(A[i] > 1) {
        A[i]++;
    } else {
        A[i] = A[i-1] + 2;
    }
}
```

# Ejemplos

```
i = 0, j = 0;

while(i + j < 10) {
    i++;
    j++;

    a = i - j;
}
```

# Ejemplos

```
Nodo *p = <inicio lista enlazada>;
do {
    p = p->next;
} while(p != NULL);
```

# Complejidad del mejor caso

## Definición Formal

Sea $D_n$ el conjunto de entradas de tamaño n para el problema y sea $I$ un elemento de $D_n$. Sea $t(I)$ la complejidad del algoritmo cuando la entrada es $I$. Entonces:

$$W(n) = \text{mín}\{t(I) | I \in D_n\}$$

# Complejidad del peor caso

## Definición Formal

Sea $D_n$ el conjunto de entradas de tamaño n para el problema y sea $I$ un elemento de $D_n$. Sea $t(I)$ la complejidad del algoritmo cuando la entrada es $I$. Entonces:

$$W(n) = \text{máx}\{t(I) | I \in D_n\}$$

# Complejidad del caso promedio

## Definición Formal

Sea $P(I)$ [1] la probabilidad de que se presente la entrada $I$.
Entonces:

$$A(n) = \sum_{I \in D_n} P(I)t(I)$$

[1] $\sum P(I) = 1$

# Ecuaciones de Recurrencia

Es posible analizar el siguiente código?

```
int fib(int n)
{
    if(n == 0) {
        return 0;
    } else if(n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Cuantas operaciones toma ejecutar este algoritmo?

# Solución

Para analizar algoritmos recursivos, se deben resolver ecuaciones de recurrencia. En el caso de Fibonacci:

$$T(n) = T(n-1) + T(n-2)$$

La pregunta es entonces, como resolver dicha ecuación?

- Sustitución hacia atrás.
- Casos especiales.
- Teorema Maestro (el cual se vera en la Unidad 3).

# Sustitución Hacia Atrás

- Consiste en reemplazar iterativamente la recurrencia.
- Se buscan patrones que indiquen una posible solución.
- Una vez que se encuentra un patrón, se reemplaza una condición de borde para encontrar la solución.

# Ejemplo

Analicemos una búsqueda lineal en un arreglo, pero de forma recursiva.

```
void buscar(int i, int *arreglo, int elemento)
{
    if(i > 0) {
        if(arreglo[i] == elemento) {
            /* Encontrado! */
        }

        buscar(i - 1, arreglo);
    }
}
```

En Análisis de Código produce la siguiente ecuación de recurrencia:

# Ejemplo

Analicemos una búsqueda lineal en un arreglo, pero de forma recursiva.

```
void buscar(int i, int *arreglo, int elemento)
{
    if(i > 0) {
        if(arreglo[i] == elemento) {
            /* Encontrado! */
        }

        buscar(i - 1, arreglo);
    }
}
```

En Análisis de Código produce la siguiente ecuación de recurrencia:

$$T(n) = T(n-1) + c$$

$$T(0) = c_0$$

# Ejemplo

Entonces:

$$T(n) = T(n-1) + c$$

$$T(n-1) = T(n-2) + c$$

# Ejemplo

Entonces:

$$T(n) = T(n-1) + c$$

$$T(n-1) = T(n-2) + c$$

$$T(n) = T(n-2) + c + c = T(n-2) + 2c$$

$$T(n-2) = T(n-3) + c$$

$$T(n) = T(n-3) + c + c + c = T(n-3) + 3c$$

# Ejemplo

Entonces:

$$T(n) = T(n-1) + c$$

$$T(n-1) = T(n-2) + c$$

$$T(n) = T(n-2) + c + c = T(n-2) + 2c$$

$$T(n-2) = T(n-3) + c$$

$$T(n) = T(n-3) + c + c + c = T(n-3) + 3c$$

$$T(n-3) = T(n-4) + c$$

$$T(n) = T(n-4) + c + c + c + c = T(n-4) + 4c$$

# Ejemplo

$$T(n) = T(n-1) + 1c$$

# Ejemplo

$$T(n) = T(n-1) + 1c$$
$$T(n) = T(n-2) + 2c$$

# Ejemplo

$$T(n) = T(n-1) + 1c$$
$$T(n) = T(n-2) + 2c$$
$$T(n) = T(n-3) + 3c$$

# Ejemplo

$$T(n) = T(n-1) + 1c$$
$$T(n) = T(n-2) + 2c$$
$$T(n) = T(n-3) + 3c$$
$$T(n) = T(n-4) + 4c$$
$$T(n) = T(n-5) + 5c$$

