**TECNICAS DE PROGRAMACION AVANZADAS**

Contenido

[**Tema 1 Eficiencia de los algoritmos** 2](#_Toc73608481)

[**1.1 Criterio para medir la eficiencia** 2](#_Toc73608482)

[**1.2 Tiempo de ejecución** 2](#_Toc73608483)

[**1.3 Orden de magnitud** 2](#_Toc73608484)

[**1.4 Comparativa de los órdenes de magnitud** 4](#_Toc73608485)

[**1.5 Reglas para calcular complejidad del código** 5](#_Toc73608486)

[**1.6 Algoritmo recursivo** 6](#_Toc73608487)

[**Análisis de algoritmos** 7](#_Toc73608488)

[**Tema 2 Divide y vencerás** 10](#_Toc73608489)

[**1.1 Planteamiento general** 10](#_Toc73608490)

[**1.2 Determinar caso base** 11](#_Toc73608491)

[**1.3 Ejemplos** 12](#_Toc73608492)

[**1.4 Algoritmos de ordenación** 12](#_Toc73608493)

[**1.4.1 Ordenación por mezcla (Mergesort)** 12](#_Toc73608494)

[**TEMA 3 TABLAS HASH** 13](#_Toc73608495)

[**1. Planteamiento general** 13](#_Toc73608496)

[**2. Colisiones** 13](#_Toc73608497)

[**3. Factor de carga** 14](#_Toc73608498)

# **Tema 1 Eficiencia de los algoritmos**

Un **algoritmo** es la descripción precisa de pasos a seguir para alcanzar la solución de un problema dado. Debe ser preciso, sin ambigüedades, determinista, finito e independiente del lenguaje de programación.

## **1.1 Criterio para medir la eficiencia**

Para un mismo problema se pueden plantear varios algoritmos correctos, pero debemos compararlos según **un criterio determinado**. Por su legibilidad, la facilidad de depuración, el uso eficiente de los recursos (tiempo de ejecución y memoria), etc. De las anteriores, la mejor para medir la eficiencia del algoritmo, es el tiempo de ejecución.

## **1.2 Tiempo de ejecución**

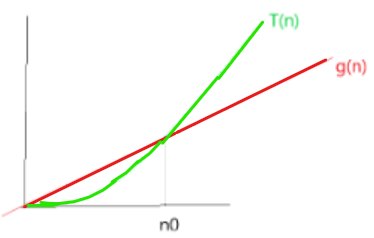
El **tiempo de ejecución** depende de varios factores, como la número de veces que se ejecuta cada instrucción o el compilador u ordenador utilizados. En el tiempo de ejecución también influye el número de datos (tamaño) de entrada, la estructura y la distribución de los datos de entrada, y la calidad del código (número de veces que se ejecuta cada instrucción). Dependiendo de estos factores, podemos tener tres casos:

* **Caso Peor**: máximo tiempo de ejecución.
* **Caso Mejor**: mínimo tiempo de ejecución.
* **Caso Medio**: caso más probable. El tiempo de ejecución es la media de todos los casos posibles.

Para comprobar la eficiencia de un algoritmo, usaremos como media el tiempo de ejecución en el peor caso.

## **1.3 Orden de magnitud**

El orden de complejidad consiste en encontrar un representante para todas la funciones que crecen de una determinada manera. Aunque todas las funciones al principio pueden crecer de formas muy diferentes, normalmente llega un punto en el que se acercan. Gracias a esto, podemos saber hasta que numero de repeticiones es mejor seguir un algoritmo u otro.

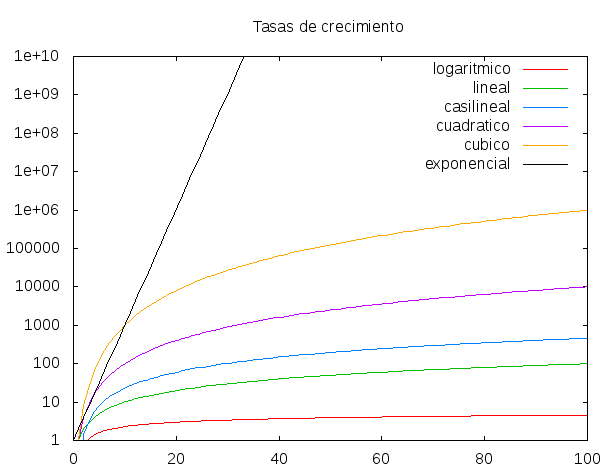
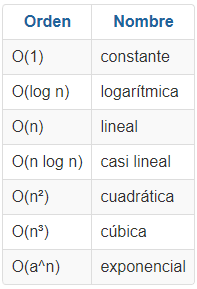
* La notación **O( f(n) )** representa los máximos recursos que necesitamos para resolver el problema de tamaño ‘n’. (Caso Peor)
* La notación **Ω( f(n) )** representa los mínimos recursos para resolver el problema de tamaño ‘n’. (Caso Mejor)
* La notación **Θ( f(n) )** representa el orden exacto, es decir el tiempo de ejecución es el mismo para el mejor caso como para el peor.

