Resumenes y tipos de problemas F...

mike_

Fundamentos de análisis de algoritmos

1º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería
Universidad de Huelva

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

Resumen y tipos de problemas FAA

Conceptos básicos

Definición de algoritmo

De forma sencilla, un algoritmo es una serie de pasos con los que, dado un problema, se llega a su solución. Este algoritmo tendrá una complejidad espacial (memoria que necesitará ocupar para resolver el problema) y una complejidad temporal (tiempo que tomará hasta llegar a la solución). Nosotros nos centraremos en esta última, el análisis de la complejidad temporal de los algoritmos.

Definición de cota superior e inferior – Notación asintótica

Como ya hemos aclarado, analizaremos la complejidad temporal de los algoritmos para poder calcular el tiempo que un algoritmo tardará en resolver un problema. Para ello, tendremos en cuenta el tamaño del problema, el cual será n, por tanto, para un problema de tamaño n el algoritmo tardará un tiempo que será definido por T(n).

El tiempo de ejecución, T(n), vendrá definido por una ecuación en función de n (tamaño del problema). Según cómo vaya aumentando el tiempo se ejecución conforme va creciendo el tamaño del problema dado, el algoritmo será más rápido o más lento. Para ello, es necesario acotar dicha función complejidad.

- Cota superior. Se expresa como O(...) y sería una cota que el algoritmo, por muy grande que fuese el problema, nunca alcanzaría. Si $T(n) \in O(n)$, si el tiempo de ejecución del algoritmo pertenece a la cota superior n, quiere decir que existirá una constante c que cumpla que $c \cdot n$, a partir de un valor inicial de n, n_0, siempre será mayor que T(n).
  Ejemplo: $T(n) = 8n + 4$. $T(n) \in O(n)$ porque: $8n + 4 \leq c \cdot n$, para $n \geq n_0$. Tomamos como $n_0 = 1$, se busca ahora un valor para c que cumpla que, para cualquier valor de n, $c \cdot n$ sea mayor que T(n). En este caso, con $n = 1 \rightarrow 8 + 4 \leq c \rightarrow c \geq 12$. Cualquier valor de c que cumpla esa condición multiplicado por n dará un valor superior a $8n + 4$.
  También se puede comprobar mediante la notación asintótica que una función complejidad de un algoritmo no pertenece a O(n) porque es mayor y tiene que utilizarse otra cota:
  $T(n) = 2n^2$. ¿$T(n) \in O(n)$? $\rightarrow 2n^2 \leq c \cdot n \rightarrow 2n \leq c \rightarrow$ No exista ninguna constante c que multiplicada por cualquier n de un valor mayor que 2n (c es constante, por tanto, no puede ser igual a n, que es variable).
- Cota inferior. Se expresa como Ω(...) y sería lo contrario a la cota superior, tendrá valores que se encuentren por debajo de la función complejidad del algoritmo.
  Ejemplo: $T(n) = 8n + 4$. $T(n) \in \Omega(n)$ porque $8n + 4 \geq c \cdot n$, para $n \geq n_0$. Tomamos como $n_0 = 1$, se busca un valor de c que cumpla que $c \cdot n$ sea menor que T(n), por ejemplo, $c = 1$, $8n + 4$ siempre será mayor que n, para $n \geq 1$.
- Cota exacta. Que un algoritmo pertenezca a la cota exacta de algo quiere decir que tanto superiormente como inferiormente el algoritmo está acotado por una función de mismo grado. Se expresa como $T(n) \in \Theta(...)$.
  Ejemplo: $T(n) = 8n + 4 \in \Theta(n)$ ya que está acotada superiormente por 12n e inferiormente por n, ambas cotas tienen el mismo grado, que es n.

Comparación de algoritmos

Una vez conocidos los conceptos de cota superior e inferior, se puede comparar algoritmos para ver cual es más rápido. Para ello, se comprobará si un algoritmo hace de cota (superior o inferior) del otro algoritmo.

Teniendo dos algoritmos A y B, con sus funciones de complejidad T_A y T_B, si los valores de T_A se encuentran por encima de los de T_B, entonces el algoritmo B es más rápido que A puesto que su tiempo de ejecución es menor, por tanto, $T_B \in O(T_A)$, que también significa que $T_A \in \Omega(T_B)$, B es cota inferior de A.

La forma de realizar la comparación es a través del cálculo de límites cuando ambos algoritmos tienden a infinito (el tamaño del problema va creciendo). El método utilizado será L'Hôpital. Los tres resultados posibles del cálculo del límite son:

- $\lim_{n\to\infty}\left(\frac{Ta}{Tb}\right) = \infty$ → T_A crece más rápido que T_B, por tanto, $T_B \in O(T_A)$ o $T_A \in \Omega(T_B)$.
- $\lim_{n\to\infty}\left(\frac{Ta}{Tb}\right) = h$ → T_A y T_B crecen de forma paralela, por tanto, $T_A \in \Theta(T_B)$.
- $\lim_{n\to\infty}\left(\frac{Ta}{Tb}\right) = 0$ → T_A crece más lento que T_B, por tanto, $T_A \in O(T_B)$ o $T_B \in \Omega(T_A)$.

Con esto queda demostrado qué algoritmo es más rápido.

Análisis de Algoritmos

Hasta ahora, se ha visto qué es un algoritmo, su función complejidad y cómo se acota. Pero ¿cómo se obtiene a partir de un algoritmo su función complejidad? Pues a partir del código de este.

