Capítulo 3

Análisis de algoritmos.

3.1. Competencias del tema

La competencia a desarrollar en el tema es:

- **CTEC3** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

3.2. Introducción.

Como se ha mencionado en los temas introductorios, es preciso seleccionar el algoritmo más adecuado para resolver un problema cuando se dispone de varios para resolver dicho problema. Para ello es fundamental el *análisis de algoritmos*, el cual nos permitirá determinar la eficiencia de cada uno y tomar una decisión razonada sobre la elección del más eficiente. Aunque no existe un método general que nos permita analizar la eficiencia de los algoritmos, existen algunas técnicas básicas que suelen resultar útiles. Dentro de dichas técnicas se pueden resaltar el análisis de las distintas estructuras de control y el de las ecuaciones de recurrencia empleadas en los algoritmos recursivos.

El análisis de los algoritmos se suele hacer de dentro hacia afuera, es decir de lo particular a lo general. En primer lugar se calcula el tiempo requerido por las instrucciones individuales y después se combinan esos tiempos en base a las estructuras de control que relacionan las instrucciones del programa. En este capítulo vamos a hacer una distinción entre los algoritmos no recursivos, dentro de los cuales analizaremos las distintas estructuras de control, y los algoritmos recursivos.

3.3. Algoritmos no recursivos.

A continuación se muestran las distintas reglas que se van a aplicar a cada una de las instrucciones utilizadas en los algoritmos:

- **Asignaciones y expresiones simples.** El tiempo requerido para evaluar una constante, una expresión formada por términos simples o una asignación simple es $t(I) = c$, de donde $O(t(I)) = O(c)$ y por la propiedad 4 de la notación $O()$ sabemos que $O(c) = O(1)$.

  Toda asignación está sujeta a la semántica de copia, es decir, se copiará completamente el contenido de la variable origen en la variable de destino. De esta manera, la complejidad de una asignación vendrá dada por el tamaño de los datos que se asignen.

- **Secuencia de instrucciones.** El tiempo de ejecución de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución respectivos. Para una secuencia de dos instrucciones I_1 e I_2, se tiene que su tiempo de ejecución viene dado por la suma de los tiempos de ejecución de I_1 e I_2, es decir:

$$t(I_1; I_2) = t(I_1) + t(I_2)$$

Aplicando la regla de la suma, su orden es

$$O(t(I_1; I_2)) = O(t(I_1) + t(I_2)) = O(máximo(t(I_1), t(I_2)))$$

- **Instrucciones condicionales.** El tiempo de ejecución requerido por una instrucción condicional **si ... entonces ... finsi** es el necesario para evaluar la condición más el requerido para el conjunto de instrucciones que se ejecutan cuando se cumpla la condición.

$$t(si...entonces...finsi) = t(condición) + t(consecuente)$$

El tiempo de ejecución requerido por una instrucción condicional del tipo **si ... entonces ... si no ... finsi** es el resultante de evaluar la condición más el máximo entre los requeridos para computar el conjunto de instrucciones del consecuente y la alternativa.

$$t(si...entonces...sino...finsi) =$$
$$t(condición) + máximo(t(consecuente), t(alternativa))$$

Aplicando la regla de la suma, se tiene que

$$O(t(si...entonces...sino...finsi)) =$$
$$O(t(condición) + máximo(t(consecuente), t(alternativa)) =$$
$$O(máximo(t(condición), máximo(t(consecuente), t(alternativa))))$$

- **Instrucciones de iteración**

La complejidad temporal de un bucle **para ... finpara** es el producto del número de iteraciones por la complejidad de instrucciones del cuerpo del bucle.

Para bucles del tipo **mientras ... hacer ... finmientras** y **repetir ... hasta que** se sigue la regla anterior, considerando la estimación del número de iteraciones para el peor caso posible.

- **Llamadas subprogramas**

El tiempo necesario para evaluar la llamada a un subprograma es igual a la suma del tiempo de evaluación de los argumentos más el tiempo necesario para evaluar el cuerpo del subprograma con los parámetros formales sustituidos por los argumentos.