Definamos la función T(), que pertenece a la clase de complejidad f():

* Si T(n) pertenece a **O( f(n) ) 🡪** T(n) nunca crece más rápido que f(n) (Cota superior)
* Si T(n) pertenece a **Ω( f(n) ) 🡪** T(n) crece al mismo ritmo que f(n) (Cota inferior)
* Si T(n) pertenece a **Θ( f(n) ) 🡪** representa la tasa de crecimiento de T(n) (Orden Exacto)

Relación entre los distintos órdenes de magnitud de las funciones f(n)

O(1) < O(log n) < O(n) < O(n·log (n)) < O(n2) < O(n3)<...< O(2n) < O(n!) < O(nn)



Orden constante: no depende del tamaño de entrada

Logarítmica: algoritmo con iteración o recursión no estructural

Orden lineal: depende del número de datos de entrada

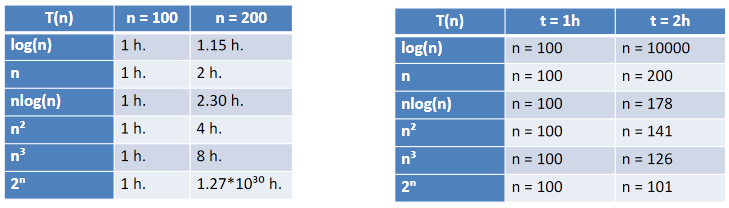
Orden cuasi-lineal: depende del tamaño de entrada y la rapidez a la hora de descartar datos.

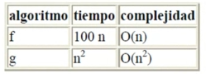
Orden cuadrático: aparece en bucle o ciclos

Orden polinomial (na), exponencial (an), factorial (n!)

## **1.4 Comparativa de los órdenes de magnitud**

Para comparar ordenes, tenemos dos opciones, aplicar el efecto de duplicar el tamaño del dato de entrada (n), o duplicar el tiempo disponible (t)



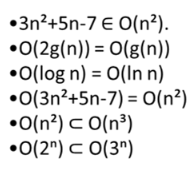
Ejemplo:

Para valores mas pequeños de 100 debemos usar el algoritmo g(502 = 2500 < 100x50 = 5000).

Para valores más grandes de 100 debemos usar el algoritmo f(1502 = 22500 > 100\*150 = 15000).

## **1.5 Reglas para calcular complejidad del código**

Las reglas intuitivas para calculo consideran el termino más importante:

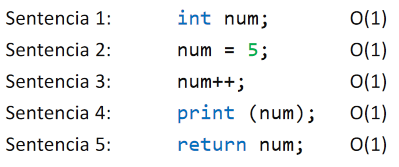
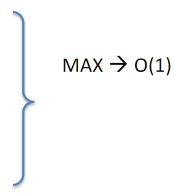


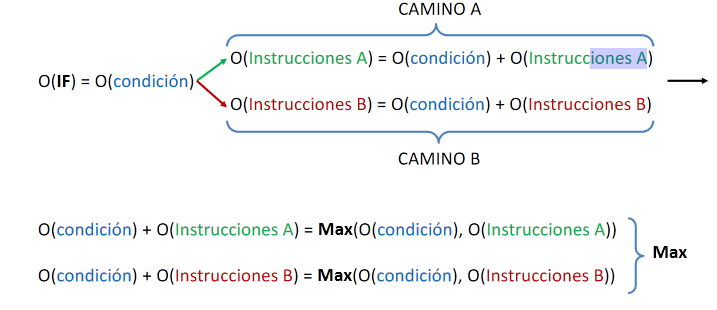
* Regla de la suma: Si f1∈O(g1) y f2 ∈O(g2) entonces f1+f2 ∈O(max(g1,g2)).

