2. Introducción a la eficiencia de algoritmos

2.1 Qué es la eficiencia

La eficiencia nos permite medir qué recursos (tiempo de ejecución o memoria) necesita un algoritmo. Además, nos permite comparar dos algoritmos que resuelven un problema.
Por ejemplo, en el clásico problema de la asignación, en el que tenemos 30 trabajadores y 30 trabajos, y queremos asignarle a cada trabajador un trabajo dependiendo de factores como sus habilidades, experiencia, etc, las posibilidades se elevarían hasta la cifra de 30!. Para que tenga una visualización de lo que significa 30!, $30! = 265252859812191058636308480000000$. Nada mas y nada menos que un numero con 33 cifras.

Por lo tanto en nuestro problema de la asignación,si tuviéramos un ordenador que evaluase un billón de posibilidades por segundo y hubiéramos empezado a evaluar en la era de los dinosaurios, aún estaría nuestro ordenador evaluando, por tanto, este problema no es viable resolverlo con un ordenador. Normalmente cuando tengamos que resolver un problema, es posbile que dispongamos de varios algoritmos, que siguen técnicas de diseño diferente. Nuestro objetivo será escoger el mejor. Para ello se debe establecer parámetros que nos diga cuando un algoritmo es mejor que otro. Para llevar a cabo esta selección seguiremos algunos de los siguientes enfoques:

- Enfoque empírico: programamos los diferentes algoritmos que resuelven mi problema y los haceos competitir. Para ello seleccionamos diferentes conjuntos de entrada para los algoritmos y ejecutamos el programa asociado.
- Enfoque teórico: determinamos de forma matemática (contando instrucciones en una computadora ideal) qué recursos necesita cada algoritmo (recurso aqui se entiende como tiempo de ejecución o memoria) en función del tamaño del conjunto de entrada.
- Enfoque híbrido: Se hace uso de los dos enfoques anteriores. Para ello se refina las necesidades

en recursos calculadas de forma teórica realizando una experimentación en un ordendor real.

Los recursos que más nos interesan en este libro será el tiempo de ejecución de un algoritmo y el espacio de almacenamiento. No obstante se pondrá más énfasis en el tiempo de ejecución. Antes de avanzar en los diferentes enfoques para estudiar los recursos que necesita un algoritmo, aclararemos dos conceptos que normalmente se confunden: algortimo e implementación

Algoritmo : conjunto de pasos ordenados que resuelven un problema.
Implementación : traducción de esa secuencia a un lenguaje de programación

Podemos tener por lo tanto diferentes implementaciones de un mismo algoritmo pero la eficiencia en resumen tiene que ser igual. El enunciado del *Principio de Invarianza* sería el siguiente:

Principio de invarianza: si hay dos implementaciones de un mismo algoritmo, la eficiencia va a cambiar por una constante, es decir, tienen la misma eficiencia.

2.2 Cómo calcular la eficiencia

La eficiencia de un algoritmo está en función del tamaño o *talla* del problema. Por ejemplo en el algoritmo de búsqueda secuencial (ver sección 1.2.1) la eficiencia está en función del número de elementos que tenga el vector en el que se busca un elemento. Veámos como llevar a cabo las diferentes aproximaciones.

2.2.1 Eficiencia empírica

En primer lugar se implementa el algoritmo en un lenguaje de programación. Una vez construido el programa, para diferentes tamaños de problema ejecutamos el programa y obtenemos el numero de segundos que tarda cada ejecución. Con estos datos construimos una gráfica. Esta gráfica contiene en el eje x los diferentes tamaños de problemas y en el eje y los segundos que tarda el programa para cada tamaño de problema.

Así por ejemplo en la búsqueda secuencial (ver sección 1.2.1) el tiempo de ejecución (lo que tarda nuestro algoritmo) estará en función del tamaño del vector.No obstante para cada tamaño de problema tenemos diferentes posibilidades:

- Mejor ejecución: el elemento se encuentra en la primera posición. Independiente del tamaño del vector el algoritmo se ejecuta más rápido.
- Peor ejecución: el elemento no se encuentra. Independiente del tamaño del vector el algoritmo siempre tarda más
- Caso promedio: el elemento puede encontrarse con la misma probabilidad en cualquier posición.

En C++ para calcular los segundo, usamos la librería ctime. Así dado el siguiente main que invoca a la búsqueda secuencial (ver sección 1.2.1):

```cpp
#include <ctime>
#include <iostream>
using namespace std;

int main ()
{
      VectorDinamico v;
      //rellenamos con datos v
      ...
      //Codigo para medir el tiempo
      time_t t_antes, t_despues;

      time(&t_antes); //medimos el tiempo
      //buscamos en el vector v el valor x.
      v.BusquedaSecuencial (x);
      time(&t_despues); //lo volvemos a medir
      cout << difftime(t_despues, t_antes)<<endl; //obtenemos la diferencia
      return 0;
}
```

Este método presenta varios problemas:

1. Depende de los casos ejecutados
2. Depende del hardware y librerías usadas
3. Para comprobar dos algoritmos hay que ejecutarlos en las mismas condiciones (hardware, librerías, casos...)

Este método es menos objetivo que el enfoque teórico.

2.2.2 Eficiencia teórica

La ventaja del enfoque teórico para obtener la eficiencia de un algoritmo es que no depende ni del ordenador que se usa, ni del lenguaje de programación, ni de las habilidades del programador, ni de librerias usadas, etc. Con la eficiencia teórica daremos una notación asintótica del tiempo de ejecución. Esta notación asíntotica pretende expresar una cota superior (podría ser también inferior) del tiempo que tarda nuestro algoritmo en un número infinito de tamaños de problemas.

Orden de eficiencia ($T(n)$): un algoritmo tiene un tiempo de orden $T(n)$ si existe una implementación del algoritmo cuyo tiempo de ejecución, $f(n)$, está acotado superiormente por $c \cdot T(n)$, siendo c una constante mayor que cero y existe un n_0 tal que se cumple $\forall n \;\; n \geq n_0$ (siendo n el tamaño del problema).