Si un subprograma P tiene como argumentos a $f_1, ..., f_n$, entonces el tiempo de evaluación de $P(f_1, ..., f_n)$ es:

$$t(P(f_1, ..., f_n)) = \sum_{i=1}^{n} t(f_i) + t(cuerpo)$$

Por tanto su orden será:

$$O(t(P(f_1,...,f_n))) = O(\sum_{i=1}^{n} t(f_i) + t(cuerpo)) = O(máximo(\sum_{i=1}^{n} t(f_i), t(cuerpo)))$$

Ejemplos:

1.- $x \leftarrow x + 1$

$O(t(n)) = O(c) = O(1)$ (Complejidad constante)

2.- **para** i **de** 1 **hasta** n **hacer**
$x \leftarrow x + 1$
finpara

$O(t(n)) = O(\sum_{i=1}^{n} c) = O(cn) = O(n)$ (Complejidad lineal)

3.- **para** i **de** 1 **hasta** n **hacer**
para j **de** 1 **hasta** n **hacer**
$suma \leftarrow suma + a(i,j)$
finpara
finpara

$O(t(n)) = O(\sum_{i=1}^{n} \sum_{j=1}^{n} c) = O(cn^2) = O(n^2)$ (Complejidad cuadrática)

4.- **si** $(n \ mod \ 2 = 0)$ **entonces**
para i **de** 1 **hasta** n **hacer**
$a(i) \leftarrow -a(i)$
finpara
finsi

$$\begin{aligned} O(t(n)) &= O(t(condición) + t(consecuente)) \\ &= O(máximo(c, \sum_{i=1}^{n} c)) \\ &= O(máximo(c, cn)) = O(cn) = O(n) \text{ (Complejidad lineal)} \end{aligned}$$

5.- $i \leftarrow 1$
mientras $(i \leq n)$ **hacer**
$x \leftarrow x * i$
$i \leftarrow i + 2$
finmientras

$$\begin{aligned} O(t(n)) &= O(t(asignación) + t(mientras))) \\ &= O(máximo(t(asignación), t(mientras)) \\ &= O(máximo(c, \sum_{i=1}^{ndiv2} 2c)) \\ &= O(máximo(c, 2c(ndiv2))) = O(2c(ndiv2)) \\ &= O(c'n) = O(n) \text{ (Complejidad lineal)} \end{aligned}$$

6.- $x \leftarrow 1$
mientras $(x \leq n)$ **hacer**
$x \leftarrow 2 * x$
finmientras

El cálculo de iteraciones de este bucle equivale a calcular el valor de t en el siguiente conjunto de instrucciones:

$x \leftarrow 1$
$t \leftarrow 1$
mientras $(x \leq n)$ **hacer**
$x \leftarrow 2 * x$
$t \leftarrow t + 1$
finmientras

Por ejemplo, si $n = 32$, entonces $t = 6$

Se cumple que $2^t \geq n$. Por tanto, despejando t: $t = \lceil log_2\ n \rceil$

Luego el tiempo de ejecución del bucle es:

$t(mientras) = 2clog\ n$

Finalmente, se tiene que

$$\begin{aligned} O(t(n)) &= O(t(asignaciones) + t(mientras)) \\ &= O(máximo(t(asignaciones), t(mientras))) \\ &= O(máximo(2c, 2clog\ n)) \\ &= O(2clog\ n) = O(log\ n) \text{ (Complejidad logarítmica)} \end{aligned}$$

7.- Evaluación de la complejidad de un algoritmo iterativo que calcula el n-ésimo término de la serie de Fibonacci.

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

`Algoritmo` $Fibonacci(n; -; fib)$
inicio
si $(n \leq 1)$ **entonces** $fib \leftarrow 1$
si no
$mayor \leftarrow 1$
$menor \leftarrow 1$
para i **de** 2 **hasta** n **hacer**
$auxiliar \leftarrow menor$
$menor \leftarrow mayor$
$mayor \leftarrow mayor + auxiliar$
finpara
$fib \leftarrow mayor$
finsi
fin

El tiempo de ejecución del algoritmo es

$$t(Fibonacci) = t(n) = t(si...entonces...si\ no...finsi)$$
$$= t(condición) + máximo(t(consecuente), t(alternativa))$$
$$= c + máximo(c, t(para...finpara))$$
$$= c + máximo(c, 3c \times (n-1)) = c + 3c \times (n-1) = 3cn - 2c$$

Y el orden de complejidad es

$$O(t(n)) = O(3cn - 2c) = O(n) \text{ (Complejidad lineal)}$$

8.- Evaluación de la complejidad de un algoritmo que calcula un número combinatorio.

