Algorítmica

Grado en Ingeniería Informática

Tema 1 – La eficiencia de los algoritmos

Manuel Pegalajar Cuéllar
manupc@ugr.es
Departamento de Ciencias de la Computación e Inteligencia Artificial
http://decsai.ugr.es

UNIVERSIDAD DE GRANADA

DECSAI

Objetivos del tema

- Conocer el concepto de algoritmo.
- Plantearse la búsqueda de varias soluciones distintas para un mismo problema y evaluar la bondad de cada una de ellas.
- Tomar conciencia de la importancia del análisis de la eficiencia de un algoritmo como paso previo a su implementación en un lenguaje de programación.
- Conocer la notación asintótica para describir la eficiencia de un algoritmo, distinguiendo entre los distintos tipos de análisis que se pueden realizar: caso más favorable, más desfavorable y promedio.
- Saber realizar el análisis de eficiencia de un algoritmo, tanto a nivel teórico como empírico, y saber contrastar resultados experimentales con los teóricos.
- Conocer las técnicas básicas de resolución de ecuaciones de recurrencia: expansión de la recurrencia, método de la ecuación característica y utilización de fórmulas maestras.

Estudia este tema en…

- G. Brassard, P. Bratley, “Fundamentos de Algoritmia”, Prentice Hall, 1997, pp. 65-166
- J.L. Verdegay: Lecciones de Algorítmica. Editorial Técnica AVICAM (2017).

Anotación sobre estas diapositivas:

El contenido de estas diapositivas es esquemático y representa un apoyo para las clases presenciales teóricas. No se considera un sustituto para apuntes de la asignatura.

Se recomienda al alumno completar estas diapositivas con notas/apuntes propios, tomados en clase y/o desde la bibliografía principal de la asignatura.

La eficiencia de los algoritmos

1. El concepto de algoritmo.
2. La eficiencia de algoritmos: Problema, tamaño, instancia. Principio de Invarianza.
3. La notación asintótica. Órdenes peor, mejor y exacto.
4. Análisis de algoritmos.
5. Resolución de recurrencias.

Definición de algoritmo

Secuencia finita ordenada de pasos exentos de ambigüedad tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa* ***al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Definición de algoritmo

Secuencia finita ordenada de pasos exentos de ambigüedad tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa* ***al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Definición de algoritmo

Secuencia **finita** ordenada de pasos exentos de ambigüedad tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa* ***al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Definición de algoritmo

Secuencia finita **ordenada** de pasos exentos de ambigüedad tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa **al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Definición de algoritmo

Secuencia finita ordenada de pasos **exentos de ambigüedad** tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa* ***al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Definición de algoritmo

Secuencia finita ordenada de pasos exentos de ambigüedad tal que, al **llevarse a cabo con fidelidad**, dará como resultado la **tarea para la que se ha diseñado**.

- Su nombre proviene del matemático persa del siglo IX *Abd Allah Muhhamad ibn Musa* ***al-Khwarizmi***.

- Ejemplos:
  - Un programa de PC.

Propiedades/características de un algoritmo:

- **Es una noción abstracta**. No depende del lenguaje donde se implemente (C++, Basic, Fortran,...).
- **Está bien definido**. Cada paso está claramente expresado y sin ambigüedades.
- **Es coherente**. Con los mismos datos iniciales siempre se obtiene el mismo resultado.
- **Finitud**. El algoritmo debe terminar.
- **Efectividad**. Debe resolver el problema planteado.

Si alguna de estas características se incumple, entonces no es un algoritmo

- **No es un algoritmo**: Receta de cocina mal planteada:

1. Poner aceite en una sartén.
2. Esperar a que el aceite esté caliente.
3. Echar carne.
4. Echar sal.
5. Mientras que la carne no esté lista, mover la carne.
6. Servir la carne.

Este "algoritmo" no es válido:

- No está **bien definido** (¿cuánto aceite se echa? ¿cuánta sal? ¿A qué potencia de fuego?).
- No es **coherente** (no siempre proporciona el mismo resultado –a veces muy aceitoso/salado, etc.-).
- Es más, no hemos dicho que se encienda el fuego, por lo que **tampoco es finito** dado que el aceite jamás llegará a estar caliente.

No es un algoritmo: Código fuente

Depende del lenguaje de programación (**No es noción abstracta**)

```
void SentarComensales() {
        list<int> orden;
        orden.push_front(0);
        vector<bool> ya_sentados(afinidades.size(), false);
        ya_sentados[0] = true;
        mejor_aversion = numeric_limits<int>::max();
        Cena(ya_sentados, 0, orden);
}
```

- **No es un algoritmo**: pseudocódigo mal hecho

Diferentes Errores:

Ambigüedades: ¿Qué es R, Qué es C? ¿Una lista, pila, cola…?

Mala definición de operaciones: ¿Qué se supone que hace C-R[0]?, ¿y %? ¿y nearest(x, C)?

Mala definición de datos de entrada y/o salida: ¿Qué es M?

$$
\begin{array}{l}
\mathcal{R} = [\text{random()}\,\%|\ \Omega\ ||] \\
\mathcal{C}\ \text{-}\ \mathcal{R}[0] \\
\textbf{for}\ \text{pos in } [0\text{,n-2}]\ \textbf{do} \\
\quad \text{bestPos} = \text{nearest}(\mathcal{M}[\mathcal{R}[\text{pos}]],\mathcal{C}) \\
\quad \mathcal{R}.\text{push}(\mathcal{C}[\text{bestPos}]) \\
\quad \mathcal{C}.\text{erase}(\text{bestPos}) \\
\textbf{end for} \\
\text{return } \mathcal{R}
\end{array}
$$

- **No es un algoritmo**: pseudocódigo mal hecho. Otro ejemplo.

```
solution = next PriorityQueue(<length, path>)
if path is completed then
    localLength = path.length
else
    minPossible = estimate path
    localLength = solution.length
end if
if localLength < best Solution then
    for city not in path do
        add city to the path
        compute new min Length
        Explore new solution if it's worth
    end for
end if
```

- Un ejemplo de “**buen algoritmo**”:

  Algoritmo SumaNElementos(N: Máximo valor a sumar)

  Devuelve sumaa: Suma de los N primeros números naturales

  1. suma ← 0
  2. Para cada valor v entre 0 y N inclusive, hacer:
  3. suma ← suma+v
  4. Fin-Para
  5. Devolver a SUMA

- **Cumple con la definición y todas sus propiedades:**

  - ✓ Secuencia finita ordenada de pasos exentos de ambigüedad tal que, al llevarse a cabo con fidelidad, dará como resultado la tarea para la que se ha diseñado.
  - ✓ Está bien definido, es coherente, termina y es efectivo.
  - ✓ Es abstracto: No depende de dónde se implemente.

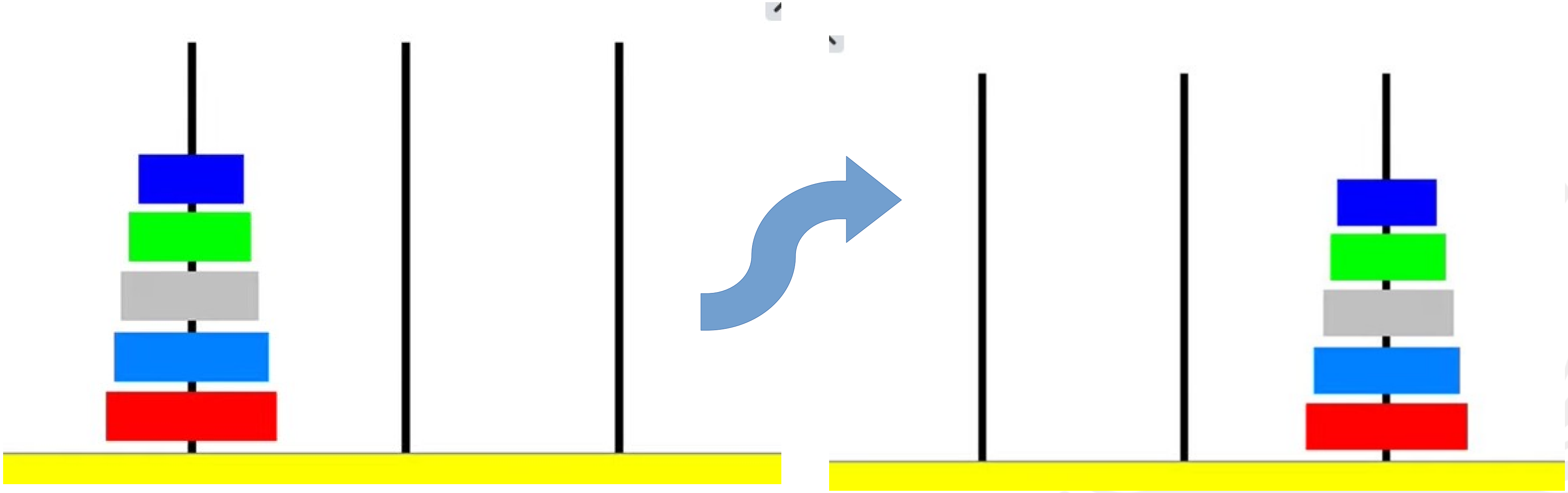
- Ejemplo: El problema de las **Torres de Hanoi**.

Las ***Torres de Hanoi*** es un conocido pasatiempo matemático. Se dispone de 3 soportes verticales, uno de ellos (el **origen**) con ***N*** discos, ordenados de mayor a menor tamaño (en orden ascendente).