# Ejemplo

$$
\begin{aligned}
T(n) &= T(n-1) + 1c \\
T(n) &= T(n-2) + 2c \\
T(n) &= T(n-3) + 3c \\
T(n) &= T(n-4) + 4c \\
T(n) &= T(n-5) + 5c \\
T(n) &= T(n-k) + kc
\end{aligned}
$$

# Ejemplo

$$T(n) = T(n-1) + 1c$$
$$T(n) = T(n-2) + 2c$$
$$T(n) = T(n-3) + 3c$$
$$T(n) = T(n-4) + 4c$$
$$T(n) = T(n-5) + 5c$$
$$T(n) = T(n-k) + kc$$

Como conocemos $T(0)$, si asumimos $n = k$:

$$T(n) = T(n-n) + nc = T(0) + nc = c_0 + cn$$

Entonces la solución final es:

# Ejemplo

$$T(n) = T(n-1) + 1c$$
$$T(n) = T(n-2) + 2c$$
$$T(n) = T(n-3) + 3c$$
$$T(n) = T(n-4) + 4c$$
$$T(n) = T(n-5) + 5c$$
$$T(n) = T(n-k) + kc$$

Como conocemos $T(0)$, si asumimos $n = k$:

$$T(n) = T(n-n) + nc = T(0) + nc = c_0 + cn$$

Entonces la solución final es:

$$T(n) = cn + c_0$$

# Ejemplo

```
int busquedaBinaria(int i, int f, int *a, int e)
{
    if(i > f) {
        return -1;
    }

    int m = (i + f) / 2;

    if(a[m] > e) {
        return busquedaBinaria(i, m - 1, a, e);
    } else if(a[m] < e) {
        return busquedaBinaria(m + 1, f, a, e);
    } else {
        return m; /* Encontrado! */
    }
}
```

# Ejemplo

Cual es la ecuación de recurrencia que define el tiempo de ejecución de la búsqueda binaria?

# Ejemplo

Cual es la ecuación de recurrencia que define el tiempo de ejecución de la búsqueda binaria?

$$T(n) = T(n/2) + c$$

y la condición de borde es:

$$T(1) = c_0$$

# Ejemplo

Cual es la ecuación de recurrencia que define el tiempo de ejecución de la búsqueda binaria?

$$T(n) = T(n/2) + c$$

y la condición de borde es:

$$T(1) = c_0$$

Hacemos sustitución hacia atrás:
$T(n) = T(n/2) + c$
$T(n/2) = T(n/4) + c$

# Ejemplo

Cual es la ecuación de recurrencia que define el tiempo de ejecución de la búsqueda binaria?

$$T(n) = T(n/2) + c$$

y la condición de borde es:

$$T(1) = c_0$$

Hacemos sustitución hacia atrás:
$T(n) = T(n/2) + c$
$T(n/2) = T(n/4) + c$
Entonces:
$T(n) = T(n/4) + c + c = T(n/4) + 2c$

# Ejemplo

$$T(n) = T(n/2) + 1c$$

# Ejemplo

$$T(n) = T(n/2) + 1c$$
$$T(n) = T(n/4) + 2c$$

# Ejemplo

$$T(n) = T(n/2) + 1c$$
$$T(n) = T(n/4) + 2c$$
$$T(n) = T(n/8) + 3c$$

# Ejemplo

$$T(n) = T(n/2) + 1c$$
$$T(n) = T(n/4) + 2c$$
$$T(n) = T(n/8) + 3c$$
$$T(n) = T(n/16) + 4c$$
$$T(n) = T(n/32) + 5c$$