Se desea calcular $\begin{pmatrix} n \\ m \end{pmatrix} = \frac{n!}{m!(n-m)!}$

Primero se mostrará una función que calcula el factorial de un número.

```
Función Factorial(n; - ; -):entero
inicio
  fact ← 1
 si (n > 1) entonces
      para i de 2 hasta n hacer
            fact ← fact * i
       finpara
  finsi
  devolver fact
fin
```

El tiempo de ejecución de la función $Factorial$ es:

$$t(Factorial(n)) = t(asignación) + t(si...entonces...finsi)$$
$$= c + t(condición) + t(consecuente)$$
$$= c + c + t(para...finpara) = 2c + c \times (n-1) = c(n+1)$$

El orden de complejidad de la función $Factorial$ es

$$O(t(Factorial(n)) = O(t(n)) = O(c(n+1)) = O(n)$$

El algoritmo parametrizado que calcula el número combinatorio es:

```
Algoritmo Combinatorio(n, m; -; c)
inicio
 si (n < m)   entonces c ← 0
       si no     c ← Factorial(n)/(Factorial(m) * Factorial(n - m))
  finsi
fin
```

El tiempo de ejecución del algoritmo $Combinatorio$ despreciando el tiempo de evaluación de los argumentos del algoritmo factorial es:

$$t(Combinatorio) = t(si...entonces...si\ no...finsi)$$
$$= t(condición) + máximo(t(consecuente), t(alternativa))$$
$$= c + máximo(c, t(asignación))$$

$$
\begin{aligned}
&= c + máximo(c, t(Factorial(n) \\
&\qquad\qquad +t(Factorial(m)) \\
&\qquad\qquad +t(Factorial(n-m))) \\
&= c + máximo(c, c(n+1) + c(m+1) + c(n-m+1)) \\
&= c + máximo(c, c(2n+3)) = c(2n+4)
\end{aligned}
$$

y el orden de complejidad es

$$O(Combinatorio(n, m, c))) = O(c(2n+4)) = O(n) \text{ (Lineal)}$$

9.- Multiplicación de matrices.

```
Algoritmo Producto(A, B, n, p, m; −; C)
inicio
  para i de 1 hasta n hacer
      para j de 1 hasta m hacer
          C(i, j) ← 0
          para k de 1 hasta p hacer
              C(i, j) ← A(i, k) ∗ B(k, j) + C(i, j)
          finpara
      finpara
  finpara
fin
```

El tiempo de ejecución es

$$
\begin{aligned}
&t(Producto(A, B, n, p, m, C)) = \\
&\qquad = t(para\ i\ ...finpara) \\
&\qquad = n \times t(para\ j\ ...finpara) \\
&\qquad = n \times m \times (t(asignación) + t(para\ k\ ...finpara)) \\
&\qquad = n \times m \times (c + p \times c) \\
&\qquad = n \times m \times (1+p) \times c = n\ m\ (p+1)\ c
\end{aligned}
$$

Suponiendo que $n = máximo(n, p, m)$, el orden de complejidad es

$$O(Producto(A, B, n, p, m, C)) = O(n\ m\ (p+1)\ c) = O(n^3) \text{ (Cúbica)}$$

10.- Ordenación por inserción directa.

Los métodos de ordenación son un muy ilustrativos en el estudio de la complejidad computacional al ser representativos de algunos de los métodos más habituales para resolver problemas. En algunos de ellos se puede estudiar con relativa facilidad el caso peor, el caso mejor y el caso medio.

La complejidad de un método de ordenación se suele evaluar en base a las comparaciones y los desplazamientos (intercambios entre elementos) que se realizan en el proceso de ordenación. Aunque el coste de un desplazamiento es superior al de una comparación, ambas operaciones se consideran elementales y por tanto, del mismo coste.

Como ejemplo vamos a nalizar el método de inserción directa. En dicho método, en la pasada i-ésima, cuando se inserta el elemento i-ésimo, se supone que los $i-1$ primeros elementos del vector ya están ordenados y después se inserta el elemento i en la posición adecuada. Para aumentar la eficiencia en la búsqueda de la posición de inserción y no tener controlar que no nos salgamos del vector en dicha búsqueda, se coloca un elemento centinela en la posición 0.

Una vez hechas estas aclaraciones, se puede comprobar que en la pasada i el mejor caso se da cuando el elemento a insertar es mayor que los $i-1$ primeros, en cuyo caso solo hay que realizar una comparación (solo se compara con el que ocupa la posición $i-1$) y ningún desplazamiento. Por otra parte, el peor caso se da cuando el elemento i es más pequeño que los $i-1$ primeros. En ese caso el número de comparaciones será i (se compara con los $i-1$ primeros y con el centinela) y de desplazamientos será $i-1$ (hay que desplazar los $i-1$ primeros elementos). El número de comparaciones y desplazamientos del caso medio se obtienen como media del caso peor y mejor.

```
Algoritmo OrdenacionInsercion(n; v; )
inicio
   para i de 2 hasta n hacer
      v[0] ← v[i]
      j ← i − 1
      mientras v[j] > v[0]
            v[j + 1] ← v[j]
            j ← j − 1
      finmientras
      v[j + 1] ← v[0]
   finpara
fin
```

Sean $c_{max}, c_{min}, c_{med}$ las comparaciones en los casos peor, mejor y medio cuando se inserta el elemento i.