Se desea pasar todos los discos desde el soporte **origen** hasta el soporte **destino** con las siguientes reglas:

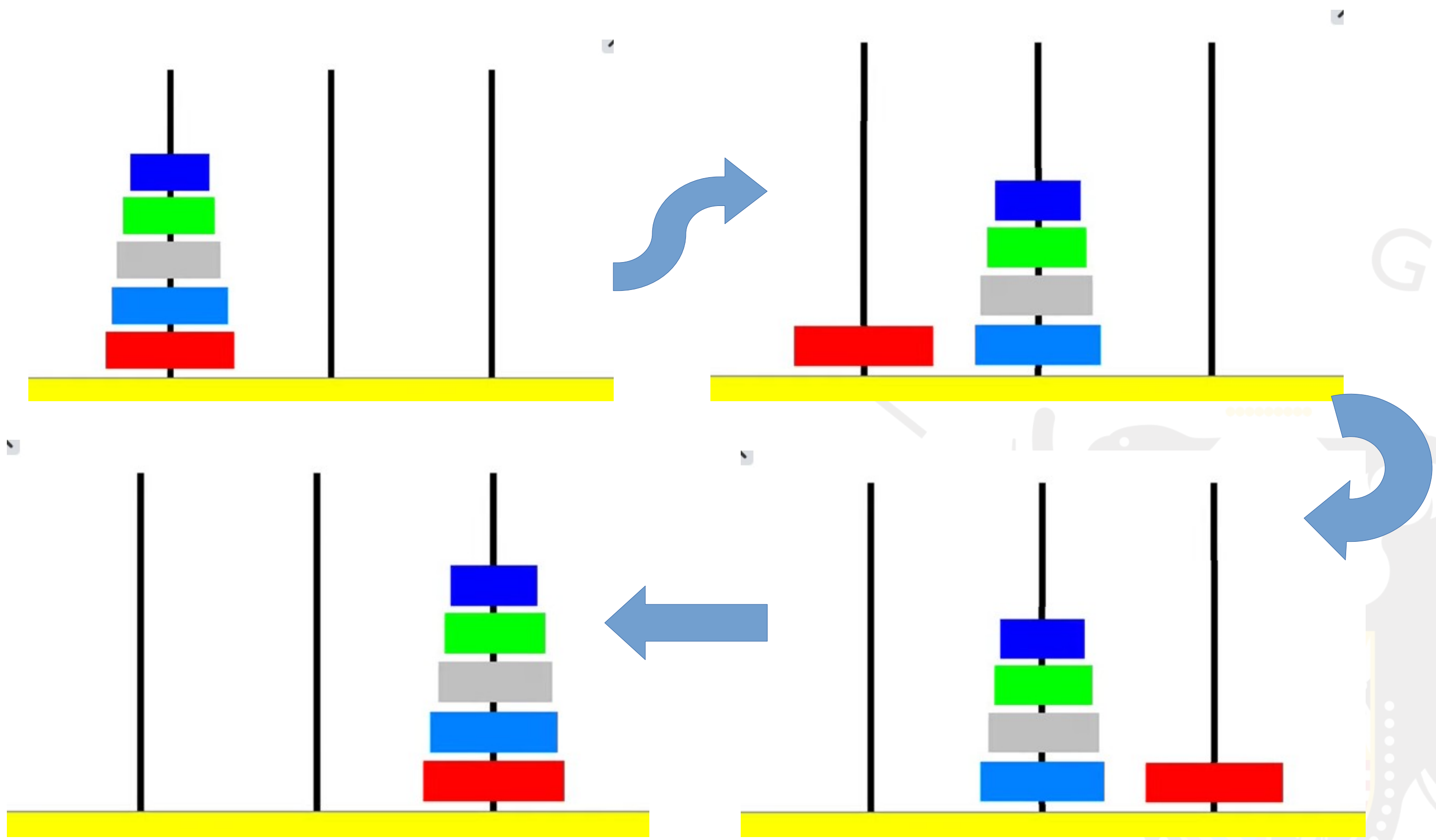
- Sólo es posible mover un disco simultáneamente.
- No se puede depositar un disco de mayor tamaño sobre otro de menor tamaño.

¿Sería posible dar un procedimiento exacto para resolver este rompecabezas?



- Idea general

Asumiendo que la torre origen tiene un número de N discos, la idea general del método podría consistir en mover N-1 discos de **origen** al soporte **intermedio**, usando **destino** como auxiliar. Luego, pasar el último disco (el de mayor tamaño) de **origen** a **destino** y, finalmente, pasar todos los N-1 discos de **intermedio** a **destino**, usando **origen** como auxiliar.



- Solución: El procedimiento dado es un **algoritmo**.
  - **ENTRADAS:**
    - **Altura**: Número de discos existentes en el origen
    - **Origen**: Soporte origen desde el que se mueven los discos
    - **Destino**: Soporte origen desde el que se mueven los discos
    - **Intermedio**: Soporte intermedio a usar como auxiliar
    - **SALIDAS:** Se proporciona por pantalla los pasos a seguir.
  - **PROCEDIMIENTOS AUXILIARES:**
    - **Mover(x,y)** -> Mueve el disco superior desde el soporte x al soporte y.
  - **EFECTO:** Mueve Altura discos de Origen a Destino, usando Intermedio como auxiliar.

PROCEDIMIENTO Hanoi(Altura, Origen, Destino, Intermedio) :

```
Si Altura >= 1 :
    Hanoi(Altura-1, Origen, Intermedio, Destino)
    Mover Disco de Origen a Destino
    Hanoi(Altura-1, Intermedio, Destino, Origen)
```

Solución: Ejemplo de implementación en C/C++:

In [1]:
```

#include <iostream>
using namespace std;

void Hanoi(int Altura, int Origen, int Destino, int Intermedio);
void Mover(int x, int y);



```

In [2]:
```
void Mover(int x, int y) {
    cout<<"Mover disco de "<<x<<" a "<<y<<endl;
}

```

In [3]:
```

void Hanoi(int Altura, int Origen, int Destino, int Intermedio) {

    if (Altura>=1) { // Si queda algún disco en Origen
        Hanoi(Altura-1, Origen, Intermedio, Destino);
        Mover(Origen, Destino);
        Hanoi(Altura-1, Intermedio, Destino, Origen);
    }
}

```

- Solución: Ejemplo de ejecución:

```

void Hanoi(int Altura, int Origen, int Destino, int Intermedio) {

    if (Altura>=1) { // Si queda algún disco en Origen
        Hanoi(Altura-1, Origen, Intermedio, Destino);
        Mover(Origen, Destino);
        Hanoi(Altura-1, Intermedio, Destino, Origen);
    }
}


int Origen= 1, Destino= 3, Intermedio= 2;
int Altura= 3;

Hanoi(Altura, Origen, Destino, Intermedio);
```

```
Mover disco de 1 a 3
Mover disco de 1 a 2
Mover disco de 3 a 2
Mover disco de 1 a 3
Mover disco de 2 a 1
Mover disco de 2 a 3
Mover disco de 1 a 3
```

La eficiencia de los algoritmos

1. El concepto de algoritmo.
2. La eficiencia de algoritmos: Problema, tamaño, instancia. Principio de Invarianza.
3. La notación asintótica. Órdenes peor, mejor y exacto.
4. Análisis de algoritmos.
5. Resolución de recurrencias.

DECSAI

¿Porqué estudiar eficiencia?

- Existe un método para resolver un problema: ¿Es viable implementarlo?
- Existen varios métodos que resuelven el mismo problema. ¿Cuál de ellos es mejor? ¿En qué situaciones? Ejemplos: Algoritmos de ordenación (inserción, burbuja, selección, heapsort, quicksort, mergesort…)

¿Cómo medimos la eficiencia?

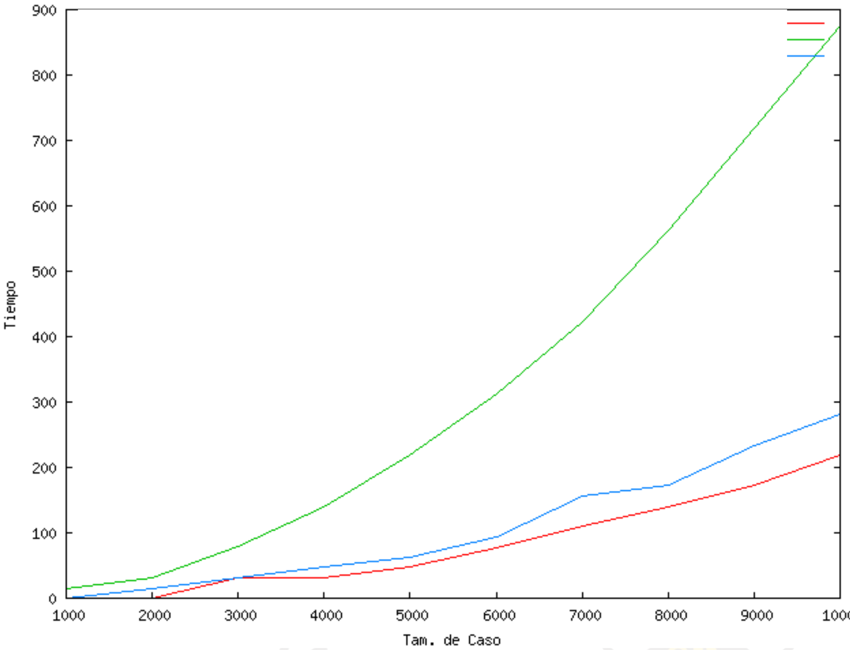
- **En tiempo:** Tiempo que tarda un algoritmo en resolver un problema.
- **En espacio:** Recursos (memoria, espacio en disco, etc.) necesarios para resolver el problema.

- **En la asignatura nos centramos en el estudio de la eficiencia basándonos en el tiempo de ejecución de un algoritmo.**

Notación que utilizaremos

- **Problema:** Es el problema general que queremos resolver (ordenación, búsqueda, multiplicación de matrices, etc.)
- **Instancia del problema:** Problema concreto a resolver. Ejemplo: Ordenar el vector (9, 6, 10, 24, 11, 14).
- **Caso:** Instancia o conjunto de instancias de un problema con dificultad idéntica o muy similar. **Tres tipos: Caso peor, caso mejor, caso promedio.**
- **Tamaño del caso:** Tamaño de la instancia o instancias del problema a resolver. Para el problema de ordenar un vector del ejemplo anterior, su tamaño de caso es n=6.

- Ejemplo: Se han implementado los algoritmos de ordenación por Inserción (rojo), Burbuja (verde) y Selección (azul).
- Se han ordenado vectores aleatorios de tamaño 1000, 2000, 3000, …, 10000 **(los mismos vectores con los 3 algoritmos)**.
- Se ha medido el tiempo que tarda cada algoritmo en ordenar el vector. ¿Cuál es el mejor algoritmo, a simple vista?



Gráfica

- **Eje X: Tamaño del caso**
- **Eje Y: Tiempo de ejecución**

¿Qué es el caso peor?

Es la instancia o el conjunto de instancias del problema que hacen que nuestro algoritmo ejecute el **máximo número de operaciones** posible y, por tanto, que tarde el máximo tiempo en ejecutarse comparado con otras instancias de caso **del mismo tamaño**.

¿Qué es el caso mejor?

Es la instancia o el conjunto de instancias del problema que hacen que nuestro algoritmo ejecute el **mínimo número de operaciones** posible y, por tanto, que tarde el mínimo tiempo en ejecutarse comparado con otras instancias de caso **del mismo tamaño**.