Es decir cualquier implementación del algoritmo tarda menos que $c \cdot T(n)$.

$$f(n) \leq c \cdot T(n)$$

Ejemplo 2.2.1

Sea la función $f(n) = 2n+3$. Obtener el valor de n y la c para que $f(n) \leq c \cdot n$. En este caso $T(n) = n$. Para demostrar que

$$2n+3 \leq c \cdot n$$

Igualamos para saber el punto exacto en el que se cruzan las funciones:

$$2n+3 = c \cdot n$$

Despejamos:

$$2n - c \cdot n + 3 = 0$$

Sacamos factor común:

$$n(2-c)+3 = 0$$

Y despejamos la n:

$$n = \frac{3}{c-2}$$

Por tanto, cualquier valor para c tal que $c > 2$ sería válido. Fijando un valor de c, por ejemplo $c = 3$ obtenemos que n toma el valor 3.

□

Ejercicio 2.1
Dada $f(n) = 3n^3 + 5n$. Obtener el valor de n y c para que $f(n) \leq cn^3$ □

Ejemplos mas comunes de los órdenes de eficiencia

De más eficiente a menos:

1. Orden constante: $T(n) = a$
2. Orden logarítmico: $T(n) = log(n)$ (no importa la base)
3. Raíz cuadrada: $T(n) = \sqrt{n}$
4. Orden lineal: $T(n) = n$
5. Polinomios: $T(n) = n^2, n^3, ...n^k$
6. Tiempo exponencial: $T(n) = c^n$ con c una constante
7. Factorial: $T(n) = n!$
8. Tiempo exponencial con base variable: $T(n) = n^n$

Clase de eficiencia

Al conjunto de todas las funciones que están acotadas superiormente por $T(n)$ pertenecen a la clase de eficiencia representada por $T(n)$.
Dado el conjunto de todas las funciones correspondientes al tiempo de ejecución de todos los algoritmos, podemos clasificar estas funciones como pertenecientes a una clase de eficiencia. Así diremos que la función $f(n) = 2n+3$, que vimos en el Ejemplo 2.2.1, tiene clase de eficiencia $T(n) = n$.
Para saber qué clase de eficiencia tiene un polinomio seguimos los siguientes pasos:

1. Quitamos las constantes
2. Miramos el que tarde más en forma funcional

Ejemplo 2.2.2
Obtener el orden de eficiencia de $f(n) = 3n + 4log_2(n) + 5\sqrt{n}$

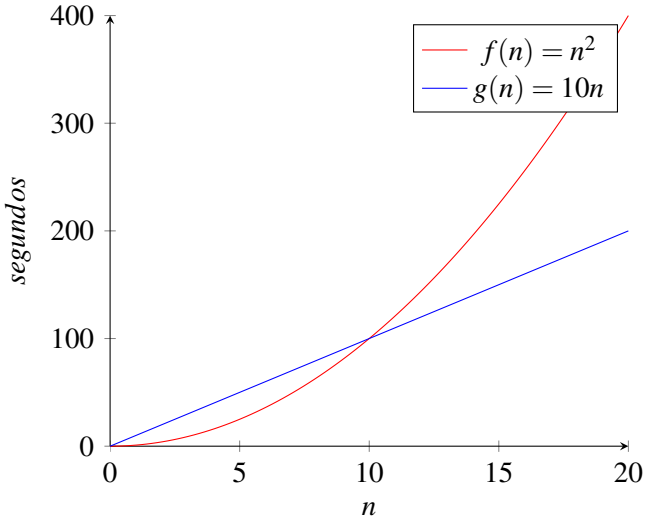
1. Quitamos las constantes de $f(n)$
2. Miramos el que tarde más en forma funcional que es n
3. Nuestro orden de eficiencia es $O(n)$

□

Las funciones se agrupan según su eficiencia en clases de eficiencia, que son:

- Lineal: $T(n) = n$
- Constante: $T(n) = a$
- Logarítmica: $T(n) = log(n)$
- Otros

Ejemplo 2.2.3
En este ejemplo se analiza asíntoticamente cuando es mejor una función frente a otra. Ambas representando tiempos de ejecución dado un tamaño de problema. Hacemos un estudio asintótico de dos algoritmos distintos con $f(n) = n^2$ y $g(n) = 10n$ y obtenemos la siguiente gráfica:



g(n) es más eficiente siempre y cuando $n > n_0$ (como vemos en la gráfica, n_0 es 10). Vamos a calcular de forma mas formal n_0:

$$10n \leq n^2$$
$$10n = n^2$$
$$n^2 - 10n = 0$$
$$n(n-10) = 0$$
$$n = 10$$

A partir de $n = 10$, $g(n)$ es más eficiente. □

Ejemplo 2.2.4

$$f(n) = \begin{cases} n^2 & \text{si } n \text{ es par} \\ n^4 & \text{si } n \text{ es impar} \end{cases}$$

$g(n) = n^3$

En este ejemplo no se puede establecer una relación de orden ya que para $f(n)$ hay un número infinito de veces en los que se comportará de una manera y otro número infinito de veces en los que se comportará de la otra, ya que los números pares e impares son infinitos.
En el ejemplo 2.2.3, $g(n)$ era peor en un número finito de datos ($n < 10$) por eso nos hemos decantado por $g(n)$, era mejor en un número infinito de casos.

□

O-grande

Definimos como *O-Grande* al tiempo de ejecución de un algoritmo en el **peor** de los casos.
Diremos que un algoritmo con tiempo de ejecución $g(n) \in O(f(n))$ si $\exists c \in \mathbb{R}_0^+$ y un n_0[1] $\in \mathbb{N}$ tal que $\forall n \geq n_0$[2] *O(T(n)* se puede establecer como el conjunto de todas las funciones que quedan por debajo de $T(n)$. (en un numero infinito de tamaños de problema)

$O(T(n)) = \{t : \mathbb{N} \to \mathbb{R} | (\exists c \in \mathbb{R}^+ y n_0 \in \mathbb{N})\ y\ (\forall n \in \mathbb{N} \text{ con } t(n) \leq cT(n)\}$

Ejemplo 2.2.5
Dar ejemplos de funciones $t(n)$ que están en $O(n^2)$.