Sean $d_{max}, d_{min}, d_{med}$ los desplazamientos en los casos peor, mejor y medio cuando se inserta el elemento i.

Según se ha visto antes, en la pasada i-ésima, las comparaciones y desplazamientos para los tres casos son:

$c_{max} = i$; $c_{min} = 1$; $c_{med} = \frac{i+1}{2}$

$d_{max} = i-1$; $d_{min} = 0$; $d_{med} = \frac{i-1}{2}$

Sean $C_{max}, C_{min}, C_{med}$ las comparaciones totales en los casos peor, mejor y medio.

Sean $D_{max}, D_{min}, D_{med}$ los desplazamientos totales en los casos peor, mejor y medio.

Las comparaciones y desplazamientos totales se obtienen calculando el sumatorio de los valores para la pasada i-ésima, haciendo variar i desde 2 hasta n:

$$C_{max} = \sum_{i=2}^{n} c_{max} = \sum_{i=2}^{n} i = \frac{(2+n)(n-1)}{2} = \frac{n^2+n-2}{2}$$

$$C_{min} = \sum_{i=2}^{n} c_{min} = \sum_{i=2}^{n} 1 = n-1$$

$$C_{med} = \sum_{i=2}^{n} c_{med} = \sum_{i=2}^{n} \frac{i+1}{2} = \frac{n^2+3n-4}{4}$$

$$D_{max} = \sum_{i=2}^{n} d_{max} = \sum_{i=2}^{n} (i-1) = \frac{n^2-n}{2}$$

$$D_{min} = \sum_{i=2}^{n} d_{min} = \sum_{i=2}^{n} 0 = 0$$

$$D_{med} = \sum_{i=2}^{n} d_{med} = \sum_{i=2}^{n} \frac{i-1}{2} = \frac{n^2-n}{4}$$

En el caso medio, que es el más probable, el número de comparaciones y el de desplazamientos es de orden $O(n^2)$. Los casos mejor y peor se dan cuando el vector está ordenado inicialmente o está ordenado en orden inverso, respectivamente.

3.4. Algoritmos recursivos.

Para simplificar el cálculo de la complejidad de los algoritmos recursivos, vamos a considerar que el tiempo de una acción elemental es de 1, en vez de la constante c, ya que de cara a la notación $O()$, aplicando la propiedad 4 se tiene que $O(c) = O(1)$. De otra forma, arrastraríamos la constante c de manera innecesaria en el cálculo de la complejidad.

El análisis de complejidad de los algoritmos recursivos se realiza mediante *funciones recurrentes*. Véase el siguiente ejemplo:

Función recursiva que calcula el factorial de un número.

Entorno

Nombre	Tipo	Significado
n	Entero	Parámetro de tipo dato Número del cual se va a calcular su factorial

```
Función Factorial(n; - ; -):entero
inicio
  si ((n = 0) o (n = 1))
       entonces devolver 1
       si no     devolver n * Factorial(n - 1)
    finsi
fin
```

La versión iterativa de esta función posee complejidad lineal. Parece razonable que esta versión recursiva posea la misma complejidad.

El tiempo $t(n)$ se calcula usando la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} 1 & si\ n = 0\ ó\ 1 \\ t(n-1)+3 & si\ n \geq 2 \end{cases}$$

El valor 1 de la primera parte de la expresión se debe al tiempo empleado en la condición. El valor $t(n-1)+3$ de la segunda parte de la expresión se obtiene de:

- $t(n-1)$ se debe al tiempo que se emplea en estimar $Factorial(n-1)$.
- el 3 se debe al tiempo unitario en estimar $n * Factorial(n-1)$ una vez calculado el factorial de $n-1$, más el tiempo unitario empleado en la condición, más el tiempo unitario empleado en estimar $n-1$.

Se pueden utilizar varios métodos para para resolver las ecuaciones recurrentes. Algunos de los que se usan más habitualmente son:

- *Método de expansión de recurrencias*: sustituir las recurrencias por su igualdad hasta llegar a cierta $t(n_0)$ conocida.
- *Método de acotación*: elegir dos funciones $f(n)$ y $g(n)$ que sean, respectivamente, *unas cotas superior e inferior* del mismo orden y usar la ecuación de recurrencia para probar que

$$\forall n \in N\ g(n) \leq\ t(n) \leq\ f(n).$$

- Método de la *ecuación característica*.