En la asignatura utilizaremos la **notación O(·)** para indicar la eficiencia de un algoritmo en el **caso peor**, y la **notación Ω(·)** para indicar la eficiencia en el **caso mejor**.

¿Cómo calcularemos la eficiencia de un algoritmo?

- **Método Empírico:** Se implementa el algoritmo y se mide el tiempo de ejecución.
- **Método Teórico:** Se calcula una función matemática que indique cómo evolucionará el tiempo de ejecución del algoritmo según varíe el tamaño **n** del caso.
- **Método Híbrido:** Mezcla de ambos.

La eficiencia de un algoritmo, en cualquier caso, es independiente del lenguaje de programación donde se implemente gracias al **Principio de Invarianza**.

Principio de Invarianza

Dadas dos implementaciones **I1** e **I2** de un mismo algoritmo, el tiempo de ejecución para una misma instancia de tamaño **n**, $T_{I1}(n)$ y $T_{I2}(n)$, no diferirá en más de una constante multiplicativa. Es decir, **existe una constante positiva K que verifica:**

$$T_{I1}(n) <= K * T_{I2}(n)$$

Ejemplo: Proyecto BurbujaCpp.ipynb, Tema 1

```
// Definición del algoritmo de ordenación por burbuja
// Ordena el vector v entre dos componentes posini y posfin, inclusive
void Burbuja(vector <int> &v, int posini, int posfin) {

    int i, j;
    double aux;
    bool haycambios= true;

    i= posini;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado

        for (j= posfin; j>i; j--) { // Recorremos vector de final a i
            if (v[j-1]>v[j]) { // Dos elementos consecutivos mal ordenados
                aux= v[j]; // Los intercambiamos
                v[j]= v[j-1];
                v[j-1]= aux;

                // Al intercambiar, hay cambio
                haycambios= true;
            }
        }
        i++;
    }
}
```

tamaño del vector = posfin - posini +1

Ejemplo: Proyecto BurbujaCpp.ipynb, Tema 1

```
// Definición del algoritmo de ordenación por burbuja
// Ordena el vector v entre dos componentes posini y posfin, inclusive
void Burbuja(vector <int> &v, int posini, int posfin) {

    int i, j;
    double aux;
    bool haycambios= true;

    i= posini;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado

        for (j= posfin; j>i; j--) { // Recorremos vector de final a i
            if (v[j-1]>v[j]) { // Dos elementos consecutivos mal ordenados
                aux= v[j]; // Los intercambiamos
                v[j]= v[j-1];
                v[j-1]= aux;

                // Al intercambiar, hay cambio
                haycambios= true;
            }
        }
        i++;
    }
}
```

- **Tamaño del caso "n": posfin-posini+1** (las *n* componentes a ordenar)
- **Mejor caso del algoritmo:** Cuando "v" está ya ordenado.
- **Peor caso del algoritmo:** Cuando "v" está ordenado descendentemente.

Ejemplo: Burbuja implementado en C++ y en Python

- C++: Proyecto BurbujaCpp.ipynb , Tema 1
- Python: Proyecto BurbujaPython, Tema 1

```cpp
// Definición del algoritmo de ordenación por burbuja
// Ordena el vector v entre dos componentes posini y posfin, i
void Burbuja(vector <int> &v, int posini, int posfin) {

    int i, j;
    double aux;
    bool haycambios= true;

    i= posini;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado

        for (j= posfin; j>i; j--) { // Recorremos vector de fi
            if (v[j-1]>v[j]) { // Dos elementos consecutivos m
                aux= v[j]; // Los intercambiamos
                v[j]= v[j-1];
                v[j-1]= aux;

                // Al intercambiar, hay cambio
                haycambios= true;
            }
        }
        i++;
    }
}
```

```python
# Definición del algoritmo de ordenación por burbuja
# Ordena el vector v entre dos componentes posini y posfin, inclusive
def Burbuja(v, posini, posfin) :

    haycambios= True

    i= posini;
    while (haycambios):

        haycambios=False; # Suponemos vector ya ordenado

        # Recorremos vector de final a i
        for j in range(posfin, i-1, -1):

            # Dos elementos consecutivos mal ordenados
            if (v[j-1]>v[j]) :
                aux= v[j] # Los intercambiamos
                v[j]= v[j-1]
                v[j-1]= aux

                # Al intercambiar, hay cambio
                haycambios= True

        i= i+1

    return v
```

Ejemplo: Medición del tiempo de ejecución

- **Caso:** Peor (Para este algoritmo: vector ordenado al revés)
- **Instancias:** Tamaños de 1.000 a 10.000, de 1.000 en 1.000

```
// Declaración de constantes y datos globales
const string ficheroSalida= "salidaBurbujaCpp.txt"; // Fichero
const vector<int> tamCasos= {1000, 2000, 3000, 4000, 5000,
                             6000, 7000, 8000, 9000, 10000 };
```

```
for (int i= 0; i<tamCasos.size(); i++) {

    // Generamos instancia del caso peor para el algoritmo
    vector.clear();
    for (int j= tamCasos[i]-1; j>= 0; j--) {
        vector.push_back(j);
    }

    // Ordenamos con burbuja, midiendo el tiempo de ejecución
    t0= std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que comienza la ejecuciÛn del
    Burbuja(vector, 0, vector.size()-1);
    tf= std::chrono::high_resolution_clock::now(); // Cogemos el tiempo en que finaliza la ejecuciÛn del

    // Calculamos el tiempo entre t0 y tf
    unsigned long tejecucion= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();

    // Mostramos tamaño de caso y su tiempo en microsegundos
    cout<<tamCasos[i]<<" "<<tejecucion<<endl;

    // Lo guardamos también a fichero
    fout<<tamCasos[i]<<" "<<tejecucion<<endl;
}
```

Ejemplo: Resultados

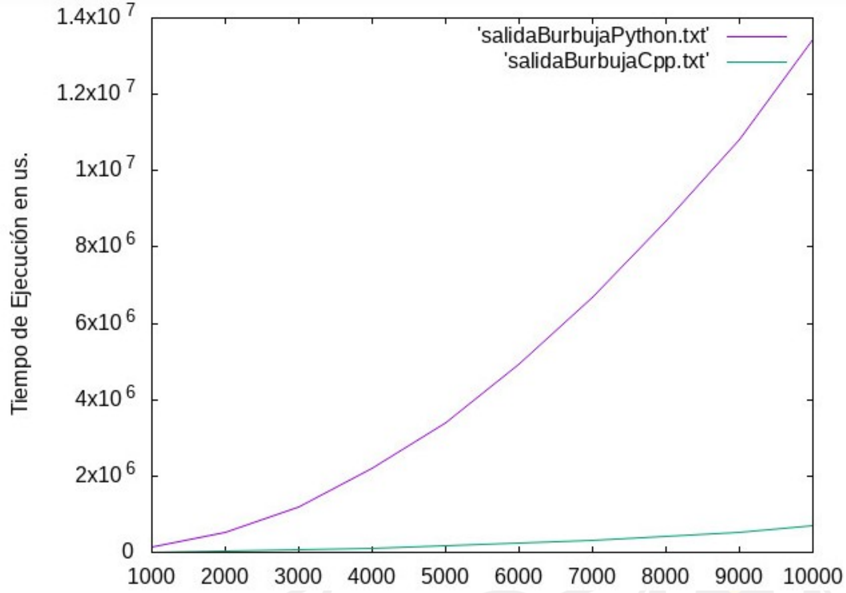
- Proyecto CalculoTiemposPythonCpp.ipynb

```
int main() {
    vector<int>tamCasoCpp, tiemposCpp, tamCasoPython, tiemposPython;
    double K; // Constante multiplicativa

    CargarFichero(ficheroCpp, tamCasoCpp, tiemposCpp);
    CargarFichero(ficheroPython, tamCasoPython, tiemposPython);
    for (int i= 0; i<tamCasoCpp.size(); i++) {
        K= tiemposPython[i]/(1.0*tiemposCpp[i]);
        cout<<"Tam. Caso: "<<tamCasoCpp[i]<<", K="<<K<<endl;
    }
    return 0;
}

main();
```

```
Tam. Caso: 1000, K=21.9928
Tam. Caso: 2000, K=19.6928
Tam. Caso: 3000, K=19.5008
Tam. Caso: 4000, K=20.5277
Tam. Caso: 5000, K=20.4375
Tam. Caso: 6000, K=20.0777
Tam. Caso: 7000, K=20.9221
Tam. Caso: 8000, K=20.7638
Tam. Caso: 9000, K=20.581
Tam. Caso: 10000, K=19.5941
```



Ejemplo: Resultados

- Proyecto CalculoTiemposPythonCpp.ipynb

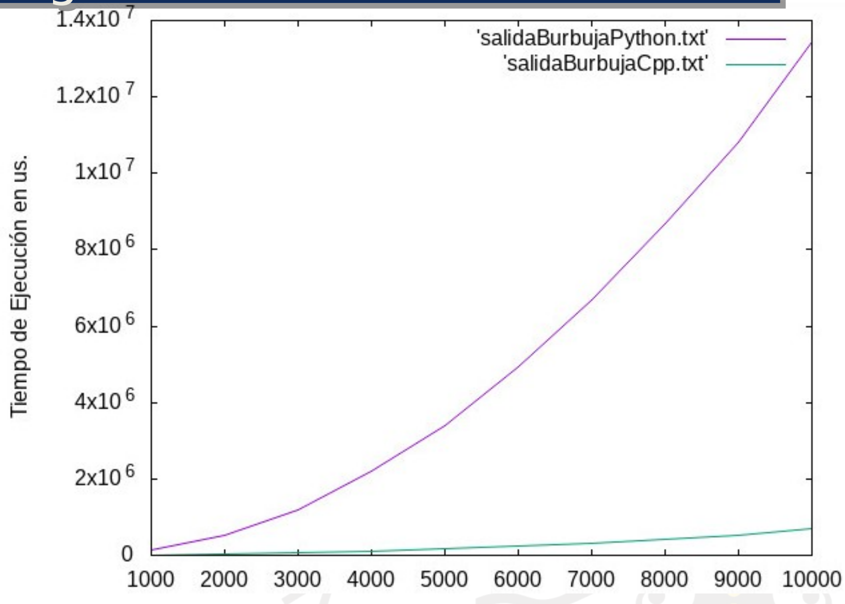
```
int main() {
    vector<int>tamCasoCpp, tiemposCpp, tamCasoPython, tiemposPython;
    double K; // Constante multiplicativa

    CargarFich
    CargarFich
    for (int i
        K= tie
        cout<<
    }
    return 0;
}

main();
```

Ambas gráficas tienen la misma forma: $f(n) \approx c \cdot n^2$, independientemente del lenguaje donde se implemente el algoritmo.

```
Tam. Caso: 1000, K=21.9928
Tam. Caso: 2000, K=19.6928
Tam. Caso: 3000, K=19.5008
Tam. Caso: 4000, K=20.5277
Tam. Caso: 5000, K=20.4375
Tam. Caso: 6000, K=20.0777
Tam. Caso: 7000, K=20.9221
Tam. Caso: 8000, K=20.7638
Tam. Caso: 9000, K=20.581
Tam. Caso: 10000, K=19.5941
```



La eficiencia de los algoritmos

La notación **O**

Se dice que **un algoritmo A es de orden O(f(n))**, donde **f(n)** es una función matemática **f(n):N->R**$^+$, cuando existe una implementación del mismo cuyo tiempo de ejecución $\mathbf{T_A(n)}$ es menor o igual que **K*f(n)**, donde **K** es constante, para “tamaños de casos grandes”.