Las funciones que están en $O(n^2)$ son todas las funciones que están acotadas asintóticamente superiormente por n^2. Así podemos decir que

$O(n^2) = \{1, 2, 3, 4, \ldots, log(n), \sqrt{n}, 2n, 2n+3, \ldots, n^2, 5n^2, \ldots\}$

Habría muchas más funciones pero estas son algunos ejemplos.

□

Ejemplo 2.2.6
Dado $O(n^2)$ dar ejemplos de funciones $t(n)$ que no están en $O(n^2)$.

Las funciones que no están en $O(n^2)$ son todas las funciones que no están acotadas asíntoticamente superiormente por n^2. Así podemos decir que $\{n^3, 2^n, n^5, n!, \ldots\} \notin O(n^2)$

Habría muchas mas funciones pero estas son algunos ejemplos.

□

[1] Tamaño del problema

[2] Este párrafo se leería así: g(n) pertenece a O de f(n) si existe una constante en R positivo y un n sub cero en N tal que para todo N sea mayor que n sub cero.

Ejemplo 2.2.7
Calcular valores de n_0 y c para los que se cumplen que $g(n) \in O(n^2)$.

$$n^2 + 5n \leq c \cdot n^2$$
$$n^2 + 5n = c \cdot n^2$$
$$n^2 - c \cdot n^2 + 5n = 0$$
$$n^2(1-c) + 5n = 0$$
$$n(n(1-c) + 5) = 0$$

Despejamos la n que está dentro del paréntesis:

$$n = \frac{5}{c-1}$$

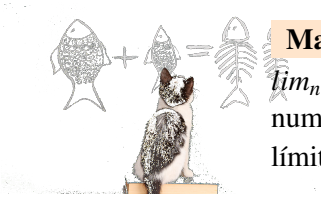
Nos queda entonces que $c > 1$, si le damos $c = 2$ entonces obtenemos que $n = 5$. Para $n \geq 5$ y $c = 2$ se cumple la definición.

□

Relaciones entre órdenes

$O(a) \subset O(log(n)) \subset O(\sqrt{(n)}) \subset O(n) \subset O(nlog(n)) \subset O(n^2) \subset_{k>2} O(n^k) \subset O(2^n) \subset O(n!) \subset O(n^n)$
Para saber cuál de entre dos funciones es mayor, podemos calcular su limite cuando n tiende a infinito:

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



Matemáticas de fondo 2.2.1 **Regla de L'Hôpital**.Dado el siguiente límite $lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{0}{0}$(o igual a $\frac{\infty}{\infty}$) la regla de L'Hôpital dice que se derive el numerador y denominador por separado y se obtenga sobre las derivadas el límite $lim_{n\to\infty} \frac{f(n)}{g(n)} = lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

Ejemplo 2.2.8

Tenemos $f(n) = log(n)$ y $g(n) = n$, ¿cuál de los dos es más grande? Aplicamos L'Hôpital:
$\lim_{n\to\infty} \frac{log(n)}{n} = \lim_{n\to\infty} \frac{\frac{1}{n}}{n} = \lim_{n\to\infty} \frac{1}{n} = 0 \longrightarrow g(n) > f(n)$

□

Reglas del O-Grande

1. $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Longrightarrow f(n) \in O(h(n))$
2. $a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n^1 + a \in O(n^d)$
3. **No** importa la base de los logaritmos: $log_a(n) \in O(log_b(n))$
4. La base del exponente de las potencías **sí** importa: $3^n \notin O(2^n) \longrightarrow 3^n \in O(3^n)$
5. El exponente de las potencias **sí** importa: $a^{n^2} \notin O(a^n)$; $a^{2n} \notin O(a^n) \rightarrow 2^{2n} = 4^n$

Ejemplo 2.2.9
¿Son ciertas las siguientes afirmaciones?

- $3n^3 + 2n^2 \in O(n^3) \longrightarrow$ Sí.
- $3^n \in O(2^n) \longrightarrow$ No.
- $2^{n+1} \in O(2^n) \longrightarrow$ Sí, ya que $2^{n+1} = 2 \cdot 2^n$
- $f(n) \in O(n) \rightarrow 2^{f(n)} \in 2^n \longrightarrow$ No. Por ejemplo si $f(n) = 3n$ se cumple que $3n \in O(n)$ pero no se cumple $2^{3n} = 8^n \notin 2^n$
- $O(2^n) \subset O(3^{log_2(n)}) \longrightarrow$ No. Ya que $3^{log_2(n)} = 3^{\frac{log_3(n)}{log_3(2)}} = n^{\frac{1}{log_3(2)}}$

□

Normas para obtener la eficiencia de un código

Las *operaciones elementales* las contaremos como una instrucción (inst.) básica que pertenece a $O(1)$, dentro de este grupo se incluyen:

1. Declaración
2. Asignaciones
3. Comparaciones simples
4. Operaciones aritméticas
5. Entrada y salida de datos básicos
6. Acceso a un array
7. etc

Ejemplo 2.2.10
Vamos a ver cómo de eficiente es el siguiente código

```
1 int A[MAX]; //1 inst.
2
3 for (int i=0; i<n; i++) //2 inst.; 1 inst; 1 inst.
4       A[i] = 0;    //2 inst
```

En la primera vuelta del bucle, hemos realizado 2 inst. para inicializar la i y otro compararla con n. A partir de ahí, en el resto de las vueltas gastaremos una 1 inst para comparar i con n, otra para incrementar i y dos más para igualar la componente del vector a cero:

$$(1+1)+1+\sum_{i=0}^{n-1}[(1+1)+(1+1)] = 3+\sum_{i=0}^{n-1}4 = 3+4\sum_{i=0}^{n-1}1 = 3+4n \in O(n)$$