A continuación se mostrarán ejemplos correspondientes a dichos métodos.

3.4.1. Expansión de recurrencias

Mediante este método se expande la recurrencia hasta llegar a un valor conocido de $t(n)$ que suele ser el del caso elemental de la recursividad. A continuación se detallan algunos ejemplos.

1. Cálculo de la complejidad de la función *factorial*. Aplicando la expansión de recurrencias resulta:

$$\begin{aligned} t(n) &= t(n-1)+3 \\ &= (t(n-2)+3)+3 = t(n-2)+6 \\ &= (t(n-3)+3)+6 = t(n-3)+9 \\ &... \\ &= t(n-k)+3k \end{aligned}$$

   Para $k = n$ se tiene que:

$$t(n) = t(0)+3n = 1+3n.$$

   Luego la complejidad de la función recursiva *Factorial* es

   $O(t(n)) = O(1+3n) = O(n)$. Complejidad Lineal.

2. Cálculo de la complejidad de algoritmos recursivos funcionales con distintos órdenes de complejidad:

   1) Función doblemente recursiva:

```
Función Recursiva1(n; −; −) : entero
inicio
  si (n ≤ 1) entonces devolver 5
     si no devolver Recursiva1(n − 1) + Recursiva1(n − 1)
 finsi
fin
```

Planteando las ecuaciones de recurrencia:

$$t(n) = \begin{cases} 1 & si\ n \leq 1 \\ 2t(n-1)+4 & si\ n > 1 \end{cases}$$

El tiempo $2t(n-1)$ de la segunda parte de la expresión se debe a la doble llamada recursiva.

El tiempo 4 de la segunda parte de la expresión se debe al tiempo de la suma, más el tiempo del doble cálculo de $n-1$, más el tiempo de la condición.

Expandiendo las recurrencias, se tiene:

$t(n) = 2t(n-1)+4$

$= 2(2t(n-2)+4)+4 = 2^2t(n-2)+(2+1)\times 4$

...

$= 2^kt(n-k)+(2^{k-1}+2^{k-2}+...+1)\times 4 = 2^kt(n-k)+(2^k-1)\times 4$

Para $k=n$, $t(0)=1$ y, por tanto,

$t(n) = 2^nt(0)+(2^n-1)\times 4 = 2^n+(2^n-1)\times 4$

$= 5\times 2^{n+1}-4$

Luego el orden de complejidad es

$O(t(n)) = O(5\times 2^{n+1}-4) = O(2^n)$, exponencial.

2) Función simplemente recursiva:

```
Función Recursiva2(n; −; −) : entero
inicio
  si (n ≤ 1)
     entonces devolver 1
     si no devolver 2 ∗ Recursiva2(ndiv2)
 finsi
fin
```

Se tiene que

$$t(n) = \begin{cases} 1 & si\ n \leq 1 \\ t(n/2)+3 & si\ n > 1 \end{cases}$$

El tiempo $t(n/2)$ de la segunda parte de la expresión se debe a la llamada recursiva.

El tiempo 3 de la segunda parte de la expresión se debe al tiempo unitario empleado el calcular el producto devuelto, más el tiempo unitario en calcular $ndiv2$, más el tiempo unitario de la condición.

Expandiendo la recurrencia

$t(n) = t(n/2)+3$

$= (t(n/4)+3)+3 = t(n/4)+2\times 3$

$= (t(n/8)+3)+2\times 3 = t(n/8)+3\times 3$

...

$= t(n/2^k) + k \times 3$

Cuando $n/2^k = 1$, $k = log_2 n$ y, por tanto,

$t(n) = t(1) + 3 \times log_2 n = 1 + 3 \times log_2 n$

Luego el orden de complejidad es

$O(t(n)) = O(1 + 3 \times log_2 n) = O(log_2 n)$, logarítmica.

Este caso es similar al de la búsqueda binaria en un vector ordenado.

3) Cálculo de la complejidad de un algoritmo recursivo imperativo

```
Algoritmo Recursiva3(n; x, r; -)
inicio
  si (n = 0)   entonces r ← x
    si no
        para i de 1 hasta n hacer
                x ← 2 * x
                r ← r + 1
        finpara
        Recursivo3(n − 1, x, r)
  finsi
fin
```

La ecuación de recurrencias que se plantea es:

$$t(n) = \begin{cases} 2 & si\ n = 0 \\ 4 \times n + 2 + t(n-1) & si\ n > 0 \end{cases}$$

El tiempo 2 de la primera parte de la expresión se debe a los tiempos unitarios del predicado y de la asignación.