Formalmente:

$$A \text{ es } O(f(n)) \leftrightarrow \exists K \in \Re^+, \exists n_0 \in N : T_A(n) \leq K \cdot f(n) \; \forall n > n_0$$

La **notación O** nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del **caso peor del problema**:

“*Como mucho, sabemos que el algoritmo no tardará más de K*f(n) en ejecutarse, en el peor de los casos*”.

Ejemplos de órdenes de eficiencia

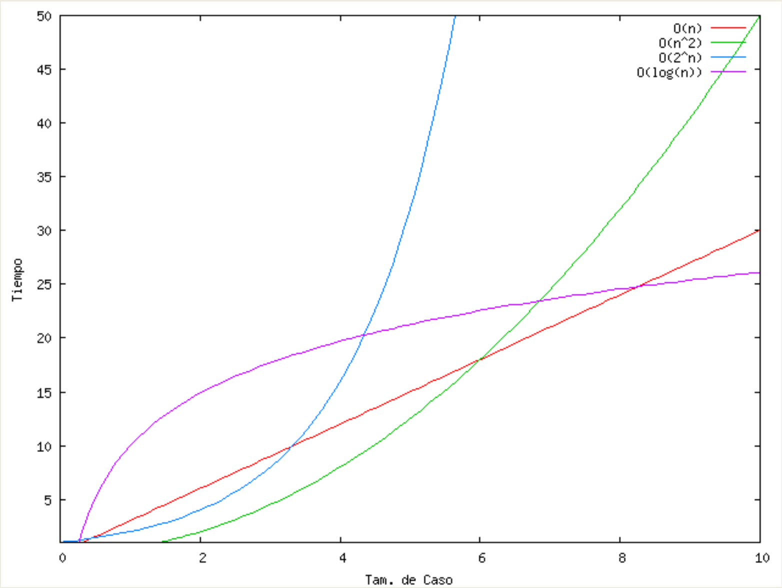
- Constante: O(1)
- Lineal: O(n)
- Cuadrático: $O(n^2)$
- Logarítmico: O(log(n))
- Exponencial: (a^n)
- Etc.

Si decimos que un algoritmo **A** es de orden **O(f(n))**, queremos decir que siempre podremos encontrar una constante positiva **K** tal que, **para valores muy grandes del tamaño de caso *n***, el tiempo de ejecución del algoritmo siempre será inferior o igual a **K** multiplicando a **f(n)**:

$$\mathbf{T_A(n) <= K \cdot f(n)}$$

Ejemplo: Gráfica de varias funciones f(n)

- **Algoritmo A:** O(n)
- **Algoritmo B:** O(n^2)
- **Algoritmo C:** O(2^n)
- **Algoritmo D:** O(log(n))



Cuestiones

- ¿Qué algoritmo es más eficiente?
- Si tuvieras que resolver un problema, ¿qué algoritmo de los anteriores implementarías para resolverlo?

Equivalencia entre órdenes de eficiencia

Para saber si dos órdenes de eficiencia **O(f(n))** y **O(g(n))** son equivalentes o no, podemos aplicar las siguiente reglas:

- **O(f(n)) ≡ O(g(n))** sii: $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to K \in R^{+}$
- **O(f(n)) > O(g(n))** sii: $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to \infty$

  orden f(n) peor que orden g(n)
- **O(f(n)) < O(g(n))** sii: $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to 0$

Por comodidad, ante equivalencia de órdenes de eficiencia siempre nos referiremos al más simple.

Ejemplo 1. Algoritmo A: **$O(n^2)$** ; Algoritmo B: **$O((4n+1)^2 + n)$**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{(4n+1)^2 + n} = \lim_{n\to\infty} \frac{n^2}{(16n^2 + 1^2 + 2\cdot 4n\cdot 1) + n} \to \frac{1}{16}$$

Son equivalentes

Ejemplo 2. Algoritmo A: **$O(2^n)$** ; Algoritmo B: **$O(3^n)$**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n \to 0$$

A es más eficiente que B

Ejemplo 3. Algoritmo A: **O(n)** ; Algoritmo B: **O(n·log(n))**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{n\cdot\log(n)} = \lim_{n\to\infty} \frac{1}{\log(n)} \to 0$$

A es más eficiente que B

Ejemplo 4. Algoritmo A: **$O((n^2+29)^2)$** ; Algoritmo B: **$O(n^3)$**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(n^2+29)^2}{n^3} \to \infty$$

B es más eficiente que A

Ejemplo 5. Algoritmo A: **O(n)** ; Algoritmo B: **O(log(n))**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{\log(n)} = \lim_{n\to\infty} \frac{10^n}{n} \to \infty$$

B es más eficiente que A

La notación Ω

Se dice que **un algoritmo A es de orden Ω(f(n))**, donde **f(n)** es una función matemática **f(n):N->R**$^+$, cuando existe una implementación del mismo cuyo tiempo de ejecución **T_A(n)** es mayor o igual que **K*f(n)**, donde **K** es constante, para "tamaños de casos grandes".

Formalmente:

$$A \text{ es } \Omega(f(n)) \leftrightarrow \exists K \in \Re^+, \exists n_0 \in N : T_A(n) \geq K \cdot f(n) \; \forall n > n_0$$

La **notación Ω** nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del **caso mejor del problema**:

"*Como poco, sabemos que el algoritmo no tardará menos de K*f(n) en ejecutarse, en el mejor de los casos*".

Ejemplo: Proyecto OrdenOmegaBurbujaCpp.ipynb, Tema 1

```
// Definición del algoritmo de ordenación por burbuja
// Ordena el vector v entre dos componentes posini y posfin, inclusive
void Burbuja(vector <int> &v, int posini, int posfin) {

    int i, j;
    double aux;
    bool haycambios= true;

    i= posini;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado

        for (j= posfin; j>i; j--) { // Recorremos vector de final a i
            if (v[j-1]>v[j]) { // Dos elementos consecutivos mal ordenados
                aux= v[j]; // Los intercambiamos
                v[j]= v[j-1];
                v[j-1]= aux;

                // Al intercambiar, hay cambio
                haycambios= true;
            }
        }
        i++;
    }
}
```

Orden exacto θ

Se dice que **un algoritmo A es de orden exacto θ(f(n))**, donde **f(n)** es una función matemática **f(n):N->R**$^+$, cuando el tiempo de ejecución del algoritmo puede expresarse como $T_A(n)= k*f(n)$. En este caso, el algoritmo es **simultáneamente de orden O(f(n))** y **Ω(f(n))**.

Propiedades de los órdenes de eficiencia

Transitividad: $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$. Ídem para Θ y Ω.

Reflexiva: $f(n) \in O(f(n))$. Ídem para Θ y Ω.

Simétrica: $f(n) \in \Theta(g(n))$ sii $g(n) \in \Theta(f(n))$.

Suma: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) + T2(n) \in O(\text{máx}\{f(n),\ g(n)\})$.

Producto: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) \times T2(n) \in O(f(n) \times g(n))$

Ejemplo. Demostración de la transitividad

Normalmente, podemos usar el **Principio de Invarianza** para las demostraciones.

Transitividad: f(n) es O(g(n)) y g(n) es O(h(n))**->** f(n) es O(h(n))

f(n) es O(g(n)) **->** f(n)<=K1·g(n), para al menos un K1 positivo

g(n) es O(h(n)) **->** g(n)<=K2·h(n), para al menos un K2 positivo

Entonces, sustituyendo g(n) por K2·h(n) tenemos:

f(n)<=K1·g(n)<=K1·K2·h(n)=K3·h(n) **->** f(n) es O(h(n)), con K3=K1·K2

Otras propiedades de los órdenes de eficiencia

- **Regla del máximo:**

$$O(f(n) + g(n)) = \max\{ O(f(n)) , O(g(n)) \}$$

- **Regla de la suma:**

$$O(f(n) + g(n))= O(f(n)) + O(g(n))$$

- **Regla del producto:**

$$O(f(n) * g(n))= O(f(n)) * O(g(n))$$

Ejemplo 1. Aplicación de la regla de la suma

En un programa, se ejecuta un código que tiene eficiencia $O(n^2)$ y, a continuación, un código que tiene eficiencia $O(n)$. ¿Cuál es la eficiencia del programa completo?

Por la regla de la suma: $O(f(n) + g(n)) = O(f(n)) + O(g(n))$

Por tanto: $O(n^2) + O(n) = O(n^2+n)$ **->** El programa tiene eficiencia $O(n^2+n)$

Ejemplo 2. Aplicación de la regla del máximo

Un programa se ejecuta en un tiempo igual a $T(n)=n^2+n$, ¿cuál es su eficiencia?

Por el principio de invarianza: $T(n)<=K\cdot(n^2+n)$ **->** Nos vale cualquier $K>=1$

Entonces: El orden del programa es $O(n^2+n)$

Por la regla del máximo: $O(f(n) + g(n)) = \max\{ O(f(n)) , O(g(n)) \}$

Por tanto: $O(n^2+n)= \max\{O(n^2), O(n)\}$ **-> El programa es $O(n^2)$**

Ejemplo 3. Aplicación de la regla del producto

Un programa consiste en un bucle cuya eficiencia es **O(n)**. Dentro del bucle, en cada iteración se ejecuta un código cuya eficiencia es **O(n^2)** . ¿Cuál es la eficiencia del programa?

En total, si la ejecución del bucle es O(n) y en cada iteración se ejecuta el cuerpo del bucle, que es O(n^2), en total se realizan O(n)* O(n^2) operaciones.