Cuando me dicen max(g1,g2) me quedo con el valor máximo entre esos dos, y ese es el orden de complejidad.

* Regla de la multiplicación: Si f1∈O(g1) y f2 ∈O(g2) entonces f1 x f2 ∈O(f(g1) x f(g2)).

Calculamos tiempo de ejecución de:

* **Sentencias simple**: manipulaciones, lectura y escritura, acceso a arrays, ficheros, etc.
  + Cada instrucción tiene un tiempo constante
  + Escogemos como unidad el mayor tiempo de las instrucciones
  + Cada sentencia simple se considera O(1), es decir, constante
* **Composición** **secuencial**: para una secuencia de sentencias (instrucciones)
  + Aplicar regla de la suma
* **Selección condicional**: el tiempo de ejecución es el máximo de complejidad de los diferentes caminos que existan.



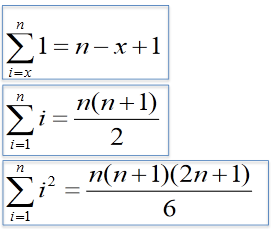
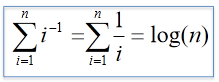
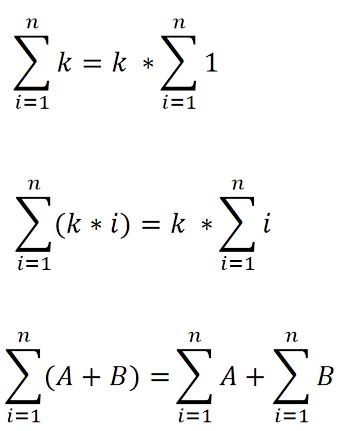
* **Repeticiones**:
  + Para el bucle for:

T(inicialización) + ∑(T(condición) + T(cuerpo) +T(incremento)) + T(condición)

T(int i = 0) + ∑(T(i<10) + T(cuerpo) +T(i++)) + T(i<10) 🡪 empezamos con la inicialización (i=0), seguimos con el bucle for, haciendo la condición, el código del bucle, e incrementando (i++). Cuando la condición no se cumple, salimos (i = 11)

* **Llamadas a otros módulos (llamadas función)**: tardo el tiempo que me cuesta hacer la invocación más el tiempo que cueste hacer esa función.

Fórmulas más habituales para resolver los sumatorios



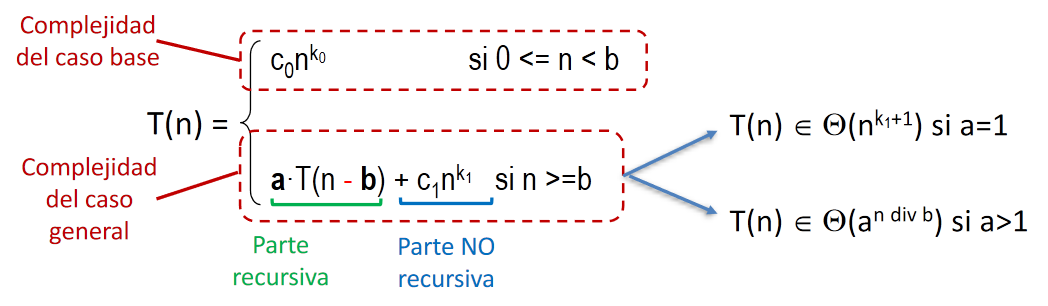
## **1.6 Algoritmo recursivo**

La resolución dependerá de cómo se vaya reduciendo N (numero)

En estos caso vamos a diferenciar dos casos posibles, el Caso Base, donde se retorna un resultado final, y el Caso General, donde la función se llama a sí misma.