# Ejemplo

$$T(n) = T(n/2) + 1c$$
$$T(n) = T(n/4) + 2c$$
$$T(n) = T(n/8) + 3c$$
$$T(n) = T(n/16) + 4c$$
$$T(n) = T(n/32) + 5c$$
$$T(n) = T(n/2^k) + kc$$

# Ejemplo

$T(n) = T(n/2) + 1c$
$T(n) = T(n/4) + 2c$
$T(n) = T(n/8) + 3c$
$T(n) = T(n/16) + 4c$
$T(n) = T(n/32) + 5c$
$T(n) = T(n/2^k) + kc$

Aplicando la condición de borde $T(1) = c_0$:

$$n/2^k = 1 \rightarrow$$

# Ejemplo

$T(n) = T(n/2) + 1c$
$T(n) = T(n/4) + 2c$
$T(n) = T(n/8) + 3c$
$T(n) = T(n/16) + 4c$
$T(n) = T(n/32) + 5c$
$T(n) = T(n/2^k) + kc$

Aplicando la condición de borde $T(1) = c_0$:

$$n/2^k = 1 \rightarrow k = \log_2 n$$

Entonces:

# Ejemplo

$T(n) = T(n/2) + 1c$
$T(n) = T(n/4) + 2c$
$T(n) = T(n/8) + 3c$
$T(n) = T(n/16) + 4c$
$T(n) = T(n/32) + 5c$
$T(n) = T(n/2^k) + kc$

Aplicando la condición de borde $T(1) = c_0$:

$$n/2^k = 1 \rightarrow k = \log_2 n$$

Entonces:

$$T(n) = T(n/2^k) + kc = T(n/2^{\log_2 n}) + c \log_2 n$$

# Ejemplo

$T(n) = T(n/2) + 1c$
$T(n) = T(n/4) + 2c$
$T(n) = T(n/8) + 3c$
$T(n) = T(n/16) + 4c$
$T(n) = T(n/32) + 5c$
$T(n) = T(n/2^k) + kc$

Aplicando la condición de borde $T(1) = c_0$:

$$n/2^k = 1 \rightarrow k = \log_2 n$$

Entonces:

$$T(n) = T(n/2^k) + kc = T(n/2^{\log_2 n}) + c \log_2 n$$

$$T(n) = T(1) + c \log_2 n = T(n) = c_0 + c \log_2 n$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

$$T(n) = 2T(n/2) + n$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

$$T(n) = 2T(n/2) + n \rightarrow T(n) = n \log_2 n$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

$$T(n) = 2T(n/2) + n \rightarrow T(n) = n \log_2 n$$

$$T(n) = 2T(n-1) + 1$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

$$T(n) = 2T(n/2) + n \rightarrow T(n) = n \log_2 n$$

$$T(n) = 2T(n-1) + 1 \rightarrow T(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$T(n) = 2T(n/2) + 1$$

# Ejercicios

Resuelva las siguientes ecuaciones de recurrencia:

$$T(n) = T(n-1) + n \rightarrow T(n) = \frac{n(n-1)}{2}$$

$$T(n) = 2T(n/2) + n \rightarrow T(n) = n \log_2 n$$

$$T(n) = 2T(n-1) + 1 \rightarrow T(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$T(n) = 2T(n/2) + 1 \rightarrow T(n) = n$$

# Casos Especiales

Si tenemos una ecuación de recurrencia de la forma:

$$T(n) = AT(n-1) + BT(n-2)$$

Esta ecuación se denomina **ecuación de recurrencia lineal homogénea**.
La solución depende de la naturaleza de las raíces de la llamada "ecuación característica:

$$s^2 - As - B = 0$$

Existen 3 casos posibles para las raíces de la ecuación de 2do grado.