Por la regla del producto: O(f(n) * g(n)) = O(f(n)) * O(g(n))

Por tanto: O(n)* O(n^2) =O(n*n^2)= O(n^3) **-> El programa es O(n^3)**

Órdenes con varios parámetros

En ocasiones, nos encontramos que el tamaño del problema no depende de una única variable **n**, sino de varias.

En estos casos, se analiza de igual forma que en el caso de una variable, considerando que la función **f** tiene varias variables.:

$$A \text{ es } O(f(n,m)) \leftrightarrow \exists K \in \Re^{+} : T_A(n,m) \leq K \cdot f(n,m) \overset{\infty}{\forall} n, m$$

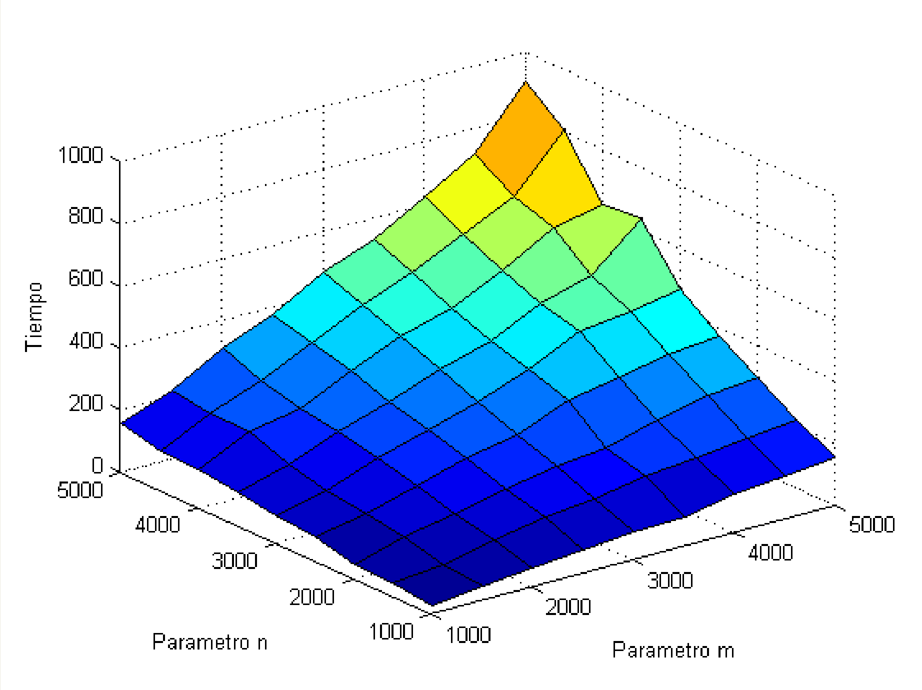
Se dice que el algoritmo será de orden **O(f(n,m))** si, para cualquier valor “muy grande” de **n** y **m**, su tiempo de ejecución $\mathbf{T_A(n,m)}$ es menor o igual que una constante positiva por **f(n,m)**.

Ejemplo. Suma de matrices de **n** filas y **m** columnas

$$A_{n,m} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

$$B_{n,m} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix}$$

$$A_{n,m} + B_{n,m} = C_{n,m};\ c_{ij} = a_{ij} + b_{ij}$$



Orden de eficiencia del algoritmo de suma de matrices: **O(n*m)**, con **n=nº filas** y **m=nº columnas** de las matrices.

1. El concepto de algoritmo.
2. La eficiencia de algoritmos: Problema, tamaño, instancia. Principio de Invarianza.
3. La notación asintótica. Órdenes peor, mejor y exacto.
4. Análisis de algoritmos.
5. Resolución de recurrencias.

Calcular de forma teórica la función **f(n)** del orden **O** de un algoritmo

1. Analizar el problema y **detectar de qué variables/parámetros depende el tamaño** del mismo.
   - Ejemplo: Para el problema de ordenación de un vector, su tiempo de ejecución dependerá del tamaño del vector a ordenar.
   - Ejemplo: Para el problema de la suma de matrices, su tamaño del caso depende del número de filas y de columnas de las matrices a sumar.
2. Aplicar las reglas de análisis de eficiencia de las operaciones en el algoritmo.

El análisis de la eficiencia de los algoritmos se basa en el tipo de sentencias que encontremos:

- Operaciones elementales
- Sentencias condicionales
- Sentencias repetitivas
- Secuencias de sentencias
- Llamadas a funciones no recursivas
- Llamadas a funciones recursivas

Sentencias simples / Operaciones elementales

- Las sentencias simples son aquellas cuya ejecución no depende del tamaño del caso (operaciones sobre tipos básicos como suma, multiplicación, comparaciones, operaciones booleanas, entrada/salida de datos desde teclado/fichero o hacia consola/fichero, etc.)

- Ejemplos:

```
x= x+8;
cin>>x;
ofstream fich;
fich.open("mifichero.txt");
```

Cálculo de eficiencia de operaciones elementales

El tiempo de ejecución de una operación elemental (sentencias simples) está acotado por una constante, y **su orden es O(1)**.

Sentencias condicionales

- Constan de la evaluación de una condición, la ejecución de un bloque de sentencias en caso de cumplirse la condición y, opcionalmente, otro bloque de código a ejecutar en caso de que no se cumpla la condición.

```
if (EvaluacionCondicion) {
        BloqueSentencias1;
} else {
    BloqueSentencias2;
}
```

Cálculo de eficiencia de sentencias condicionales (caso peor)

- El tiempo de ejecución en el caso peor de una sentencia condicional está acotado por:

max{O(EvaluacionCondicion),

O(BloqueSentencias1),

O(BloqueSentencias2) }

Cálculo de eficiencia de sentencias condicionales (caso mejor)

- El tiempo de ejecución en el **caso mejor** de una sentencia condicional está acotado por:

Ω(EvaluacionCondicion)+

mín{ Ω(BloqueSentencias1),

Ω(BloqueSentencias2) }

Ejemplo. Proyecto Code::Blocks SentenciasCondicionales, Tema 1

EvaluacionCondición (n%2==1)-> O(1), Ω(1)

BloqueSentencias1 (cout) -> O(1), Ω(1)

BloqueSentencias2 (for) -> O(n), Ω(n)

```
#include <iostream>

using namespace std;

int main()
{

    int n;

    cout<<"Dime un número: ";
    cin>>n;


    if (n%2 == 1)
        cout<<"Es impar";
    else {
        for (int i= 1; i<=n; i++)
            cout<<i;
    }
}
```

Eficiencia de la sentencia condicional:

Max{O(1), O(1), O(n)}->O(n)

Ω(1)+Mín{ Ω(1), Ω(n)}-> Ω(1)

La sentencia condicional tiene eficiencia O(n). En el peor de los casos, tardará O(n) en ejecutarse.

La sentencia condicional tiene eficiencia Ω(1). En el mejor de los casos, tardará Ω(1) en ejecutarse.

Sentencias repetitivas

- Constan de la evaluación de una condición y la ejecución de un bloque de sentencias, mientras que dicha condición se cumpla.

Mientras (Condición), **hacer**:
BloqueSentencias

Cálculo de eficiencia de sentencias repetitivas

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.

Entonces la eficiencia será:

$$O(\,g(n) + h(n)*(g(n)+f(n))\,)$$

Cálculo de eficiencia de sentencias repetitivas

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.

Entonces la eficiencia será:

$$\mathbf{O(\ g(n) + h(n)*(g(n)+f(n))\ )}$$

La evaluación de la condición se realiza al menos una vez

Sentencias repetitivas

Mientras (Condición), **hacer**:
BloqueSentencias

Cálculo de eficiencia de sentencias repetitivas

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.

Entonces la eficiencia será:

$$\mathbf{O(\ g(n) + h(n)*(g(n)+f(n))\ )}$$

En cada iteración, se ejecuta el bloque de sentencias y después se vuelve a evaluar la condición.

Sentencias repetitivas

Mientras (Condición), **hacer**:
BloqueSentencias

Cálculo de eficiencia de sentencias repetitivas

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.

Entonces la eficiencia será:

O(g(n) + h(n)*(g(n)+f(n)))

Y todo ello se realiza un total de h(n) veces.

Sentencias repetitivas

Mientras (Condición), **hacer**:
BloqueSentencias

Ejemplo.

EvaluacionCondición (n>0)-> g(n)= 1

BloqueSentencias -> f(n)= 1

Repeticiones -> n

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        cout<<n;
        n--;
    }

    return 0;
}
```

Eficiencia de la sentencia repetitiva:

g(n) + h(n)*(g(n)+f(n)) = 1+n*(1+1)= 2*n+1 -> O(2*n+1)

Aplicando la **regla del máximo**: **O(2*n+1)= max{O(2*n), O(1)}= O(2*n)**

Simplificando la constante: La sentencia repetitiva es **O(n)**

Sentencias repetitivas: Bucles for en C++/Java

Constan de la evaluación de una condición, la ejecución de un bloque de sentencias y una actualización, mientras que dicha condición se cumpla.

for (Inicialización; Condición; Actualización), **hacer:**
    BloqueSentencias

Cálculo de eficiencia de bucles for

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.
- La actualización tenga eficiencia a(n)
- La inicialización tenga eficiencia i(n)

Entonces la eficiencia será:

$$O(\, i(n)+g(n) + h(n)*(g(n)+f(n)+a(n)) \,)$$

Cálculo de eficiencia de sentencias repetitivas (bucles for)

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.
- La actualización tenga eficiencia a(n)
- La inicialización tenga eficiencia i(n)

Entonces la eficiencia será:

$$O(\ i(n)+g(n)\ +\ h(n)*(g(n)+f(n)+a(n))\ )$$

La inicialización se ejecuta siempre. La condición, al menos una vez

Sentencias repetitivas

for (Inicialización; Condición; Actualización), **hacer:**
BloqueSentencias

Cálculo de eficiencia de sentencias repetitivas (bucles for)

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.
- La actualización tenga eficiencia a(n)
- La inicialización tenga eficiencia i(n)

Entonces la eficiencia será:

O(i(n)+g(n) + h(n)*(g(n)+f(n)+a(n)))

En cada iteración se ejecuta el bloque de sentencias, luego la actualización y finalmente la evaluación de la condición.

Sentencias repetitivas

for (Inicialización; Condición; Actualización), **hacer:**
BloqueSentencias

Cálculo de eficiencia de sentencias repetitivas (bucles for)

Suponiendo que:

- El bloque de sentencias tenga eficiencia f(n),
- La evaluación de la condición tenga eficiencia g(n)
- El bucle se ejecute h(n) veces.
- La actualización tenga eficiencia a(n)
- La inicialización tenga eficiencia i(n)

Entonces la eficiencia será:

$$O(\, i(n)+g(n) + h(n)*(g(n)+f(n)+a(n))\,)$$

Y todo ello se ejecuta h(n) veces

Sentencias repetitivas

for (Inicialización; Condición; Actualización), **hacer:**
BloqueSentencias

Ejemplo (I).

Comenzaremos analizando el bucle interno.

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo (I).

Comenzaremos analizando el bucle interno.

Inicialización: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo (I).

Comenzaremos analizando el bucle interno.

Inicialización: O(1)

Condición: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo (I).

Comenzaremos analizando el bucle interno.

Inicialización: O(1)

Condición: O(1)

Actualización: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo (I).

Comenzaremos analizando el bucle interno.

Inicialización: O(1)

Condición: O(1)

Actualización: O(1)

Bloque de sentencias: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo (I).

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Comenzaremos analizando el bucle interno.

Inicialización: O(1)

Condición: O(1)

Actualización: O(1)

Bloque de sentencias: O(1)

Veces que se ejecuta: $\log_2(n)$

En la iteración 1, i=1

En la iteración 2, i=2

En la iteración 3, i=4; en la 4, i=8, … total: log2(n) hasta que i>n

Ejemplo (I).

Comenzaremos analizando el bucle interno.

Eficiencia del bucle for:

$$\mathbf{O(1)+O(1)+O(\log_2(n))*(O(1)+O(1)+O(1))=}$$

$$\mathbf{O(\log_2(n))}$$

```cpp
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

$\mathbf{O(\log_2(n))}$

Ejemplo (II).