A efectos prácticos, se suele dejar así:

$$\sum_{i=0}^{n-1}1 = n$$

□

Regla de la suma

Sean dos trozos de código, c_1 y c_2, independientes con eficiencia $T_1(n)$ y $T_2(n)$ respectivamente:

$$T_1(n)+T_2(n) \in O(max(f(n),g(n))) \begin{cases} T_1(n) \in O(f(n)) \\ T_2(n) \in O(g(n)) \end{cases}$$

Ejemplo 2.2.11

Sentencias if else:

```
int a=5, b=100;

if (a < b)        //O(1)
      cout << b;

else
      for (int i = 0; i < n; i++)    //O(n)
            A[i] = 0;

//O(max(1,n)) = O(n)
```

La eficiencia del algoritmo anterior es la suma de O(1) y O(n), que es O(n). □

Regla del producto

Sean dos trozos de código, $c1$ y $c2$, dependientes con tiempos de ejecución $T_1(n)$ y $T_2(n)$ respectivamente:

$$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n)) \begin{cases} T_1(n) \in O(f(n)) \\ T_2(n) \in O(g(n)) \end{cases}$$

La diferencia con la regla de la suma es que la suma de los dos trozos de código son **independientes** (como por ejemplo, una sentencia if else) mientras que en la regla del producto, son **dependientes** (como por ejemplo, bucles anidados).

Ejemplo 2.2.12

```
//Rellenar una matriz cuadrada
for (int i=0; i<n; i++) //Bucle for que empieza en 0 y acaba en n-1: O(n)
      for (int j=0; j<n; j++) //Bucle for que empieza en 0 y acaba en n-1: O(n)
            A[i][j] = 0; //Asignacion: O(1)
```

Al tener dos bucles anidados, para saber a qué O-Grande pertenecen debemos multiplicarlos: por lo que obtenemos que el código anterior pertenece a $O(n^2)$:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} \underbrace{1+1+1+\cdots+1}_{n} = \sum_{i=0}^{n-1} n = n\sum_{i=0}^{n-1} 1 = n\cdot n = n^2 \in O(n^2)$$

□

Ejemplo 2.2.13
El siguiente ejemplo combina ambas reglas.
Tenemos un if y un else, al ser códigos independientes debemos estudiarlos por separado:

```
1  //El if cuesta O(n^2)
2  if(A[0][0] == 0)
3  {
4        for (int i=0; i<n; i++)
5              for (int j=0; j<n; j++)
6                    A[i][j] = 0;
7  }
8
9  //la parte else cuesta  O(n)
10 else
11       for (int k=0; k<n; k++)
12             A[k][k] = 0;
```

Al ser una suma, debemos ver el $max(O(n^2), O(n))$, al ser el máximo $O(n^2)$, el código anterior es $O(n^2)$ □

Ejemplo 2.2.14
¿Cuánto miden los siguientes fragmentos de código? Ambos miden lo mismo.

```
1  for (int i=1; i<=n; i*=2)              for (int i=n; i>=1; i/=2)
2        A[i] = 0;                              A[i] = 0;
```

Hay que observar que en el for la variable i no se incrementa de 1 en 1, sino que en el código izquierdo la i se multiplica por 2 y en el código derecho la i se divide por 2. Recordando las mátematicas explicadas en la sección 1.2.1, podemos deducir que el número de veces que se repite el for en ambos casos es $log_2(n)$ veces:

$$\sum_{i=1}^{log_2 n} 1 = log_2(n) \in O(log_2(n))$$

□

Ejemplos:Cálculo de la eficiencia del código

Ejemplo 2.2.15

```
1  int funcion1 (int n)
2  {
3        int suma = 0; //O(1)
4
5        for (int i=0; i<n; i++)
6              suma += i; //O(1)
7
8        return suma; //O(1)
9  }
10
11 int funcion2 (int n)
12 {
13       int suma = 0; //O(1)
14
15       for (int i=0; i<n; i++)
16             suma += funcion1(i); //Aunque la suma sea O(1), funcion1 es O(n)
17
18       return suma; //O(1)
19 }
20
21 int main ()
22 {
23       int n; //O(1)
24       cin >> n; //O(1)
25       cout << funcion2(n); //O(n^2)
26 }
```

Empezamos analizando funcion1:
En el for tenemos:

$$\sum_{i=0}^{n-1} 1 = n$$

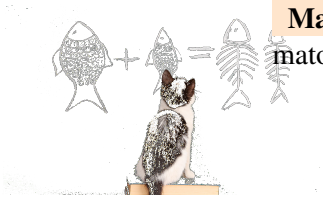
Por lo que toda la funcion1 es de orden $O(n)$
Analizamos funcion2:
En el for tenemos:

$$\sum_{i=0}^{n-1} n = n \sum_{i=0}^{n-1} 1 = n^2$$

Por lo que toda la funcion2 es de orden cuadrático, $O(n^2)$
Al ser la funcion2 de orden cuadrático la operación de más orden, por la regla de la suma, todo el main es de orden cuadrático, $O(n^2)$

□

En el ejemplo 2.2.16 al obtener la eficiencia aparece una sumatoria especial definida progresión aritmética. Veamos como se obtiene el resultado de una progresión aritmética.