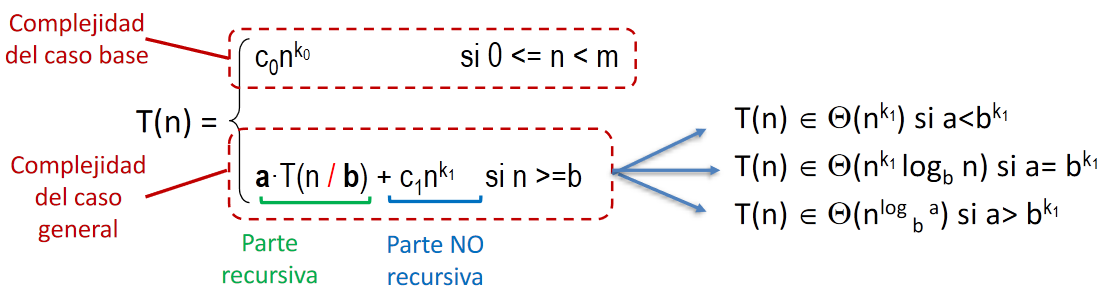
* Reducción por **sustracción**

Me acerco al caso base restando.



* c0 = complejidad del caso base
* nko = coste de las instrucciones del caso base
* Parte recursiva:
  + a = número de llamadas recursivas idénticas en cada invocación
  + b = cantidad de los elementos en los que se reduce n
* Parte no recursiva:
  + c1 = complejidad del caso general parte no recursiva
  + nk1 = coste de las instrucciones del caso general parte no recursiva
* Reducción por **división**

Me acerco al caso base dividiendo el valor n.

****

* c0 = complejidad del caso base
* nko = coste de las instrucciones del caso base
* Parte recursiva:
  + a = número de llamadas recursivas idénticas en cada invocación
  + b = valor por el que reduzco n al llamar a la función
* Parte no recursiva:
  + c1 = complejidad del caso general parte no recursiva
  + nk1 = coste de las instrucciones del caso general parte no recursiva

## **Análisis de algoritmos**

Tenemos un bucle for () donde imprimimos los números.

**1/ Bucle for básico** // for ( i=0; i<n; i++) // for( i=0; i>0; i--)

Vamos a recorrer nuestro algoritmo n veces de uno en uno.

n=10 -> consola -> 0-1-2-3-4-5-6-7-8-9

Tenemos 1asignación más n (1comparación + 1incremento) + 1comparación = 1 + n(1 + 1) + 1 = **2n +** **2**

Si n = 10 -> 1asignación + 10(1comparación + 1incremento) + 1condición -> 1+10\*2+1 = 10\*2 + 2 = 22 veces hasta completar el bucle.

**2/ Bucle for incremento distinto de uno** // for( i=0; i<n; i+=a)

Vamos a recorrer nuestro algoritmo n veces de ‘a’ en ‘a’.

n=10 y a=2 -> consola -> 0-2-4-6-8

La condición se ejecutará n/a ya que llegamos antes a ‘n’ aumentando de ‘a’ en ‘a’.

Tenemos 1asignación + n/a (1comparación + 1incremento) + 1comparación = 1 + n/a (1 + 1) + 1 = **(2n/a) + 2**

Si n=10; i+=2 -> 1asignación + 10/2(1comparación + 1incremento) + 1comparación = 1 + (10/2)(2) + 1 = (2\*10 / 2) + 2 = 12 veces hasta completar el bucle.

**3/ Bucle for con condición n/a** // for(i=0; i<n/a; i++)

Vamos a recorrer nuestro algoritmo n/a veces de uno en uno

n=10 y a=2 -> consola -> 0-1-2-3-4

La condición se ejecutará n/a ya que dividimos ‘n’ entre ‘a’.

Tenemos 1asignación + n/a (1comparación + 1incremento) + 1comparación = 1 + n/a (1 + 1) + 1 = **(2n/a) + 2**

Si n=9; i+=3 -> 1asignación + 9/3(1comparación + 1incremento) + 1comparación = 1 + (9/3)(2) + 1 = (2\*9 / 3) + 2 = 8 veces hasta completar el bucle.

**4/ Bucle for con condición n/a + incremento i+=b** // for(i=0; i<n/a; i+=b)

Vamos a recorrer nuestro algoritmo n/a veces de ‘b’ en ‘b’.

n=10 y a=2 y b= -> consola -> 0-2-4

Mezcla de la 3 y la 4. Reducimos ‘a’ veces ‘n’ y aumentamos de ‘b’ en ‘b’

Tenemos 1asignación + n/a/b (1comparación + 1incremento) + 1comparación = 1 + n/a/b (1 + 1) + 1 = **(2n/a/b) + 2**

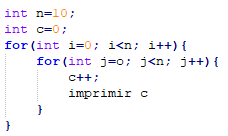
Si n=10; i<n/2; i+=2 -> 1asignación + 10/2/2(1comparación + 1incremento) + 1comparación = 1 + (10/2/2)(2) + 1 = (2\*10 / 4) + 2 = 7 veces hasta completar el bucle.