Ahora pasamos al bucle externo (while):

Condición: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

$O(\log_2(n))$

Ejemplo (II).

Ahora pasamos al bucle externo (while):

Condición: O(1)

Bloque sentencias: O($\log_2$(n) + 1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

O($\log_2$(n))

Ejemplo (II).

Ahora pasamos al bucle externo (while):

Condición: O(1)

Bloque sentencias: O(log$_2$(n) + 1)

Veces que se ejecuta: n

Total: 1+n*(1+(log$_2$(n) + 1))= O(n*log$_2$(n))

El bucle while tiene eficiencia: **O(n*log$_2$(n))**

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

O(log$_2$(n))

Secuencias de sentencias

- Constan de la ejecución secuencias de bloques de sentencias.

Sentencia 1;
Sentencia 2;
…
Sentencia S;

Cálculo de eficiencia de secuencias de sentencias

Asumiendo que cada sentencia **i** tiene eficiencia $\mathbf{O(f_i(n))}$, la eficiencia de la secuencia se obtiene aplicando las reglas de la suma y del máximo:

$$\mathbf{O(\ f_1(n)+f_2(n)+\ldots+f_S(n)\ )=\max\{\ O(f_1(n)),\ O(f_2(n)),\ \ldots,\ O(f_S(n))\ \}}$$

Ejemplo.

Primera sentencia: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo.

Primera sentencia: O(1)

Segunda sentencia: O(1)

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo.

Primera sentencia: O(1)

Segunda sentencia: O(1)

Tercer bloque de sentencias: O(n*log2(n))

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Ejemplo.

Primera sentencia: O(1)

Segunda sentencia: O(1)

Tercer bloque de sentencias: O(n*log2(n))

Total: Max{O(1), O(1), O(n*log2(n))}

Eficiencia del programa: **O(n*log2(n))**

```
int main()
{
    int n;

    cout<<"Dime un número: ";
    cin>>n;

    while (n>0) {
        for (int i= 1; i<=n; i*=2)
            cout<<i;
        n--;
    }

    return 0;
}
```

Eficiencia de una función

- La eficiencia de una función es el máximo entre las eficiencias de las sentencias que la componen.

Eficiencia de una **llamada a función**

- La eficiencia de **una llamada a función** dependerá de si sus parámetros de entrada dependen o no del tamaño del problema.

Pregunta.

¿Cuál es la eficiencia de la función?

```
bool esPrimo(int valor) {

    double tope= sqrt(valor);
    for (int i= 2; i<=tope; i++) {

        if (valor%i == 0)
            return false;
    }

    return true;
}
```

Pregunta.

¿Cuál es la eficiencia de los códigos siguientes?

¿ Alguno es O(n²)?
¿ Alguno es O(n·√n)?
¿ Alguno es O(1) ?
¿ Alguno es O(n)?
¿Alguno es O(log(n))?
¿Alguno es O(n*log(n))?
¿Alguno es O(√n*log(n))?

```
for (int i= 1; i<n; i++)
    if (esPrimo(1234567))
        cout<<i;
```

```
for (int i= 1; i<n; i++)
    if (esPrimo(i))
        cout<<i;
```

```
for (int i= 1; i<2000; i++)
    if (esPrimo(i))
        cout<<i;
```

```
for (int i= n; i>0; i/=2)
    if (esPrimo(i))
        cout<<i;
```

Moraleja: Nos tenemos que fijar muy bien en cuáles son las variables de las que depende el tamaño del caso, y también en el número de veces que se ejecuta cada bucle.

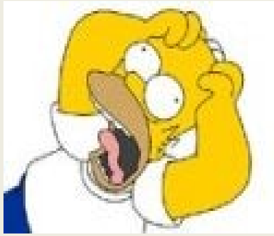
Pregunta. ¿Cuál es la eficiencia del siguiente código?

```
void Ejemplo(int *v, int N) {
    for (int i= 0; i<N; i++) {
        v[i]= (i*2+20-4*i)/N;
        v[i]= LlamarV(v, N-1)*LlamarV(v, N-2);
    }
}

int LlamarV(int *s, int N) {
    for (int i= N-1; i>0; i= i/2)
        V[i]= V[i]-1;
    return V[0];
}
```

¿Quién vota por…

- $O(n^3)$?
- $O(n^2)$?
- $O(n)$?
- $O(n*\log(n))$?
- $O(n*\log^2(n))$?



Ejercicio: Analizar la eficiencia del siguiente algoritmo

```
void ejemplo1 (int n)
{
  int i, j, k;

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

Ejercicio: Analizar la eficiencia del siguiente algoritmo

```
bool esPalindromo(char v[]) {
  bool pal= true; // Suponemos que lo es
  int inicio= 0, fin= strlen(v)-1; // Inicio y fin de la cadena

  while ((pal) && (inicio<fin)) {

        if (v[inicio] != v[fin])
              pal= false;
        inicio++;
        fin--;
  }
  return pal;
}
```

Ejercicio: Analizar la eficiencia del siguiente algoritmo

```
void F(int num, int num2) {

        for (int i= num; i<=num2; i*=2) {
                cout<<i<<endl;
        }
}
```

Ejercicio: Analizar la eficiencia del siguiente algoritmo

```
void F(int *v, int num, int num2) {

    int i= -1, j= num2;

    while (i<=j) {
        do {
            i++; j--;
        } while (v[i]<v[j]);
    }

}
```

Eficiencia de una función recursiva

- Se calcula el tiempo de ejecución **T(n)** de la función con respecto al tamaño **n** del caso del problema, considerando el tamaño que resuelven las llamadas recursivas.
- Se expresa como una **ecuación en recurrencias**.
- Se resuelve la ecuación en recurrencias para calcular el orden de eficiencia.

Ejemplo.

¿Cuál es la eficiencia de la función?

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

Variable de la que depende el tamaño del problema: *n*

Ejemplo.

¿Cuál es la eficiencia de la función?

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

Variable de la que depende el tamaño del problema: *n*

Tiempo que tarda en ejecutarse la función: T(n)

Evaluación de la condición: O(1)

Ejemplo.

¿Cuál es la eficiencia de la función?

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

Variable de la que depende el tamaño del problema: *n*

Tiempo que tarda en ejecutarse la función: T(n)

Evaluación de la condición: O(1)

Tiempo el bloque *if*: O(1)

Ejemplo.

¿Cuál es la eficiencia de la función?

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

Variable de la que depende el tamaño del problema: *n*

Tiempo que tarda en ejecutarse la función: T(n)

Evaluación de la condición: O(1)

Tiempo del bloque *if*: O(1)

Tiempo del bloque *else*: O(1)+*lo que tarde la función en resolver el problema de tamaño n: T(n-1)* -> 1+T(n-1)

Eficiencia de una función recursiva

- **Casos base**: No depende de la recursividad.
- **Casos generales:** Depende de la recursividad.
- **Caso mejor:** El caso general más favorable.
- **Caso peor:** El caso general más desfavorable.

Ejemplo.

¿Cuál es la eficiencia de la función?

$$T(n) = \begin{cases} 1 & n <= 1 \text{(caso base)} \\ 1 + T(n-1) & n > 1 \text{(caso general)} \end{cases}$$

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

Ejemplo.

```
6  int BusquedaBinaria(double *v, int posini, int posfin, double aBuscar) {
7
8      int centro= (posini+posfin)/2;
9      if (posini>posfin) return -1;
10     else if (v[centro] == aBuscar) return centro;
11     else if (aBuscar < v[centro])
12        return BusquedaBinaria(v, posini, centro-1, aBuscar);
13     else
14        return BusquedaBinaria(v, centro+1, posfin, aBuscar);
15 }
```

Variables de las que depende el problema: *n= posfin-posini+1*

Tiempo de ejecución de la función: T(n)

Líneas 8-10: O(1)

Líneas 11-14: max{condición, Bloque-if, Bloque-else}=

max{O(1), T(n/2), T(n/2)}

Ejemplo.

```
int BusquedaBinaria(double *v, int posini, int posfin, double aBuscar) {

    int centro= (posini+posfin)/2;
    if (posini>posfin) return -1;
    else if (v[centro] == aBuscar) return centro;
    else if (aBuscar < v[centro])
       return BusquedaBinaria(v, posini, centro-1, aBuscar);
    else
       return BusquedaBinaria(v, centro+1, posfin, aBuscar);
}
```

Casos base:

- **T(n)= O(1) si n<1 (primer *if*)**
- **T(n)= O(1) si v[n/2]= aBuscar**

Caso general:

- **T(n)= 1+T(n/2)**

$$T(n) = \begin{cases} 1 & \text{caso base} \\ 1 + T(n/2) & \text{caso general} \end{cases}$$

Ejemplo.

```
unsigned long Fibonacci(int n) {

    if (n<=1) return n;
    return Fibonacci(n-1)+Fibonacci(n-2);

}
```

Tamaño del problema: *n*

Caso base: n<=1

Eficiencia caso base: T(n)= O(1)

Caso general:

Dos llamadas recursivas consecutivas, que resuelven los problemax de tamaño *n-1* y de tamaño *n-2*, que tardan en ejecutarse *T(n-1)* y *T(n-2)*

$$T(n) = \begin{cases} 1 & \text{caso base} \\ T(n-1) + T(n-2) & \text{caso general} \end{cases}$$

Ejemplo (I).