# Ecuaciones Lineales Homogéneas

2 raíces distintas, $s_1$ y $s_1$

$$a_n = \alpha s_1^n + \beta s_2^n$$

2 raíces iguales, $s$

$$a_n = \alpha s^n + \beta n s^n$$

Raíces complejas conjugadas $s_1 = r\angle\theta$ y $s_2 = r\angle -\theta$

$$a_n = r^n(\alpha \cos n\theta + \beta \sin n\theta)$$

# Ejemplo

Volvamos a la recurrencia de Fibonacci:

$$T(n) = T(n-1) + T(n-2)$$

La ecuación característica es:

$$s^2 - s - 1 = 0$$

Y las raíces de la ecuación característica son:

$$r_1 = \frac{1+\sqrt{5}}{2}, \qquad r_2 = \frac{1-\sqrt{5}}{2}$$

# Ejemplo

Entonces la solución a la recurrencia es:

# Ejemplo

Entonces la solución a la recurrencia es:

$$T(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

# Ejemplo

Entonces la solución a la recurrencia es:

$$T(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Como encontramos los valores de $\alpha$ y $\beta$? Usando las condiciones iniciales:

$$T(0) = 0, \qquad T(1) = 1$$

Resolviendo el sistema de ecuaciones para $\alpha$ y $\beta$:

# Ejemplo

Entonces la solución a la recurrencia es:

$$T(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Como encontramos los valores de $\alpha$ y $\beta$? Usando las condiciones iniciales:

$$T(0) = 0, \qquad T(1) = 1$$

Resolviendo el sistema de ecuaciones para $\alpha$ y $\beta$:

$$\alpha = \frac{1}{\sqrt{5}}, \qquad \beta = -\frac{1}{\sqrt{5}}$$

# Ejemplo

Entonces, la solución de la ecuación de recurrencia es:

$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

# Análisis Asintótico

| $\mu s$ | $33n$ | $46n \log n$ | $13n^2$ | $3{,}4n^3$ | $2^n$ |
|---|---|---|---|---|---|
| n | | | | | |
| 10 | 0.00033s | 0.015s | 0.0013 s | 0.0034 s | 0.001 s |
| 100 | 0.003 s | 0.03 s | 0.13 s | 3.4 s | $4 \cdot 10^{16}$ años |
| 1000 | 0.033 s | 0.45 s | 13 s | 0.94 h | |
| 10000 | 0.33 s | 6.1 s | 22 min | 39 dias | |
| 100000 | 3.3 s | 1.3 min | 1.5 dias | 108 años | |

# Análisis Asintótico

| $\mu s$ | $33n$ | $46n \log n$ | $13n^2$ | $3{,}4n^3$ | $2^n$ |
|---|---|---|---|---|---|
| Tiempo max | Tamaño de entrada máximo soluble | | | | |
| 1 s | 30000 | 2000 | 280 | 67 | 20 |
| 1 min | 1800000 | 82000 | 2200 | 260 | 26 |

# Análisis Asintótico

- En general, importa la estructura del algoritmo por sobre micro-optimizaciones.
- Si se tiene un computador $x$ veces mas rápido que otro, esto no implica que se pueden resolver problemas $x$ veces mas grandes.
- Las constantes en el tiempo de ejecución no importan mucho a menos que las entradas sean pequeñas.
- Tampoco importa si la complejidad se mide en operaciones, en unidades de tiempo, etc. (Ya que las constantes multiplicativas no importan).

# Análisis Asintótico

| n | $3n^3$ ns | $19500000n$ ns |
|---|---|---|
| 10 | 3 $\mu s$ | 0.2 s |
| 100 | 3 ms | 2.0 s |
| 1000 | 3 s | 20 s |
| 2500 | 50 s | 50 s |
| 10000 | 49 min | 3.2 min |
| 1000000 | 95 años | 5.4 hrs |

# Notación "Gran-O"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in O(g(n))$$

Si y solo si existen constantes $c > 0$ y $n_0$ tal que:

$$f(n) \leq cg(n) \qquad \forall n > n_0$$

# Notación "Gran-O"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in O(g(n))$$

Si y solo si existen constantes $c > 0$ y $n_0$ tal que:

$$f(n) \leq cg(n) \qquad \forall n > n_0$$

Esto implica que $O(g(n))$ es el conjunto de todas las funciones $f$ que tienen como cota superior asintótica a $g$.