Matemáticas de fondo 2.2.2 **Progresión Aritmética**. Sea la siguiente sumatoria

$$\sum_{i=S}^{t} f(i) = f(s) + f(s+1) + \cdots + f(t)$$

Esta sumatoria se caracteriza porque para obtener el numero siguiente al actual le sumamos d. Es decir $f(s+1) = f(s) + d$. Para simplificar el razonamiento que deriva el resultado de la sumatoria supongamos que $d = 1$. En este caso la sumatoria de $i = 1$ hasta $i = n$ es:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \cdots + n = \frac{n}{2}(n+1)$$

Para deducirlo, nos debemos de dar cuenta de que si sumamos el primero con el último, el segundo con el penúltimo, etc. obtenemos el mismo resultado, (n+1). Es decir:

$$\begin{aligned} n+1 &= n+1 \\ n+1 &= n-1+2 \\ n+1 &= n-2+3 \\ &\vdots \\ n+1 &= \tfrac{n}{2} + \tfrac{n}{2} + 1 \end{aligned}$$

El $\frac{n}{2}$ se refiere al numero de parejas, sumando $n+1$, que puedo formar con los n términos. De tal forma obtengo:

$$n+1 = (n-1)+2 = (n-2)+3 = \cdots = \frac{(n(n+1))}{2}$$

Ejemplo 2.2.16

```
1 void Ordena_Seleccion (const int *v, int n)
2 {
```

```
3       for (int i=0; i<(n-1); i++)
4             for (int j=i+1; j<n; j++)
5                   if (v[j] < v[i]) //Todo este if es de orden O(1)
6                         Intercambiar (v[i], v[j]);
7   }
8
9   void Intercambiar (int &a, int &b)
10  {
11        //Toda esta funcion es de orden O(1)
12        int aux = a;
13        a = b;
14        b = aux;
15  }
```

En la función Ordena Seleccion tenemos dos bucles anidados, en el segundo for tenemos:

$$\sum_{j=i+1}^{n-1} 1 = n-i-1$$

Tras aplicar la regla del producto con el primer for, nos queda que:

$$\sum_{i=0}^{n-2} n-i-1 = \sum_{i=0}^{n-2} n - \underbrace{\sum_{i=0}^{n-2} i}_{\text{Progresión Aritmética}} - \sum_{i=0}^{n-2} 1 = n(n-1) - (n-2)\frac{(n-1)}{2} - (n-1) =$$

$$= n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) = (n^2-1) - (\frac{n^2-3n+2}{2}) - n + 1 \in O(n^2)$$

□

Ejemplo 2.2.17

```
1   void funcion (int n)
2   {
3         int x=0, y=0; //O(1)
4
5         for (int i=1; i<n; i++)
6         {
7               if (i % 2 == 0)   //todo el if vale O(n)
8               {
```

```
9                   //Solo ejecutamos el contenido de este for la mitad de las
10                  //veces, cuando se cumple la condicion del if
11
12                  for (int j=i; j<n+1; j++)
13                        x++;    //O(1)
14
15                  for (int j=1; j<i+1; j++)
16                        y++;    //O(1)
17            }
18      }
19  }
```

Empezamos analizando los for dentro del if:

$$\sum_{j=i}^{n} 1 = n - i + 1$$

$$\sum_{j=1}^{i} 1 = i$$

Y por la regla de la suma tenemos que todo el if tiene orden de $O(n)$:

$$n - i + 1 + i = n + 1 \in O(n)$$

Y si analizamos el for mas externo, vemos que no siempre se entra en el if, solamente se entra la mitad de las veces.

$$\sum_{i=1}^{n/2} n = n\frac{n}{2} = \frac{n^2}{2} \in O(n^2)$$

Concluimos que la función es cuadrática, $O(n^2)$ □

Ejemplo 2.2.18

```
1   void funcion (int n)
2   {
3         int x=2, contador=0;
4
5         while (x <= n)
6         {
7               x *= 2;     //O(1)
8               contador++; //O(1)
9         }
10        cout << contador;
11  }
```

Al ir aumentando x multiplicándose por dos, el bucle lo repetiremos $log_2(n)$ veces (ver matemáticas de fondo 1.2.1), ya que la operación que hacemos en el bucle es $2^x = n$ y esa x es $x = log_2(n)$:

$$\sum_{contador=0}^{log_2(n)} 1 = log_2 n + 1 \in O(log_2(n))$$

□

Ejemplo 2.2.19

De los siguiente códigos, ¿cuál es de mayor orden?

```
int n, j;                    int n, j;
int x=0, i=1;                int i=2, x=0;
do {                         do {
      j=1;                         j=1;
      while (j<=n)                 while (j<=i)
      {                            {
            j *= 2;                      j*=2;
            x++;                         x++
      }                            }
      i++;                         i++;
} while (i<=n);              } while (i<=n)
```

El código de la izquierda repite su while $log_2(n+1)$ veces, y analizando el do while obtenemos que:

$$\sum_{i=1}^{n} log_2(n) + 1 = log_2(n) \sum_{i=1}^{n} 1 = nlog_2(n) + n \in O(nlog_2(n))$$

El código de la derecha repite su while $log_2 i$ veces, y analizándolo junto al do while obtenemos que:

$$\sum_{i=2}^{n} log_2 i = log_2(2) + log_2(3) + \cdots + log_2(n) = log_2(2 \cdot 3 \cdots n) = log_2(n!) \in O(log_2(n!))$$

Esta sumatoria puede acotarse por $nlog_2 n$ ya que:

$$log_2(n!) = log_2 2 + log_2 3 + \cdots + log_2 n \leq log_2 n + log_2 n + \cdots + log_2 n = nlog_2 n$$

□

Ejemplo 2.2.20

```
int suma = 0;
int k, j, n;
for (k=1; k<=n; k+=4)
      for (int j=1; j<=k; j*=2)
            suma++; //O(1)
```

Si estudiamos el segundo for anidado, vemos que:

$$\sum_{j=1}^{log_2 k} 1 = log_2 k$$

Y si ya estudiamos los bucles anidados en conjunto vemos que se ejecuta $\frac{n}{4}$ veces:

$$\sum_{k=1}^{n/4} log_2 k = log_2 1 + log_2 2 + \cdots + log_2(\frac{n}{4}) = log_2(\frac{n}{4}!) \in O(log(n!))$$

□

Cálculo de la eficiencia teórica en funciones recursivas

Ejemplo 2.2.21

```
int factorial (int n) //El tiempo de ejecucion de factorial es T(n)
{
      if (n<=1)
            return 1; //O(1)

      else
            return n*factorial(n-1);//Tiempo de ejecucion T(n-1)+1
                                    //(+1 por el producto y return).
}
```