El tiempo $4n$ de la segunda parte de la expresión se debe a los tiempos unitarios de las dos asignaciones y los dos productos (4 tiempos unitarios) incluidos en el esquema $para$.

El tiempo 2 de la segunda parte de la expresión se debe al tiempo unitario empleado el calcular $n - 1$, más el tiempo unitario de la condición.

El tiempo $t(n-1)$ de la segunda parte de la expresión se debe a la llamada recursiva.

Expandiéndola

$t(n) = 4n + 2 + t(n-1)$

$= 4n + 2 + (4(n-1) + 2 + t(n-2)) = 4(n + (n-1)) + 4 + t(n-2)$

$= 4(n + (n-1)) + 4 + (4(n-2) + 2 + t(n-3))$

$= 4(n + (n-1) + (n-2)) + 6 + t(n-3)$

...

$= 4(n + (n-1) + (n-2) + ...) + 2(k-1) + (4(n-k+1) + 2 + t(n-k))$

$= 4(n + (n-1) + (n-2) + ... + (n-k+1)) + 2k + t(n-k)$

Para $k = n$ se tiene que $t(0) = 2$ y, por tanto,

$t(n) = 4(n + (n-1) + (n-2) + ... + (n-n+1)) + 2n + t(0)$

$= 4(n + (n-1) + (n-2) + ... + 1) + 2n + 2$

$= 4(\frac{1+n}{2}n) + 2n + 2 = 2n^2 + 4n + 2$

Luego el orden de complejidad es

$$O(t(n)) = O(2n^2 + 4n + 2) = O(n^2), \text{ cuadrática.}$$

4) Cálculo de la complejidad de algunos algoritmos complejos basados en divide y vencerás.

Como se verá en el tema correspondiente al método divide y vencerás, existen casos como el de la búsqueda binaria, cuyo cálculo es relativamente simple y se puede englobar dentro del caso de una llamada recursiva donde el tamaño del ejemplar se reduce a la mitad, caso que se ha visto anteriormente. Sin embargo, hay otros problemas de mayor complejidad como el método de ordenación por fusión y el quicksort, cuyo cálculo es más complejo. Para estos dos algoritmos tenemos una parte cuyo tiempo es lineal (fusión de vectores ordenados en el método de fusión o el algoritmo de la partición en el quicksort) y tenemos dos llamadas recursivas donde el tamaño del ejemplar se reduce a la mitad (en el caso del quicksort este sería el caso ideal).

Para estos casos más complejos la ecuación de recurrencias que se plantea es:

$$t(n) = \begin{cases} c & si\ n = 1 \\ 2 \times t(n/2) + bn + a & si\ n > 1\ y\ n = 2^k \end{cases}$$

a, b y c son constantes.

a sería el tiempo de las condiciones evaluadas para realizar las llamadas recursivas.

bn sería el tiempo de la fusión de vectores ordenados o el de la partición.

El tiempo $2t(n-1)$ de la segunda parte de la expresión se debe a las dos llamadas recursivas.

Expandiéndola y asumiendo que $n = 2^k$.

$$\begin{aligned} t(n) &= 2t(2^{k-1}) + a + b2^k \\ &= 2 * (2t(2^{k-2}) + a + b2^{k-1}) + a + b2^k \\ &= 2^2 t(2^{k-2}) + a + 2a + 2b2^k \\ &= 2^3 t(2^{k-3}) + a + 2a + 4a + 3b2^k \\ &... \\ &= 2^k t(1) + a(1 + 2 + 2^2 + ... + 2^{k-1}) + kb2^k \end{aligned}$$

Ya que $1 + 2 + 2^2 + ... + 2^{k-1} = 2^k - 1$, $n = 2^k$ y $k = log_2 n$ se tiene que

$t(n) = cn + a(n-1) + bnlog_2 n$

Luego el orden de complejidad es

$$O(t(n)) = O(cn + a(n-1) + bnlog_2 n) = O(nlog_2 n).$$

3.4.2. Acotación

Cálculo de la complejidad de una función recursiva que obtiene el k-ésimo elemento de la sucesión de Fibonacci:

```
Función Fibonacci(n; −; −) : entero
inicio
  si (n ≤ 1)
    entonces devolver 1
    si no devolver Fibonacci(n − 1) + Fibonacci(n − 2)
```

finsi

fin

La ecuación de recurrencias es:

$$t(n) = \begin{cases} n & si\ n \leq 1 \\ t(n-1) + t(n-2) + 4 & si\ n > 1 \end{cases}$$

Los tiempos $t(n-1)$ y $t(n-2)$ de la segunda parte de la expresión se deben a las llamadas recursivas.