**1/ Cuando tenemos un bucle for() dentro de otro bucle for()**

* Primer bucle -> 1asignación + n (1comparación + 1incremento) + 1comparación = 1 + n(1 + 1) + 1 = **2n +** **2**
* Segundo bucle -> 1asignación + n (1comparación + 1incremento) + 1comparación = 1 + n(1 + 1) + 1 = **2n +** **2 ->** se repite tantas veces como el primero -> **(2n+2)n**

1asignación + 1bucle + nbucle 🡪 **n2**

Ejemplo

Imprime de 0 a 9 + 10 a 19 +… +90 a 99

El primer bucle se repite 10 veces. El segundo también, pero en si mismo otras 10 veces.

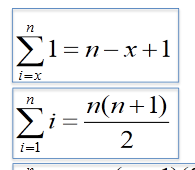
**n2** -> 102 = 100 números impresos

Calcular el tiempo

Bucle for 🡪 Tinicializacion + ∑(inicialización hasta condición) (Tcondicion+Tcuerpo+Tincremento) + Tcondición

Bucle if 🡪 Tcondicion + Max(TcaminoA, TcaminoB, ...)

Bucle while 🡪 ∑((desde primera interacción hasta las veces que se repite)( Tcondición +Tcuerpo)) + Tcondición



# **Tema 2 Divide y vencerás**

Cuando tenemos un problema con varios individuos, lo dividimos en problemas más pequeños para poder solucionarlos. Posteriormente, combinamos las soluciones obteniendo el resultado final.

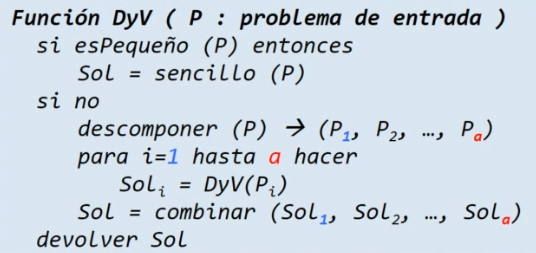
## **1.1 Planteamiento general**

Idea general:

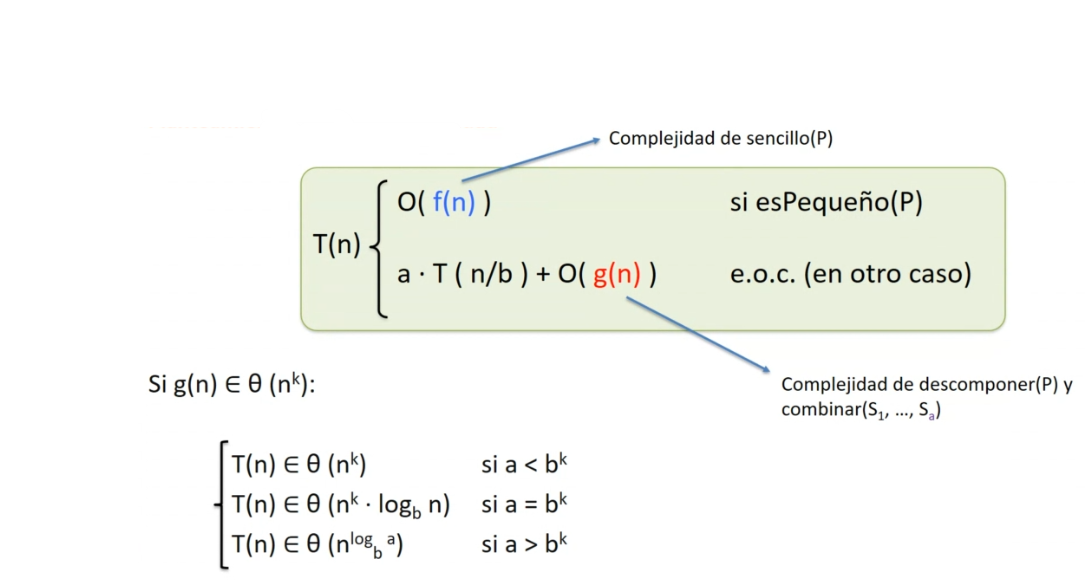
* Descomponer el problema en subproblemas más pequeños, pero iguales de tamaño.
* Resolver cada subproblema
* Combinar las soluciones
* Repetir hasta que los subproblemas tengan una solución trivial o directa.