# Ejemplo

Demostremos que $n \in O(n^2)$:

# Ejemplo

Demostremos que $n \in O(n^2)$:

$$n < n^2 \rightarrow$$

# Ejemplo

Demostremos que $n \in O(n^2)$:

$$n < n^2 \rightarrow n > 1$$

Si elegimos $n_0 = 1$ y $c = 1$, entonces:

# Ejemplo

Demostremos que $n \in O(n^2)$:

$$n < n^2 \rightarrow n > 1$$

Si elegimos $n_0 = 1$ y $c = 1$, entonces:

$$n < 1n^2 = n^2 \qquad \forall n > 1$$

Por ende $n \in O(n^2)$

# Otro ejemplo

- Demostrar que $n^3 \notin O(n^2)$.

# Otro ejemplo

- Demostrar que $n^3 \notin O(n^2)$.
- Demostrar que $2^n \in O(e^n)$.

# Otro ejemplo

- Demostrar que $n^3 \notin O(n^2)$.
- Demostrar que $2^n \in O(e^n)$.
- Demostrar que $3 \in O(1)$.

# Otro ejemplo

- Demostrar que $n^3 \notin O(n^2)$.
- Demostrar que $2^n \in O(e^n)$.
- Demostrar que $3 \in O(1)$.
- Demostrar que $n \log n \in O(n^2)$.

# Método alternativo

Otra forma de demostrar que una función $f$ pertenece al conjunto $O(g(n))$ es:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Si el limite existe y no es infinito, entonces $f$ no crece mas rápido que $g$.

## Recordatorio

Es posible usar la regla de l'hopital para calcular el limite.

# Notación "Gran-Omega"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in \Omega(g(n))$$

Si y solo si existen constantes $c > 0$ y $n_0$ tal que:

$$f(n) \geq cg(n) \qquad \forall n > n_0$$

# Notación "Gran-Omega"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in \Omega(g(n))$$

Si y solo si existen constantes $c > 0$ y $n_0$ tal que:

$$f(n) \geq cg(n) \qquad \forall n > n_0$$

Esto implica que $\Omega(g(n))$ es el conjunto de todas las funciones $f$ que tienen como cota inferior asintótica a $g$.

# Método alternativo

Otra forma de demostrar que una función $f$ pertenece al conjunto $\Omega(g(n))$ es:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$

Si el limite existe o es infinito, entonces $f$ crece mas rápido que $g$.

# Notación "Gran-Theta"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in \Theta(g(n))$$

Si y solo si existen constantes $c_1, c_2 > 0$ y $n_0$ tal que:

$$c_1 g(n) \geq f(n) \geq c_2 g(n) \qquad \forall n > n_0$$

# Notación "Gran-Theta"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in \Theta(g(n))$$

Si y solo si existen constantes $c_1, c_2 > 0$ y $n_0$ tal que:

$$c_1 g(n) \geq f(n) \geq c_2 g(n) \qquad \forall n > n_0$$

Esto implica que $\Theta(g(n))$ es el conjunto de todas las funciones $f$ que tienen como cota asintótica inferior y superior a $g$.

# Notación "Gran-Theta"

Sean $f$ y $g$ funciones positivas con dominio real. Se denota:

$$f(n) \in \Theta(g(n))$$

Si y solo si existen constantes $c_1, c_2 > 0$ y $n_0$ tal que:

$$c_1 g(n) \geq f(n) \geq c_2 g(n) \qquad \forall n > n_0$$

Esto implica que $\Theta(g(n))$ es el conjunto de todas las funciones $f$ que tienen como cota asintótica inferior y superior a $g$.
Otra interpretación es que $\Theta(g(n))$ contiene todas las funciones que asintoticamente crecen a la misma velocidad que $g$.

# Método alternativo

Otra forma de demostrar que una función $f$ pertenece al conjunto $\Theta(g(n))$ es:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \qquad 0 < c < \infty$$

Si el limite existe, entonces $f$ crece a la misma velocidad que $g$.

# Resumen notaciones

- $\Omega(g)$: Funciones que crecen mas rápido que $g$.
- $\Theta(g)$: Funciones que crecen a la misma velocidad que $g$.
- $O(g)$: Funciones que crecen a menor velocidad que $g$.