```
void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {

    int i= posIni;
    int j= centro;
    int k= 0;

    while (i<centro && j<=posFin) {


        if (v[i]<=v[j]) {
            vaux[k]= v[i];
            i++;
        } else {
            vaux[k]= v[j];
            j++;
        }
        k++;
    }

    while (i<centro) {
        vaux[k]= v[i];
        i++, k++;
    }
    while (j<=posFin) {
        vaux[k]= v[j];
        j++, k++;
    }

    memcpy(v+posIni, vaux, k*sizeof(double));
}
```

Ejemplo (I).

```
void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {

    int i= posIni;
    int j= centro;
    int k= 0;

    while (i<centro && j<=posFin) {


        if (v[i]<=v[j]) {
            vaux[k]= v[i];
            i++;
        } else {
            vaux[k]= v[j];
            j++;
        }
        k++;
    }

    while (i<centro) {
        vaux[k]= v[i];
        i++, k++;
    }
    while (j<=posFin) {
        vaux[k]= v[j];
        j++, k++;
    }

    memcpy(v+posIni, vaux, k*sizeof(double));
}
```

¿Qué hace este código?

Ejemplo (I).

```
void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {

    int i= posIni;
    int j= centro;
    int k= 0;

    while (i<centro && j<=posFin) {


        if (v[i]<=v[j]) {
            vaux[k]= v[i];
            i++;
        } else {
            vaux[k]= v[j];
            j++;
        }
        k++;
    }

    while (i<centro) {
        vaux[k]= v[i];
        i++, k++;
    }
    while (j<=posFin) {
        vaux[k]= v[j];
        j++, k++;
    }

    memcpy(v+posIni, vaux, k*sizeof(double));
}
```

¿Qué hace este código?

¿De qué variable/variables depende el tamaño del caso?

Ejemplo (I).

```
void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {

    int i= posIni;
    int j= centro;
    int k= 0;

    while (i<centro && j<=posFin) {


        if (v[i]<=v[j]) {
            vaux[k]= v[i];
            i++;
        } else {
            vaux[k]= v[j];
            j++;
        }
        k++;
    }

    while (i<centro) {
        vaux[k]= v[i];
        i++, k++;
    }
    while (j<=posFin) {
        vaux[k]= v[j];
        j++, k++;
    }

    memcpy(v+posIni, vaux, k*sizeof(double));
}
```

¿Qué hace este código?

¿De qué variable/variables depende el tamaño del caso?

¿Qué eficiencia tiene?

Ejemplo (I).

```
void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {

    int i= posIni;
    int j= centro;
    int k= 0;

    while (i<centro && j<=posFin) {


        if (v[i]<=v[j]) {
            vaux[k]= v[i];
            i++;
        } else {
            vaux[k]= v[j];
            j++;
        }
        k++;
    }

    while (i<centro) {
        vaux[k]= v[i];
        i++, k++;
    }
    while (j<=posFin) {
        vaux[k]= v[j];
        j++, k++;
    }

    memcpy(v+posIni, vaux, k*sizeof(double));
}
```

¿Qué hace este código?

¿De qué variable/variables depende el tamaño del caso?

¿Qué eficiencia tiene?

Solución: O(n)

Ejemplo (II).

```
void MergeSort(double *v, int posIni, int posFin, double *vaux) {

    if (posIni>=posFin) return;

    int centro= (posIni+posFin)/2;

    MergeSort(v, posIni, centro, vaux);
    MergeSort(v, centro+1, posFin, vaux);
    fusionaMS(v, posIni, centro+1, posFin, vaux);
}
```

Tamaño del problema: n= posFin-posIni+1

Caso base (*if*): T(n)= *O(1)*

Caso General: *Se hace el cálculo de centro - O(1) -, y luego dos llamadas recursivas que solucionan el problema de tamaño n/2. Finalmente, se ejecuta fusionaMS, que es O(n)*

Ejemplo (II).

```
void MergeSort(double *v, int posIni, int posFin, double *vaux) {

    if (posIni>=posFin) return;          O(1)

    int centro= (posIni+posFin)/2;       O(1)
                                                     T(n/2)
    MergeSort(v, posIni, centro, vaux);                     T(n/2)
    MergeSort(v, centro+1, posFin, vaux);
    fusionaMS(v, posIni, centro+1, posFin, vaux)            O(n)
}
```

Tamaño del problema: n= posFin-posIni+1

Caso base (*if*): T(n)= *O(1)*

Caso General: *Se hace el cálculo de centro - O(1) -, y luego dos llamadas recursivas que solucionan el problema de tamaño n/2. Finalmente, se ejecuta fusionaMS, que es O(n)*

$$T(n) = \begin{cases} 1 & \text{caso base} \\ 2T(n/2) + n & \text{caso general} \end{cases}$$

La eficiencia de los algoritmos

1. El concepto de algoritmo.
2. La eficiencia de algoritmos: Problema, tamaño, instancia. Principio de Invarianza.
3. La notación asintótica. Órdenes peor, mejor y exacto.
4. Análisis de algoritmos.
5. Resolución de recurrencias.

- Una vez planteada la ecuación en recurrencias de una función recursiva, debemos resolverla para conocer su orden de eficiencia.
- Métodos:
  - Desarrollo en series de la fórmula.
  - Método de la ecuación característica.

Ejemplo. Desarrollo en series de Factorial

```
unsigned long factorial(int n) {

    if (n<=1) return 1;
    else return n*factorial(n-1);
}
```

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

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

= 2*1 + T(n-2)= 2*1 + (1+T(n-3))=

= 3*1 + T(n-3)= 3*1 + (1+T(n-4))= …

…

= i*1 + T(n-i)= …

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

T(n)= 1*n ≤ K*n, luego **el algoritmo es O(n)**

Ejemplo. Desarrollo en series de BusquedaBinaria

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