El tiempo 4 de la segunda parte de la expresión se debe a los tiempos unitarios al calcular $n-1$ y $n-2$, más el tiempo unitario de la condición, más el tiempo de la suma del resultado de las llamadas.

Expandiéndola

$$\begin{aligned} t(n) &= t(n-1) + t(n-2) + 4 \\ &= (t(n-2) + t(n-3) + 4) + (t(n-3) + t(n-4) + 4) + 4 \\ &= t(n-2) + 2t(n-3) + t(n-4) + 3 * 4 \\ &... \end{aligned}$$

El desarrollo no es fácil, por lo que habrá que resolver esta ecuación de recurrencias mediante acotación:

Acotamos la función $t(n)$, sabiendo que $t(n-2) \leq t(n-1)$ (lo cual se demuestra por inducción).

$$g(n) = \begin{cases} n & si\ n \leq 1 \\ 2t(n-2) + 4 = 2g(n-2) + 4 & si\ n > 1 \end{cases}$$

$$f(n) = \begin{cases} n & si\ n \leq 1 \\ 2t(n-1) + 4 = 2f(n-1) + 4 & si\ n > 1 \end{cases}$$

Se cumple que $g(n) \leq t(n) \leq f(n)$

Expandiendo $g(n)$

$$\begin{aligned} g(n) &= 2g(n-2) + 4 = 2(2g(n-4) + 4) + 4 = 2^2 g(n-4) + 4 * (2+1) \\ &= 2^2(2g(n-6) + 4) + 4 * (2+1) = 2^3 g(n-6) + 4 * (2^2 + 2 + 1) \\ &... \\ &= 2^k g(n-2k) + 4 * (2^{k-1} + 2^{k-2} + ... + 1) = 2^k g(n-2k) + 4 * (2^k - 1) \end{aligned}$$

Si $n - 2k = 0$ entonces $g(n-2k) = g(0) = 0$

Luego $k = n/2$ y

$$\begin{aligned} g(n) &= 2^{n/2} g(0) + 4 * (2^{n/2} - 1) = 4 * 2^{n/2} - 4 \\ &= 4 * 2^{n/2} - 4 \end{aligned}$$

y su orden de complejidad es

$O(g(n)) = O(4 * 2^{n/2} - 4) = O(2^{n/2})$, orden exponencial.

Expandiendo $f(n)$

$$\begin{aligned} f(n) &= 2f(n-1) + 4 = 2(2f(n-2) + 4) + 4 = 2^2 f(n-2) + 4 * (2+1) \\ &= 2^2(2f(n-3) + 4) + 4 * (2+1) = 2^3 f(n-3) + 4 * (2^2 + 2 + 1) \end{aligned}$$

$$
\begin{aligned}
&\ldots \\
&= 2^k f(n-k) + 4 * (2^{k-1} + 2^{k-2} + \ldots + 1) = \\
&= 2^k f(n-k) + 4 * (2^k - 1)
\end{aligned}
$$

Si $n - k = 0$, entonces $f(n-k) = f(0) = 0$

Luego $k = n$ y

$$
\begin{aligned}
f(n) &= 2^n f(0) + 4 * (2^n - 1) \\
&= 4 * 2^n - 4
\end{aligned}
$$

y su orden de complejidad es

$O(f(n)) = O(4 * 2^n - 4) = O(2^n)$, orden exponencial.

Debido a la acotación, $O(g(n)) \subset O(t(n)) \subset O(f(n))$, luego $O(t(n) \approx O(2^n)$ y la complejidad de Fibonacci resulta ser exponencial, que es mucho peor que la versión iterativa que posee una complejidad lineal.

3.4.3. Método de la ecuación característica

Solo se va a analizar el caso en el que la ecuación de recurrencia es homogénea, lineal y de coeficientes constantes:

$$a_0 t(n) + a_1 t(n-1) + \ldots a_k t(n-k) = 0$$

Se puede demostrar que si $f(n)$ y $g(n)$ son soluciones de esta ecuación, cualquier combinación lineal de de dichas soluciones ($bf(n) + cg(n)$) también es solución de dicha ecuación. Por otra parte, intuitivamente, se puede observar que cualquier solución de esta ecuación será de la forma $t(n) = x^n$. Probando dicha solución en la ecuación se obtiene:

$$a_o x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0$$

Sacando factor común x^{n-k} se tiene que:

$$x^{n-k}(a_o x^k + a_1 x^{k-1} + \cdots + a_k) = 0$$

Despreciando la solución trivial $x = 0$ nos queda la *ecuación característica*:

$$a_o x^k + a_1 x^{k-1} + \cdots + a_k = 0$$

que tendrá k raíces, r_i, reales o complejas. Por lo tanto r_i^n será una solución de la recurrencia y como cualquier combinación lineal de soluciones es también una solución, se puede concluir que

$$t(n) = \sum_{i=1}^{k} c_i r_i^n$$

cumple con la ecuación de recurrencias para cualquier combinación de constantes $c_1, c_2, \ldots, c_k$. Las constantes se pueden determinar a partir de las condiciones iniciales de la ecuación de recurrencias. Hay que destacar que esta solución es válida solo cuando todas las soluciones de la ecuación característica son distintas. En el caso de que haya alguna solución múltiple la solución tiene otra forma, que se verá en un ejemplo posterior.

Vamos a ver el ejemplo de la sucesión de Fibonacci. La ecuación de recurrencia, si obviamos el tiempo de las operaciones $n - 1$ y $n - 2$, el de la suma y el de la comparación para el caso general, se obtiene de:

$t(n) = \begin{cases} n & si\ n \leq 1 \\ t(n-1) + t(n-2) & si\ n > 1 \end{cases}$ En este caso la ecuación de recurrencia será $t(n) = t(n-1) + t(n-2)$ con lo cual tendremos que $t(n) - t(n-1) - t(n-2) = 0$, de donde se obtiene la ecuación característica $x^2 - x - 1 = 0$, cuyas soluciones son $r_1 = \frac{1+\sqrt{5}}{2}$ y $r_2 = \frac{1-\sqrt{5}}{2}$. La solución general será de la forma $t(n) = c_1 r_1^n + c_2 r_2^n$. Para determinar c_1 y c_2 usamos las condicones iniciales. Cuando $n = 0$ tenemos que $c_1 + c_2 = 0$ ya que $t(0) = 0$ y para $n = 1$ se tiene que $c_1 r_1 + c_2 r_2 = 1$ ya que $t(1) = 1$. De este sistema se tiene que $c_1 = \frac{1}{\sqrt{5}}$ y $c_2 = -\frac{1}{\sqrt{5}}$. Por tanto se puede concluir que:

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

Esta es la conocida como fórmula de Moivre para la sucesión de Fibonacci.

En el caso de que la ecuación de recurrencias tenga soluciones múltiples la solución es algo más complicada. Se puede demostrar que si una solución es r y se repite m veces, entonces $t(n) = r^n, t(n) = nr^n, t(n) = n^2 r^n \ldots t(n) = n^{m-1} r^n$ son soluciones diferentes de la ecuación de recurrencia. A continuación veremos un ejemplo con la siguiente recurrencia:

$t(n) = \begin{cases} n & si\ n \leq 2 \\ 5t(n-1) - 8t(n-2) + 4t(n-3) & si\ n > 2 \end{cases}$

La ecuación caractarística será $x^3 - 5x^2 + 8x - 4 = 0$ cuyas soluciones son $x = 2$ (doble) y $x = 1$. Por lo tanto la solución general es de la forma:

$$t(n) = c_1 * 1^n + c_2 2^n + c_3 n 2^n$$

Según las condiciones iniciales:

- para $n = 0$, $c_1 + c_2 = 0$ ya que $t(0) = 0$
- para $n = 1$, $c_1 + 2c_2 + 2c_3 = 1$ ya que $t(1) = 1$
- para $n = 2$, $c_1 + 4c_2 + 8c_3 = 2$ ya que $t(2) = 2$

Resolviendo el sistema se tiene que $c_1 = -2$, $c_2 = 2$ y $c_3 = -\frac{1}{2}$, con lo que la solución es de la forma:

$$t(n) = 2^{n+1} - n2^{n-1} - 2$$

con lo que $O(t(n)) = O(2^n)$.

3.5. Series recurrentes de utilidad.

Como se ha podido ver en los ejemplos desarrollados en el tema hay varias sumas de series que se presentan con frecuencia en el cálculo de la complejidad de un algoritmo. A continuación se detallan las más habituales:

- **Progresiones aritméticas**

  Si $a_n = a_{n-1} + c$, donde c es constante, entonces

  $$\sum_{i=1}^{n} a_i = \frac{n(a_1 + a_n)}{2}$$

- **Progresiones geométricas**

  Si $a_n = ra_{n-1}$, donde $r \neq 1$ es constante, entonces

$$\sum_{i=1}^{n} a_i = \frac{a_1(1-r^n)}{1-r}$$

Si $0 < r < 1$, entonces

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-r}$$

- **Suma de cuadrados**

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