Pseudocódigo

* **esPequeño (P):** función que decide si el problema es trivial y se puede resolver de forma directa (caso base).
* **sencillo (P):** función que resuelve el problema para un tamaño pequeño.
* **descomponer (P):** función que divide un problema en subproblemas más pequeños, hasta que estos pueden ser resueltos de forma sencilla.
* **combinar (Sol1, sol2, sola):** función que combina las soluciones de los subproblemas para obtener la solución principal.



Complejidad

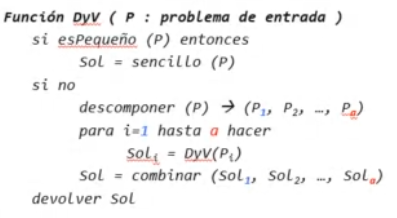


Donde **a** es el número de subproblemas y **b**, el numero en el que se divide cada problema.

## **1.2 Determinar caso base**

El umbral es el punto a partir del cual no merece la pena utilizar DyV. Se calcula haciendo que (Sencillo) sea menor que (Descomponer+DyV+Combinar), dejándolo en función de Sencillo; o viceversa, depende de lo que quieras comparar.

Ejemplo:



Sencillo 🡪 N2

(Descomponer+DyV+Combinar) 🡪 4N

¿Cuál es el umbral a partir del cual no merece la pena usar DyV?

Cuando Sencillo < DyV

N2 < 4N 🡪 N2 - 4N < 0 🡪 N<4

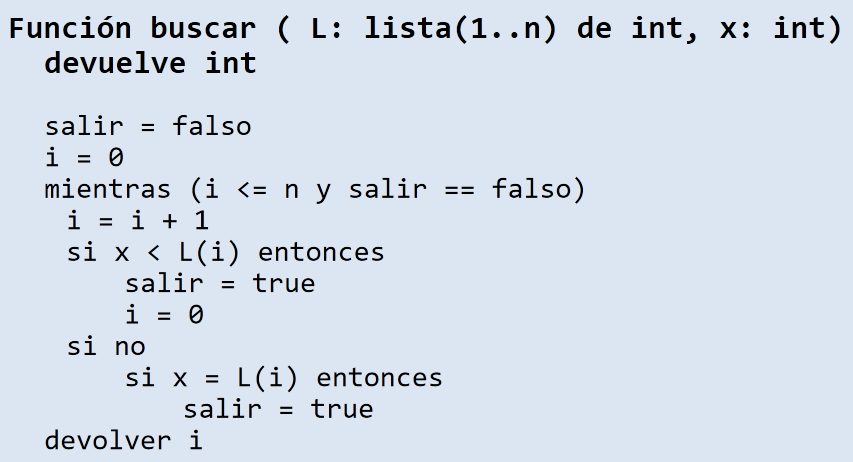
El umbral optimo dependerá de los algoritmos y las implementaciones. Para calcularlo, debemos calcular la complejidad de los algoritmos, el valor de las constantes de la implementación. Además, debemos calcular el tamaño a partir del cual el tiempo del DyV y el Sencillo coincidan.

## **1.3 Ejemplos**

**Búsqueda Binaria**

Se pide indicar la posición en la que se encuentra un dato x, en una lista ordenada ascendentemente. Si no se encuentra el número, devolvemos -1.

Solución tradicional:



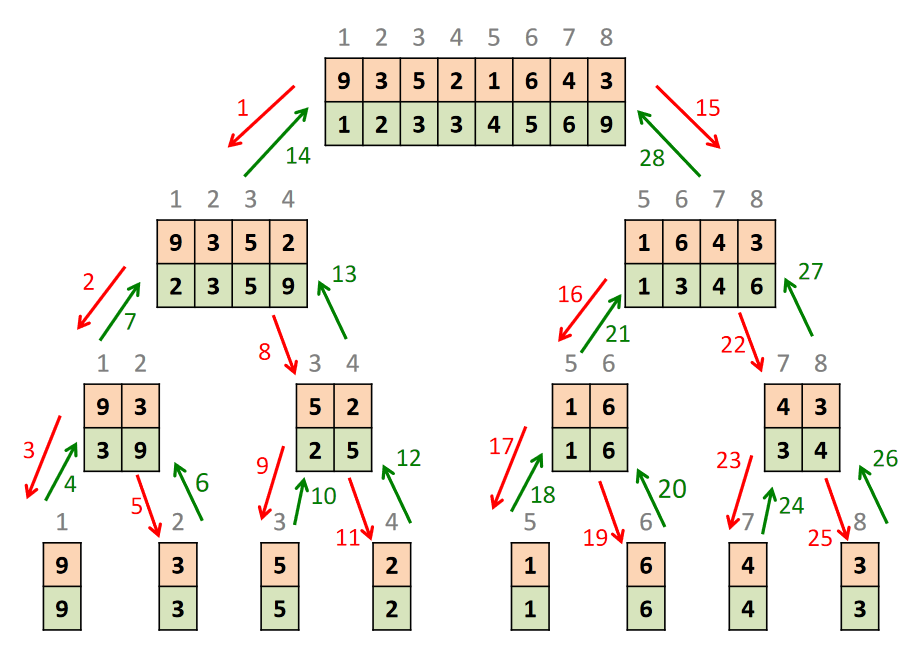
## **1.4 Algoritmos de ordenación**

Siguiendo al estrategia Dy

V, descomponemos la lista en dos partes, y ordenamos cada una de las partes por separado, para al final, combinar todas las partes ya ordenadas en una sola lista.

### **1.4.1 Ordenación por mezcla (Mergesort)**

Es un algoritmo que divide una lista por la mitad, con el objetivo de cada una de las mitades. Realiza este proceso hasta que en cada lista solo queda un elemento. En ese momento, compara y los elementos y crea una lista auxiliar con los datos ordenados.



# **TEMA 3 TABLAS HASH**

### **1. Planteamiento general**

Una tabla hash es una estructura de datos similar al array, que permite el almacenamiento de un elemento en una posición que viene determinada por el propio elemento y no por un índice. Su objetivo es optimizar el tiempo de las búsquedas e inserciones de elementos.

Cada elemento que almaceno en la tabla tiene algo que se convierte en clave. A partir de esa **clave** (el expediente de un alumno) nos devuelve un numero entero que nos diga en que posición se almacena ese elemento (alumno). Esta es la **Función Hash** (H(x)).

Para que esta función sea eficiente tiene que cumplir las siguientes características:

* Rapidez a la hora de ejecutarse. (Tiempos de orden constante).
* Asignar las posiciones de manera uniforme. Los elementos se distribuyen por todos los espacios disponibles de la tabla, en lugar de ocupar siempre las primeras.
* La clave se debe poder transformar en un tipo entero, para que podamos devolver la posición en la que se encuentra.
* El tamaño de la tabla debe ser un numero primo, ya que así las posiciones que se asignen posiciones diferentes a cada elemento (n es primo)

### **2. Colisiones**

Es una situación que se produce cuando una función hash asigna la misma posición a dos o mas elementos. Para evitar esto, tenemos dos soluciones posibles:

* **Direccionamiento abierto**: busca una nueva posición. Exploración lineal o cuadrática o Doble Hashing.
* **Encadenamiento**: organizo la tabla para poder almacenar mas de un elemento en cada posición (array de arrays).

**Exploración Lineal**: Asigno el elemento a la siguiente posición libre (array circular). La siguiente posición la busco con la función hash + los intentos necesarios para encontrar el siguiente espacio (H(x) + i). Su problema es que se generan bloques de elementos consecutivos, perjudicando la eficiencia (Clustering).

**Exploración Cuadrática**: Es igual que la Exploración Lineal, pero la siguiente posición se le asigna dando un salto (H(x) + i2). Esta exploración también tiene Clustering.

**Doble Hashing**: Tengo dos funciones Hash: H1(x) + i\*H2(x).   
H1(x) = x%longitud // H2(x) = 7-(x%7) (según el enunciado)

**Encadenamiento**: Las posiciones de la tabla pueden almacenar más de un elemento (bucket).

### **3. Factor de carga**

El factor de carga nos indica cuando es mejor ampliar la dimensión de la tabla, en vez de buscar un espacio, se calcula como el % de ocupación. Este porcentaje de ocupación se denomina Umbral del Factor de Carga. Si queremos aumentar la tabla, tenemos que buscar el doble del tamaño de la tabla y buscar el siguiente número primo, ese será el tamaño de la tabla aumentada.