El tiempo de ejecución de la función es:

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

Para calcular el orden del algoritmo, vamos descomponiendo el caso de $n \geq 2$:

$$T(n) = 1 + T(n-1) \text{para } n \geq 2$$

$$\text{Para } n \geq 3 \text{ sería } T(n-1) = T(n-2) + 1$$

$$T(n) = 1 + 1 + T(n-2) \text{para } n \geq 3$$

Para un caso general $k < n$ sería:

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

Si hacemos que $k = n-1$:

$$T(n) = (n-1) + T(n-(n-1)) = (n-1) + 1 = n \in O(n)$$

Así, concluimos que el factorial es de orden $O(n)$ □

Ejemplo 2.2.22

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

Ejecutamos de igual manera que en el ejercicio anterior, sustituyendo n por $n-1$

$$T(n) = n + T(n-1) \qquad n \geq 2$$

$$T(n) = n + (n-1) + T(n-2) \qquad n \geq 3$$

$$T(n) = n + (n-1) + (n-2) + T(n-3) \qquad n \geq 4$$

Para un $k < n$ tendríamos que:

$$T(n) = n + (n-1) + \cdots + (n-(k+1)) + T(n-k) \qquad n \geq k+1$$

Y si igualamos $n-1 = k$:

$$T(n) = n + (n-1) + \cdots + n - (n-1) + T(1) = \underbrace{n + (n-1) + \cdots + 1}_{\text{progresión aritmética}} + 1 = (n+1)\frac{n}{2} + 1 \in O(n^2)$$

□

Ejemplo 2.2.23
Algoritmo de Búsqueda Binaria:

```
1  int BB (int * v, int n, int x)
2  {
3        if (n > 0)
4        {
5              int m = (m/2);
6
7              if (v[m]==x)
8                    return m;
9
10             else //hasta aqui, todo es O(1)
11             {
12                   if (v[m] > x)
13                         return BB(v, (n/2), x); //T(n/2)
14
15                   else
16                         return BB(v+(n/2), n-(n/2), x); //T(n/2)
17
18                   //toda la funcion es T(n/2) + 1
19             }
20       }
21
22       else
23             return -1; //O(1)
24 }
```

La función de Búsqueda Binaria se correspondería con la siguiente función

$$T(n) = \begin{cases} T(\frac{n}{2})+1 & n \geq 2 \\ 1 & n \leq 1 \end{cases}$$

Ahora vamos a calcular su eficiencia. Tenemos que hacer un cambio de variable para cambiar el $\frac{n}{2}$ en función del valor anterior, como hemos hecho en los ejemplos anteriores:

$$n = 2^m$$

Y nos quedaría nuestra función así:

$$T(2^m) = 1 + T(\frac{2^m}{2}) \qquad 2^m \geq 2$$

Aplicamos logaritmos y así conseguimos que nuestra función esté en función del valor anterior:

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

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

$$T(2^m) = 3+T(2^{m-3}) \qquad m \geq 3$$

Para un $k \leq m$ sería:

$$T(2^m) = k+T(2^{m-k}) \qquad m \geq k$$

Y si igualamos $k = m$ sería:

$$T(2^m) = m+T(2^0) = m+1$$

Deshacemos el cambio de variable

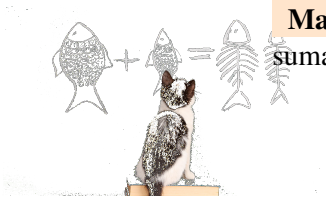
$$n = 2^m \longrightarrow log_2 n = log_2 2^m \longrightarrow log_2 n = m log_2 2 \longrightarrow m = log_2 n$$

Y sustituimos

$$T(n) = log_2 n + 1 \in O(log_2 n)$$

□

Antes de ver el siguiente ejemplo vamos a derivar cunato suma una progresión geométrica.



Matemáticas de fondo 2.2.3 **Progresión Geométrica**. Sea la siguiente sumatoria

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + \cdots + f(t)$$

Esta sumatoria se caracteriza porque para obtener el numero siguiente el actual se multiplica por a (razón de la progresión). Es decir $f(s+1) = a * f(s)$.

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + \cdots + a^n = \frac{a^{n+1}-1}{a-1}$$

Para deducirlo, multiplicamos todo por a y luego restamos:

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + \cdots + a^n = S_n$$

$$\sum_{i=0}^{n} aa^i = a^1 + a^2 + \cdots + a^{n+1} = aS_n$$

Al restar nos queda que:

$$aS_n - S_n = a^{n+1} - a^0$$

Sacamos factor común S_n:

$$S_n(a-1) = a^{n+1} - a^0$$

Y despejamos S_n:

$$S_n = \frac{a^{n+1}-1}{a-1}$$

Ejemplo 2.2.24

```
1        void funcion (int * v, int n) // tiempo de ejecucion T(n)
2       {
3             if (n==1) //Solamente un Caso Base 1
4              return v[0]); //O(1)
5
6
```

```
7               else if (n>1)
8               {
9                       int ni=(n/2), nd=n-(n/2);
10                      int * vi=new int [ni];
11                      int * vd=new int [nd];
12                      int sumai=0,sumad=0;
13                      for(int i=0;i<ni;i++){
14                              vi[i]=v[i];
15                              sumai+=vi[i];
16                       }
17                      for(int i=0;i<nd;i++){
18
19                              vd[i]=v[i+(n/2)];
20                              sumad+=vd[i];
21                       }
22                       if (sumai>sumad)  {
23                      //Ordenamos a la izquierda
24                          funcion (vi,ni); //T(n/2)
25                          for (int i=0;in;i++)// O(n)
26                            v[i]=sumai;
27                      }
28                      else {
29                          funcion(vd,nd); //T(n/2)
30                          for (int i=0;in;i++)// O(n)
31                            v[i]=sumad;
32                       }
33
34
35
36              }
37      }
```