El uso de las notaciones permite comparar funciones por su tasa de crecimiento mas que por su valor exacto.

# Propiedades

- $\Theta(g) = O(g) \cap \Omega(g)$
- Si $f \in \Theta(g)$, entonces $g \in \Theta(f)$
- Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$ (Transitividad).
- $O(f + g) = O(\text{máx}(f, g))$

# Simplificación

- $O(n^3 + n^2 + 1)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$
- $O(n + n \log n)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$
- $O(n + n \log n)$
- $O(e^n + n!)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$
- $O(n + n \log n)$
- $O(e^n + n!)$
- $O(\log n! + n^2 + n^k)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$
- $O(n + n \log n)$
- $O(e^n + n!)$
- $O(\log n! + n^2 + n^k)$
- $O(\sqrt{n} + \log n)$

# Simplificación

- $O(n^3 + n^2 + 1)$
- $O(n^5 + n \log n)$
- $O(n + n \log n)$
- $O(e^n + n!)$
- $O(\log n! + n^2 + n^k)$
- $O(\sqrt{n} + \log n)$
- $O(\sum_{i=0}^{n} n^i + \sum_{i=0}^{n} \log i)$

# Porque usar las Notaciones Gran-$O$, Gran-$\Omega$ y Gran-$\Theta$?

- Porque lo mas importante de la complejidad de un algoritmo es como crece asintoticamente.
- En general se comparan las tasas de crecimiento mas que los valores exactos.
- Por ende la complejidad algorítmica se analiza en términos de la notación Gran-O.

# Crecimiento de funciones comunes

| Complejidad | Nombre |
|---|---|
| $O(1)$ | Constante |
| $O(\log n)$ | Logarítmico |
| $O(n^k), 0 < k < 1$ | Potencia fraccionaria |
| $O(n)$ | Lineal |
| $O(n \log n)$ | Loglineal |
| $O(n^k), k > 1$ | Polinomial |
| $O(e^n)$ | Exponencial |
| $O(n!)$ | Factorial |

# Ejercicios

Sea $T(n)$ la complejidad de un algoritmo.

1. Que significa que $T(n) \in O(n^4)$
2. Que significa que $T(n) \in \Theta(n \log n)$
3. Que interpretacion tiene que $T(n) \in \Omega(n)$.

Demuestre que:

1. $e^n \in \Omega(n^2)$.
2. $1 \in O(n!)$
3. $\frac{x}{x+1} \in \Theta(1)$

# Ejercicios

$$T(n) = 2T(n/2) + n^2$$

# Ejercicios

$$T(n) = 2T(n/2) + n^2 \rightarrow T(n) \in \Theta(n^2)$$

$$T(n) = 4T(n/2) + n^2$$

# Ejercicios

$$T(n) = 2T(n/2) + n^2 \rightarrow T(n) \in \Theta(n^2)$$

$$T(n) = 4T(n/2) + n^2 \rightarrow T(n) \in \Theta(n^2 \log n)$$

$$T(n) = 4T(n/2) + n^2 \log n$$

# Nota

- Desde aqui en adelante, la complejidad algoritmica se analizara siempre en terminos de las notaciones asintoticas. ($O - \Omega - \Theta$).
- Se usa de preferencia la notacion que se ajuste mejor.

# Analisis “Simple”

Si existen varios algoritmos que resuelven el mismo problema, la logica dictamina que se deberia elegir el algoritmo con menor costo computacional (Complejidad).

- Esto generalmente implica eligir el algoritmo de menor tasa de crecimiento, usando la notacion O.
- Pero se debe considerar que para entradas pequeñas, el algoritmo mas rapido puede no ser el de menor tasa de crecimiento.

# Análisis del Caso Promedio

- El analisis del caso promedio usa $A(n)$ para hacer la comparacion entre 2 o mas algoritmos.
- La comparacion se hace en terminos de la notacion $O - \Omega - \Theta$.
- Se elige el algoritmo de menor tasa de crecimiento.

# Análisis del Peor Caso

- El analisis del peor caso usa $W(n)$ para hacer la comparacion entre 2 o mas algoritmos.
- La comparacion se hace en terminos de la notacion $O - \Omega - \Theta$.
- Se elige el algoritmo de menor tasa de crecimiento.