```
int BusquedaBinaria(double *v, int posini, int posfin, double aBuscar) {

   int centro= (posini+posfin)/2;
   if (posini>posfin) return -1;
   else if (v[centro] == aBuscar) return centro;
   else if (aBuscar < v[centro])
      return BusquedaBinaria(v, posini, centro-1, aBuscar);
   else
      return BusquedaBinaria(v, centro+1, posfin, aBuscar);
}
```

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

$= 2 + 1+T(n/8)= 3+T(n/8))=\ldots$

$\ldots$

$= i + T(n/2^i)= \ldots$

$= \log_2(n) + T(n/2^{\log 2(n)})= \log_2(n) + T(1)$ **-> el algoritmo es $O(\log_2(n))$**

- El método de la ecuación característica para **resolución de ecuaciones recurrentes** nos proporciona una metodología muy organizada para obtener la eficiencia de los algoritmos de forma simple.
- Estudiaremos los siguientes casos:
    - Ecuaciones Lineales Homogéneas de coeficientes constantes.
    - Ecuaciones Lineales No Homogéneas de coeficientes constantes.
- Además tendremos en cuenta otros aspectos:
    - Cambios de variable.
    - Cambios de recorrido (rango).

Ecuaciones lineales homogéneas de coeficientes constantes (ELH)

- Son del tipo:

$$\mathbf{T(n)= a_1 \cdot T(n-1)+a_2 * T(n-2)+\ldots+a_k \cdot T(n-k)}$$

- Todos los coeficientes a_i que van multiplicando a las "T's" son constantes numéricas.

Pasos para obtener la ecuación caracteerística (en ELH)

1. Se reescribe **T(n-i)** como $\mathbf{x^i}$, y se pasan todos los términos a un lado:

$$a_0t_n + a_1t_{n-1} + a_2t_{n-2} + \ldots + a_kt_{n-k} = 0$$

2. Se saca factor común x^{n-k}:

$$(a_0x^k + a_1x^{k-1} + a_2x^{k-2} + \ldots + a_k)x^{n-k} = 0$$

Como x^{n-k} no es 0 (los tiempos no pueden ser 0), se saca de la ecuación.

Pasos para obtener la ecuación caracteerística (en ELH)

3. Ecuación característica (resultante de los 2 pasos anteriores):

$$a_0x^k + a_1x^{k-1} + a_2x^{k-2} + \dots + a_k = 0$$

4. A la anterior expresión se le denomina **polinomio característico:**

Llamaremos polinomio característico al polinomio

$$p(x)= a_0x^k + a_1x^{k-1} + a_2x^{k-2} + \dots + a_k$$

5. Por el **Teorema Fundamental del Álgebra**, sabemos que:

$$P(x)= (x-R_1)*(x-R_2)*\dots*(x-R_k)$$

Debemos calcular las raíces del polinomio.

Pasos para obtener la ecuación caracteerística (en ELH)

6. Por último, sacamos el tiempo de ejecución con la expresión siguiente:

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^{\,n} n^j$$

- **R_i** son las raíces del polinomio.
- **c_{ij}** son coeficientes constantes.
- **r** es el número de raíces distintas del polinomio característico.
- **M_i** es la multiplicidad de la raíz **R_i** del polinomio (el número de veces que **R_i** es raíz de p(x)).

Ejemplo 1: Calcular la eficiencia de Fibonacci recursivo

Ecuación en recurrencias: T(n)= T(n-1)+T(n-2)

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^n n^j$$

T(n)-T(n-1)-T(n-2)= 0

$x^n - x^{n-1} - x^{n-2} = 0$

$(x^2 - x - 1)x^{n-2} = 0$

$p(x) = x^2 - x - 1$

p(x)= (x-(1+√5)/2)*(x-(1-√5)/2)

R_1= (1+√5)/2 M_1=1

R_2= (1-√5)/2 M_2=1

$t_n = c_{10}((1+\sqrt{5})/2)^n + c_{20}((1-\sqrt{5})/2)^n$

El algoritmo es $O(((1+\sqrt{5})/2)^n)$

Ejemplo 2: Calcular la eficiencia de un programa recursivo

Ecuación en recurrencias : $T(n)= 5T(n-1)-8T(n-2)+4T(n-3)$

$T(n)-5T(n-1)+8T(n-2)-4T(n-3)=0$

$x^n-5x^{n-1}+8x^{n-2}-4x^{n-3}= 0$

$(x^3-5x^2+8x-4)x^{n-3}=0$

$p(x)= x^3-5x^2+8x-4$

$p(x)= (x-2)^2(x-1)$

$R_1= 2 \quad M_1=2$

$R_2= 1 \quad M_2=1$

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i-1} c_{ij} R_i^n n^j$$

$t_n= c_{10}2^n n^0 + c_{11}2^n n^1 + c_{20}1^n n^0$

El algoritmo es $O(n \cdot 2^n)$

Ejemplo 3: Calcular la eficiencia de un programa recursivo

Ecuación en recurrencias : T(n)= 2T(n-1)-T(n-2)

$T(n)-2T(n-1)+T(n-2) =0$

$x^n-2x^{n-1}+x^{n-2}= 0$

$(x^2-2x+1)x^{n-2}=0$

$p(x)= x^2-2x+1$

$\mathbf{p(x)= (x-1)^2}$

$\mathbf{R_1= 1 \quad M_1=2}$

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^n n^j$$

$t_n= c_{10}1^n n^0 + c_{11}1^n n^1$

El algoritmo es O(n)

Ecuaciones lineales no homogéneas de coeficientes constantes (ELNH)

- En la ecuación recurrente aparecen otros términos no resursivos (que no son "T"). Ejemplo

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

- Todos los coeficientes a_i que van multiplicando a las "T's" son constantes numéricas.
- Todos los términos no recursivos se pueden expresar como una constante $\mathbf{b_i}$ elevado a **n** que multiplica a un polinomio $\mathbf{q_i(n)}$ que depende de **n** y tiene grado $\mathbf{d_i}$.
- **Forma general de las ELNH:**

$$a_0t_n + a_1t_{n-1} + a_2t_{n-2} + \ldots + a_kt_{n-k} = b_1^nq_1(x) + b_2^nq_2(x) + \ldots + b_z^nq_z(x)$$

- **Ejemplo para T(n)= T(n-1)+1**

  Basta con hacer $b_1=1$ y $q_1(n)=1$ y tenemos $\mathbf{T(n)= T(n-1)+1^n \cdot q_1(n)}$

Pasos para obtener la ecuación caracteerística (en ELNH)

1. Se pasan todos los términos **recurrentes** a un lado:

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + \ldots + a_kT(n-k)_{n-k} = b_1^nq_1(x) + b_2^nq_2(x) + \ldots + b_z^nq_z(x)$$

2. Se resuelve la parte recurrente como si fuera homogénea:

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + \ldots + a_kT(n-k)_{n-k} = 0$$

Y así obtendremos el polinomio característico de la parte homogénea, $\mathbf{p_H(x)}$

3. El polinomio característico de la ELNH se calcula como:

$$P(x)= p_H(x)(x-b_1)^{d1+1}(x-b_2)^{d2+1}\ldots(x-b_z)^{dz+1} = 0$$

4. Se aplica la fórmula para cálculo de la eficiencia de la forma usual.

Ejemplo 1: Cálculo de eficiencia de T(n)= T(n-1)+1

T(n)-T(n-1) =1

Parte Homogénea:

T(n)-T(n-1)= 0

$p_H(x)= x-1$

Parte No Homogénea:

Hacemos $1 = b_1^n \cdot q_1(n)$; entonces $b_1=1$, $q_1(n)=1$ con grado $d_1=0$

$p(x)= (x-1)\cdot(x-b_1)^{d1+1}=(x-1)\cdot(x-1)^1=(x-1)^2$

$R_1= 1$ $M_1=2$

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i-1} c_{ij} R_i^n n^j$$

$$t_n= c_{10}1^n n^0 + c_{11}1^n n^1$$

El algoritmo es O(n)

Ejemplo 2: Cálculo de eficiencia de T(n)= T(n-1)+n

T(n)-T(n-1) =n

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^n n^j$$

Parte Homogénea:

T(n)-T(n-1)= 0

$t_n = c_{10}1^n n^0 + c_{11}1^n n^1 + c_{12}1^n n^2$

$p_H(x)= x-1$

El algoritmo es $O(n^2)$

Parte No Homogénea:

Hacemos n = $b_1^n \cdot q_1(n)$; entonces $b_1=1$, $q_1(n)=n$ con grado $d_1=1$

$p(x)= (x-1)\cdot(x-b_1)^{d1+1}=(x-1)\cdot(x-1)^2=(x-1)^3$

$R_1= 1$ $M_1=3$

Ejemplo 3: Cálculo de eficiencia de $T(n)= T(n-1)+n+3^n$

$T(n)-T(n-1) = n + 3^n$

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^n n^j$$

Parte Homogénea:

$T(n)-T(n-1)= 0$

$t_n= c_{10}1^n n^0 + c_{11}1^n n^1 + c_{12}1^n n^2 + c_{20}3^n n^0$

$p_H(x)= x-1$

El algoritmo es $O(3^n)$

Parte No Homogénea:

Hacemos $n = b_1^n \cdot q_1(n)$; entonces $b_1=1$, $q_1(n)=n$ con grado $d_1=1$

Hacemos $3^n = b_2^n \cdot q_2(n)$; entonces $b_2=3$, $q_2(n)=1$ con grado $d_2=0$

$p(x)= (x-1)\cdot(x-b_1)^{d1+1}\cdot(x-b_2)^{d2+1}=(x-1)\cdot(x-1)^2\cdot(x-3)^1=(x-1)^3\cdot(x-3)$

$R_1= 1$ $M_1=3$; $R_2= 3$ $M_2=1$

Cambio de variable

Cuando **T(n) no está expresado en función de T(n-k)**, es necesario hacer un cambio de variable para que esto sea así.

El polinomio característico se expresa en función de la nueva variable.

Los cambios de variable más normales que haremos son $n=2^m$, $n=3^m$, $n=4^m$, $n=\log(m)$, etc.

¡No olvidar deshacer el cambio!

Ejemplo 1: Cálculo de eficiencia de BusquedaBinaria

Ecuación en recurrencias $T(n)= T(n/2)+1$

Cambio $n=2^m$; luego $m=\log_2(n)$

$T(2^m)= T(2^{m-1})+1$

$T(2^m)- T(2^{m-1})=1$ (**ELNH)**

Parte Homogénea:

$T(2^m)- T(2^{m-1})= 0$

$p_H(x)= x-1$

Parte No Homogénea:

Hacemos $\mathbf{1 = b_1^m \cdot q_1 (m)}$**; entonces** $\mathbf{b_1=1, q_1(m)=1}$ **con grado** $\mathbf{d_1=0}$

$\mathbf{p(x)= (x-1)\cdot(x-b_1)^{d1+1}= (x-1)^2 \;\; ; \;\; R_1= 1 \;\; M_1=2}$

$$t_m = \sum_{i=1}^{r} \sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j$$

$t_m= c_{10}1^m m^0 + c_{11}1^m m^1$

Deshacemos el cambio

$t_n= c_{10}1^{\log(n)}\log(n)^0 + c_{11}1^{\log(n)}\log(n)^1$

El algoritmo es O(log(n))

Ejemplo 2: Cálculo de eficiencia de MergeSort

Ecuación en recurrencias $T(n)= 2*T(n/2)+n$

Cambio $n=2^m$; luego $m=\log_2(n)$

$$t_m = \sum_{i=1}^{r} \sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j$$

$T(2^m)= 2*T(2^{m-1})+2^m$

$T(2^m)- 2*T(2^{m-1})= 2^m$ **(ELNH)**

$t_m= c_{10}2^m m^0 + c_{11}2^m m^1$

Parte Homogénea:

Deshacemos el cambio

$T(2^m)- 2*T(2^{m-1})= 0$

$t_n= c_{10}n\cdot\log(n)^0 + c_{11}n\cdot\log(n)^1$

$p_H(x)= x-2$

El algoritmo es O(n·log(n))

Parte No Homogénea:

Hacemos $2^m = b_1^m\cdot q_1(m)$**; entonces** $b_1=2$, $q_1(m)=1$ **con grado** $d_1=0$

$p(x)= (x-2)\cdot(x-b_1)^{d1+1}= (x-2)^2$; $R_1= 2$ $M_1=2$

Cambio de recorrido/rango

Cuando **la ecuación en recurrencias no es lineal**, debemos transformarla a lineal (si es posible). Este cambio se denomina de recorrido o rango. Normalmente, los cambios que haremos serán del tipo U(n)= log(T(n)).

¡No olvidar deshacer el cambio!

Ejemplo de ecuaciones no lineales:

$$\mathbf{T(n)= 2^n \cdot T(n-1)}$$

$$\mathbf{T(n)= T^2(n-1)}$$

Ejemplo 1: Cálculo de eficiencia con cambio de rango

Ecuación en recurrencias $T(n)= T^2(n-1)$

$\log(T(n))= \log(T^2(n-1))=2\cdot\log(T(n-1))$

Cambio $U(n)=\log(T(n))$

$U(n)= 2\cdot U(n-1)$

$U(n)-2\cdot U(n-1)=0$ (**ELH**)

$p(x)= x-2$

$\mathbf{R_1= 2 \quad M_1=1}$

$$U_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i-1} c_{ij} R_i^n n^j$$

$U_n= c_{10}2^n n^0 = c_{10}2^n$

Deshacemos el cambio T(n)= 2^U(n)

$t_n= 2\text{^}(c_{10}2^n)$

El algoritmo es $O(2\text{^}(c_{10}2^n))$

Ejemplo 2: Cálculo de eficiencia con cambio de rango

Ecuación en recurrencias $T(n)= 2^n T^2(n-1)$

$T(n)= 2^n T^2(n-1)$ -> $\log_2(T(n))= \log_2(2^n T^2(n-1))$

$\log_2(T(n))= \log_2(2^n) + \log_2(T^2(n-1))$ -> $U(n)= \log_2(T(n))$

$U(n)= n + 2U(n-1)$, que sí sabemos resolver:

$U(n)-2U(n-1)= n$

$(x-2)(x-1)^2= 0$

$U(n)= c_{10}2^n+c_{20}+c_{21}n$ -> $T(n)= 2\wedge(c_{10}2^n+c_{20}+c_{21}n)$

El algoritmo es $O(2\wedge(c_{10}2^n+c_{20}+c_{21}n))$

Ejercicios

Resolver las siguientes ecuaciones en recurrencias:

$T(n)= 4T(n-1)-4T(n-2)$

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

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

$T(n)= 5T(n-1)+6T(n-2)+4*3^n$

$T(n)= 2T(n/2)+log2(n)$

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

UNIVERSIDAD DE GRANADA

DECSAI

Algorítmica

Grado en Ingeniería Informática

Tema 1 – La eficiencia de los algoritmos

Manuel Pegalajar Cuéllar
manupc@ugr.es
Departamento de Ciencias de la Computación e Inteligencia Artificial
http://decsai.ugr.es