Como se puede observar en la función existe una llamada recursiva por eso el tiempo de ejecución será $T(n/2)$. A continuacion se ejecuta un for que nos cuesta n.

$$T(n) = \begin{cases} T(\frac{n}{2}) + n & n \geq 2 \\ 1 & n == 1 \end{cases}$$

Ahora vamos a calcular su eficiencia. Tenemos que hacer un cambio de variable para cambiar el $\frac{n}{2}$ en función del valor anterior, como hemos hecho en los ejemplos anteriores:

$$n = 2^m$$

Por lo tanto el tiempo de ejecución en función de m sería:

$$T(2^m) = 2^m + T(\frac{2^m}{2}) \qquad 2^m \geq 2$$

Sustituimos k veces:

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

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

$$T(2^m) = 2^m + 2^{m-1} + 2^{m-2} + T(2^{m-3}) \qquad m \geq 3$$

Para un $k \leq m$ sería:

$$T(2^m) = 2^m + 2^{m-1} + \ldots 2^{m-k+} + T(2^{m-k}) \qquad m \geq k$$

Y si igualamos $k = m$ sería:

$$T(2^m) = 2^m + 2^{m-1} + 2^{m-2} + \cdots + T(2^0) = \underbrace{2^m + 2^{m-1} + 2^{m-2} + \cdots + 1}_{\text{Progresión Geométrica de razón 2}} = \frac{2^{m+1} - 1}{2 - 1}$$

Deshacemos el cambio de variable

$$n = 2^m \longrightarrow log_2 n = log_2 2^m \longrightarrow log_2 n = m log_2 2 \longrightarrow m = log_2 n$$

Y sustituimos

$$T(n) = 2 * n - 1 \in O(n)$$

□

Ejemplo 2.2.25
El algoritmo de ordenación Mergesort ya lo vimos en la sección 1.2.4. Los pasos fundamentales son:

```
1        void Orden_MergeSort (int * v, int n) // tiempo de ejecucion T(n)
2       {
3             if (n==1) //Solamente un Caso Base 1
4              return v[0]); //O(1)
5             else
6
7             else if (n>1)
8             {
9                   int ni=(n/2), nd=n-(n/2);
10                  int * vi=new int [ni];
11                  int * vd=new int [nd];
12
13                  for(int i=0;i<ni;i++)
```

```
14                      vi[i]=v[i];
15                  for(int i=0;i<nd;i++)
16                      vd[i]=v[i+(n/2)];
17
18                  //Ordenamos a la izquierda
19                  Orden_MergeSort (vi,ni); //T(n/2)
20
21                  //Ordenamos a la derecha
22                  Orden_MergeSort (vd,nd); //T(n/2)
23
24                  //Fusionamos en v, vi y vd
25                  Fusion(v,vi,vd,ni,nd); //O(n)
26                  delete [] vi;
27                  delete [] vd;
28              }
29      }
```

Como se puede observar en el algoritmo Mergesort existen dos llamadas recursivas una primera con los elementos a la izquierda de la mitad y otro con la mitad derecha, por eso el tiempo de ejecución será $T(n/2)$ para cada mitad. La fusión nos cuesta n.

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & n \geq 2 \\ 1 & n \leq 1 \end{cases}$$

Ahora vamos a calcular su eficiencia. Tenemos que hacer un cambio de variable para cambiar el $\frac{n}{2}$ en función del valor anterior, como hemos hecho en los ejemplos anteriores:

$$n = 2^m$$

Por lo tanto el tiempo de ejecución en función de m sería:

$$T(2^m) = 2^m + 2T(\frac{2^m}{2}) \qquad 2^m \geq 2$$

Aplicando sutitución k veces:

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

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

$$T(2^m) = 2^m + 2 * 2^{m-1} + 2^2 * 2^{m-2} + 2^3 T(2^{m-3}) \qquad m \geq 3$$

Para un $k \leq m$ sería:

$$T(2^m) = 2^m + 2 * 2^{m-1} + \cdots + 2^k * 2^{m-k+} + 2^{m-k} T(2^{m-k}) \qquad m \geq k$$

Y si igualamos $k = m$ sería:

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

Deshacemos el cambio de variable

$$n = 2^m \longrightarrow log_2 n = log_2 2^m \longrightarrow log_2 n = m log_2 2 \longrightarrow m = log_2 n$$

Y sustituimos

$$T(n) = n log_2 n \in O(n log_2 n)$$

□

2.3 Tiempo Amortizado

Para introducir como calcular este tiempo veámos un ejemplo en nuestra vida diaria. Supongamos que nuestra cocina la recogimos el dia anterior, y tenemos todos los vasos limpios. Desde este punto ir a beber agua a la cocina simplemente consiste en coger un vaso llenarlo de agua, beber y dejarlo en el fregadero. El tiempo necesario es bajo mientras haya vasos limpios. Ahora suponed que vamos a la cocina a beber agua y no tengo ningún vaso limpio. Entonces el procedimiento que sigo es poner todos los vasos en el lavavajillas, programarlo, dejar que el lavavajillas haga su trabajo y finalmente colocar los vasos en el armario. Desde aqui ya puedo coger un vaso limpio, beber agua y dejarlo en el fregadero. Por lo tanto el tiempo invertido en beber agua cuando hay vasos limpios es muy pequeño frente a cuando no los hay. En este caso la secuencia de veces que voy a beber agua no son independientes y por lo tanto no puedo decir que todas las veces gaste mucho tiempo (en el peor de los casos). Así que estas acciones, beber agua, están relacionadas. El tiempo amortizado lo que intenta hacer es una media sobre llamadas sucesivas. Un ejemplo donde se aplica este análisis es cuando queremos añadir un elemento en un vector dinámico del tipo

```
1  class VD
2  {
3        private:
4
5              char * datos;//zona de memoria para almacenar los datos
6              int n_elementos; //numero de datos almacenados
7              int n; //espacio asignado a datos, reservado
8              void ampliar();
9              ...
10       public:
11             void Add (char d);
12             ..
13
14 };
```