# Ejemplo

Dado un arreglo $A$ de largo $N$:

1. Cual es la complejidad algoritmica de la operacion buscar?
2. Se quiere insertar un elemento $e$ en la posicion $i$. Cual es la complejidad algoritmica del peor caso y el caso promedio de esta operacion?
3. Se quiere eliminar un elemento $e$ en la posicion $i$. Cual es la complejidad algoritmica del peor caso y el caso promedio de esta operacion?

# Solucion

1. Si usamos busqueda lineal, la complejidad de esta es $O(n)$.
2. Si queremos insertar un elemento en la posicion $i$ en un arreglo de largo $n$, entonces es necesario mover todos los elementos desde $i$ a $n$, por lo que la complejidad esta en $O(n - i)$.
3. El mismo caso del ejemplo anterior, es necesario mover todos los elementos desde $i$ a $n$, por lo que la complejidad esta en $O(n - i)$.

# Ejemplo

Dada una Tabla de Hash encadenada con N buckets y aproximadamente (en promedio) P elementos por bucket.

1. Cual es la complejidad algoritmica de la operacion insertar?
2. Cual es la complejidad algoritmica de la operacion buscar?
3. Cual seria complejidad algoritmica de la operacion buscar si se usara sondeo lineal en lugar de encadenamiento?

# Solucion

1. Calcular la funcion de hash toma $O(1)$, y asumiendo que se puede insertar al final de una lista enlazada en $O(1)$, la insercion tomara tiempo $O(1+1) = O(1)$.
2. Buscar requiere recorrer una lista enlazada con P nodos en promedio, por lo cual la complejidad de esta operacion es $O(P)$.
3. Sondeo lineal implica recorrer toda la tabla de hash en busqueda de la llave a buscar, por lo que su complejidad es $O(NP)$.

# Ejemplo

Dado un Arbol Binario de Busqueda de altura H, con N nodos.

1. Cual es la complejidad algoritmica de la operacion insertar?
2. Cual es la complejidad algoritmica de la operacion buscar?
3. Como se comparan las operaciones del Arbol Binario de Busqueda con las operaciones de la tabla de Hash? (Hablando de Complejidad Algoritmica).

# Solucion

Nota: la altura de un arbol con N nodos es $H = \log_2 N$.

1. La operacion insertar tiene una complejidad de $O(H) = O(\log_2 N)$.
2. La operacion buscar tiene una complejidad de $O(H) = O(\log_2 N)$.
3. La tabla de hash es en general mas rapida, ya que tiene una complejidad computacional menor ($O(1) < O(\log_2 N)$), pero esta comparacion depende del numero de veces que se debe hacer cada operacion (Busqueda vs Insercion).

# Analisis de Costo Amortizado

Es un metodo de analisis y diseño de algoritmos que considera toda la secuencia de operaciones de un algoritmo. Trabaja con los siguientes supuestos:

- Ciertas operaciones pueden ser extremadamente costosas, pero ocurren con una baja frecuencia.
- Por ende el costo pequeño de las operaciones de alta frecuencia contrapesa el gran costo de las operaciones de baja frecuencia.
- Esto supone no un solo operacion/algoritmo, sino varios algoritmos que operan en algun dato en comun (tipicamente una estructura de datos).

# Analisis Costo Amortizado

La idea central es que una operacion de alto costo puede alterar el estado del dato/estructura de tal forma que la operacion de alto costo no suceda nuevamente en una cierta cantidad (larga) de tiempo.
Entonces el costo de esa operacion se "amortiza" durante el tiempo.

## Calculo del Costo Amortizado

$$C_a(n) = \frac{C_o(n)}{n}$$

- $C_o$ es el costo de todas las operaciones a realizar.
- $C_a$ es el costo amortizado de las operaciones.

# Arreglos Dinamicos

Fuera de la teoria, se requiere que los arreglos tengan largos variables, lo que usualmente implica cambiar su largo al ejecutar cierta operacion.
Un arreglo dinamico posee las siguientes operaciones:

- Insertar un elemento al final del arreglo.
- Eliminar un elemento del arreglo.
- Acceso al elemento en la posicion i-esima.

Para ello se crea una clase que encapsula el arreglo, el largo y la capacidad del arreglo asignado en memoria (el cual puede ser mayor al largo del arreglo).

# Insercion

Para insertar un elemento, hay que diferenciar 2 casos:

## La capacidad del arreglo es mayor o igual al nuevo largo del arreglo

Se actualiza el largo del arreglo y se inserta el elemento.

## La capacidad del arreglo es menor al nuevo largo del arreglo

Se solicita un arreglo del largo del doble actual, y se copian los elementos del arreglo viejo al nuevo. Se actualizan el largo y la capacidad, y se inserta el elemento.

Esta implementacion de la operacion de Insercion se denomina Doblado de Arreglos.

# Insercion

```
class ArregloDinamico<T> {
    T *arreglo;
    int largo, int capacidad;

    void insertar(T e)
    {
        if(capacidad < largo) {
           T *nuevo = new T[2 * largo];
           copy(arreglo, nuevo, largo);
           delete [] arreglo;
           arreglo = nuevo;
           capacidad = 2 * largo
        }
        largo = largo + 1;
        arreglo[largo - 1] = e;
    }
}
```

# Analisis de Costo Amortizado del Doblado de Arreglos

Si ejecutamos la operacion insertar N veces, podemos esperar lo siguiente:

- Se ejecuta N veces la insercion en si (2 ultimas lineas del codigo anterior).
- Se ejecutan $\log_2 n$ veces el doblado de arreglos en promedio. (Numero de potencias de 2 entre 0 y N).
- Cada operacion de doblado tiene un costo de $2^i$, pero esto se ejecuta $\log_2 n$ veces, por ende $i \in [0, \log_2 n]$.

Estos supuestos son los del peor caso.

# Analisis de Costo Amortizado del Doblado de Arreglos

Por ende el costo total es:

# Analisis de Costo Amortizado del Doblado de Arreglos

Por ende el costo total es:

$$C_o(n) = n + \sum_{i=0}^{\log_2 n} 2^i = 3n$$

Y el costo amortizado:

# Analisis de Costo Amortizado del Doblado de Arreglos

Por ende el costo total es:

$$C_o(n) = n + \sum_{i=0}^{\log_2 n} 2^i = 3n$$

Y el costo amortizado:

$$C_a = \frac{C_o(n)}{n}$$

# Analisis de Costo Amortizado del Doblado de Arreglos

Por ende el costo total es:

$$C_o(n) = n + \sum_{i=0}^{\log_2 n} 2^i = 3n$$

Y el costo amortizado:

$$C_a = \frac{C_o(n)}{n} = \frac{3n}{n} = 3 \in O(1)$$

Como vemos, el doblado de arreglos tiene costo amortizado constante.

# Eliminado

Para eliminar el i-esimo elemento de un arreglo dinamico, debemos hacer lo siguiente:

- Mover todos los elementos desde i+1 hasta el largo del arreglo una posicion hacia la izquierda.
- Asi el elemento i+1 sobreescribe al elemento i (eliminandolo), y el resto se corre una posicion.
- Decrementar en 1 el largo del arreglo.

Cual es el costo de esta operacion?

# Acceso

El acceso al arreglo es directo.

```
class ArregloDinamico<T> {

    T& operator[](int i)
    {
        if(i >= 0 && i < largo) {
            return arreglo[i];
        } else {
            /* Error, indice fuera de limites. */
        }
    }
}
```

Cual es el costo de esta operacion?

# Bibliografia

Para esta unidad, se recomienda leer:

1. Capitulos 1 y 3 del Libro “Algoritmos Computacionales: Introduccion al Analisis y Diseño” de Sara Baase y Allen Van Gelder.
2. “A Theoretician’s Guide to the Experimental Analysis of Algorithms” de David S. Johnson. (Disponible en Reko).
3. “Smoothed Analysis: An Attempt to Explain the Behavior of Algorithms in Practice” de Daniel Spielman y Shang-Hua Teng. (Disponible en Reko).