Si existe espacio ($n > n\_elementos$) añado al final el elemento. Esto tiene un coste constante, digamos $O(1)$, y esto podria ser asi hasta haber hecho n inserciones. Así para realizar la $n+1$ inserción, antes tengo que ampliar el espacio de memoria, lo ampliará a un espacio $2*n$. Para ello busca una nueva zona de memoria con $2*n$ localizaciones, y a continuación vuelca los elementos añadidos hasta

el momento, que son n, en el nuevo espacio de memoria. Finalmente libera la memoria y asigna a datos este nuevo espacio de memoria. La función ampliar por lo tanto nos cuesta $O(n)$. Una vez ampliada la zona de memoria, Add inserta al final el nuevo elemento. Por lo tanto en promedio para $n+1$ llamadas sucesivas todo nos ha costado:

- Por las inserciones realizadas sin ampliar, $n * O(1) = O(n)$.
- Y para el elemento $n+1$, añadirlo nos ha costado n

Por lo tanto, en promedio para cada uno de los $n+1$ elementos lo que nos cuesta es: $O(\frac{2n+1}{n}) = O(2) = O(1)$. En este caso tenemos un tiempo amortizado constante cuando se hacen $n+1$ inserciones sucesivas. Por eso añadir un elemento a un vector dinámico, como el que hemos usado aqui, tiene una eficiencia constante.

2.4 Repaso de matemáticas

2.4.1 Propiedades de los logaritmos

Un logaritmo es el número de veces que podemos dividir entre la base el número. Por ejemplo, $log_2(8) = 3$ porque podemos dividir ocho entre 2 tres veces.

1. $log_b(x \cdot y) = log_b(x) + log_b(y)$
2. $log_b(\frac{x}{y}) = log_b(x) - log_b(y)$
3. $log_b(x^y) = y \cdot log_b(x)$
4. $log_a(x) = \frac{log_b(x)}{log_b(a)}$

2.4.2 Propiedades de las exponenciales

1. $a^{xy} = (a^x)^y$
2. $a^{x+y} = a^x \cdot a^y$
3. $b^x = a^{xlog_a(b)}$
4. $b = a^{log_a(b)}$
5. $a^{x-y} = \frac{a^x}{a^y}$
6. $a^{log_b(n)} = n^{log_b(a)}$

$\lfloor x \rfloor \longleftarrow$ mayor entero menor o igual que x (en C++: se puede usar la función floor de la librería math)
$\lceil x \rceil \longleftarrow$ menor entero mayor o igual que x (en C++: se puede usar la función ceil de la librería math)
Ejemplo: $\lfloor 3{,}2 \rfloor = 3$ y $\lceil 3{,}5 \rceil = 4$

2.4.3 Sumatorias

$$\sum_{i=S}^{t} f(i) = f(s) + f(s+1) + \cdots + f(t)$$

Progresión aritmética

Para obtener el siguiente número sumanos uno al actual.

$$\sum_{i=1}^{n} i = 1+2+3+4+\cdots+n = \frac{n}{2}(n+1)$$

Para deducirlo, nos debemos dar cuenta de que si sumamos el primero con el último, el segundo con el penúltimo, etc. obtenemos el mismo resultado(n+1). El $\frac{n}{2}$ se refiere al numero de parejas, sumando $n+1$, que puedo formar con los n términos. De tal forma obtengo:

$$n+1 = (n-1)+2 = (n-2)+3 = \cdots = \frac{(n(n+1))}{2}$$

Progresión geométrica

Para obtener el siguiente número multiplicamos el actual por a.

$$\sum_{i=0}^{h} a^i = a^0 + a^1 + \cdots + a^n = \frac{a^{n+1}-1}{2-1}$$

Para deducirlo, multiplicamos todo por a y luego restamos:

$$\sum_{i=0}^{h} a^i = a^0 + a^1 + \cdots + a^n = S_n$$

$$\sum_{i=0}^{h} aa^i = a^1 + a^2 + \cdots + a^{n+1} = aS_n$$

Al restar nos queda que:

$$aS_n - S_n = a^{n+1} - a^0$$

Sacamos factor común S_n:

$$S_n(a-1) = a^{n+1} - a^0$$

Y despejamos S_n:

$$S_n = \frac{a^{n+1}-1}{a-1}$$

Suma de cuadrados

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Para deducirlo, primero debemos saber que: $(a+b)^3 = a^3 + b^3 + 3a^2b + 3ab^2$. Con esto en mente, empezamos a calcular los cubos de los $n+1$ primeros números:

$$1^3 = (1+0)^3 = 1^3$$

$$2^3 = (1+1)^3 = 1^3 + 1^3 + 31^2 + 31^2$$

.
.
.

$$(n+1)^3 = (1+n)^3 = 1^3 + n^3 + 3n + 3n^2$$

Los sumamos:

$$1^3 + 2^3 + \cdots + (n+1)^3 = (n+1) + (1^3 + 2^3 + \cdots + n^3) + 3(1+2+3+\cdots+n) + 3\sum_{i=1}^{n} i^2$$

Ahora pasamos al otro miembro la parte de $(1^3 + 2^3 + \cdots + n^3)$ y nos queda:

$$(n+1)^3 = (n+1) + \frac{3n}{2}(n+1) + 3S_n$$

Multiplicamos por 2:

$$2(n+1)^3 = 2(n+1) + 3(n+1)n + 6S_n$$

Por último, despejamos S_n:

$$S_n = \frac{(n+1)(2(n+1)^2 - 2 - 3n)}{6}$$

$$S_n = \frac{(n+1)(2n^2 + 4n + 2 - 2 - 3n)}{6}$$

$$S_n = \frac{(n+1)(2n^2 + n)}{6}$$

$$S_n = \frac{n(n+1)(2n+1)}{6}$$