Operaciones elementales

La función complejidad de un algoritmo da el tiempo que tardaría un algoritmo en resolver el problema, pero ese tiempo no viene dado en segundos o en cualquier otra medida común, sino que viene dada en unidades de tiempo asociadas a instrucciones u operaciones elementales. Estas son las operaciones más simples que se pueden ejecutar y tienen un tiempo de ejecución asociado (tiempo de CPU). Por lo que, en verdad, lo que devuelve la función complejidad de un algoritmo es el número de operaciones elementales que ha realizado hasta llegar a la solución.

Estas operaciones elementales son: la suma, la resta, la multiplicación, la división, el módulo, la asignación, la comparación, el acceso a un vector, una llamada a una función, el return …

Todas estas operaciones se consideran que tienen una unidad de tiempo de CPU.

La declaración de variables no cuenta como operación puesto que la reserva de memoria se realiza en el momento anterior a ejecutar el algoritmo. Ej: int a, float numero …

Ejemplos:

- `int a = 4 + 2;` → 1 asignación, 1 suma, en total 2 OEs
- `a++;` → `a++ es igual que a = a + 1;` → 1 asig, 1 suma, en total 2 OEs
- `pos = vector[i-1];` → 1 asig, 1 acceso vector, 1 resta, en total 3 OEs

Sentencias de control de la ejecución

Estas son las sentencias que pueden modificar la secuencia de instrucciones que ejecuta un algoritmo según al tamaño del problema o al valor de una variable por ejemplo (son los if y los bucles). Es importante destacar que dada la existencia de la posibilidad de pueda ejecutarse un bloque de código en lugar de otro o que se ejecute un número de veces que puede variar, un mismo algoritmo, según el problema dado, puede tener diferentes tiempos de ejecución. Según número de operaciones que haga, puede encontrarse en:

- Caso Mejor: el algoritmo realiza en menor número de operaciones posibles.
- Caso Peor: el algoritmo realiza el máximo número de operaciones posibles.
- Caso Medio: el algoritmo realiza un número intermedio de operaciones.

Esto hace que, según el caso, un algoritmo tenga distintas complejidades.

Por ejemplo, al buscar un elemento en un array, este puede ser el primero en ser observado y por tanto no se realizarán más comparaciones (caso mejor), o bien, puede no estar en el array, por lo que se han realizado todas las comparaciones con todos los elementos (caso peor).

Sentencias if

Su estructura es sencilla, se realiza una condición y según su resultado, se ejecutará el código perteneciente al bloque del if, o, si no, el bloque del else, pero nunca los dos.

Su análisis sería: $T_{if} = T_{condicion} + T_{Cuerpo}$, siendo $T_{Cuerpo} \rightarrow T_{CuerpoIf}$ o bien, $T_{CuerpoElse}$

Si se está analizando el caso mejor de un algoritmo, T_{Cuerpo}, de las dos opciones posibles, se tomará aquella que tenga un valor inferior. Para el caso peor, se realizará lo contrario. Y para el caso medio, se tomará la media de ambos, if y else.

Sentencias while

También tiene una estructura simple, se realiza una comparación y después, si era cierta la comparación, se realiza el cuerpo del while. La diferencia respecto al if es que puede no ejecutarse una sola vez, sino varias, en función de cómo sea la condición del while, esto hace que su análisis sea:

$T_{while} = T_{condicion} + \sum(T_{condicion} + T_{CuerpoWhile} + T_{Salto}) + T_{Salto}$

$T_{condicion}$ sería la condición del while, $T_{CuerpoWhile}$, el bloque del while, y T_{Salto} es el coste de "saltar" desde el final del bloque del while hasta la condición inicial y también el salto desde la condición a la línea siguiente después del while cuando ya no se cumpla y finalice.

El sumatorio iría desde un origen hasta un final que dependerá del código analizado y el caso en el que se encuentre. Por ejemplo, en una búsqueda lineal elemento a elemento, en el caso peor en el que no se encuentra el elemento buscado, se comprobarán las n posiciones del array, y si la condición es `while (i < n && vector[i] != elemento)`, el sumatorio iría desde i = 1 hasta i = n (en pseudocódigo, los vectores/arrays empiezan en 1).

Hay una condición antes del sumatorio puesto que al menos, sea cual sea el resultado de la condición, ésta se realizará una vez.

Sentencias for

Son similares a los while, hay una condición y un cuerpo, pero además tiene una inicialización:

$T_{for} = T_{ini} + T_{condicion} + \sum(T_{condicion} + T_{CuerpoFor} + T_{Salto}) + T_{Salto}$

En el cuerpo del for se incluye también el incremento de la variable que lo recorre.

Normalmente, el for se ejecutará todas las veces que indica la condición (suele utilizarse para recorridos, lo que implica conocer cuantas iteraciones van a realizarse siempre, ósea, no tiene casos).

Si en el cuerpo de un if, un while o un for hubiera una llamada a una función, dicha llamada se sustituiría por 1 OE, que sería la llamada a la función, y por la función complejidad de la función llamada. OJO, si en un for por ejemplo, la función recibe como parámetro "i" en lugar de "n", en su función complejidad una vez calculada, en lugar de "n", se debe poner "i". Ejemplo:

$T(n)_{funcion} = 8n$

```
for (int i = 0; i < n; i++)
    v = v + funcion(i);
```

Esto se analizaría como:

$T(n) = T_{ini} + T_{cond} + \sum(T_{condicion} + T_{CuerpoFor} + T_{Salto}) + T_{Salto} \rightarrow T(n) = 1 + 1 + \sum_{i=1}^{n}(1 + 3 + 8i + 1 + 2) + 1$

El 3 viene de la asignación de v, la suma "v + " y la llamada a la función, y el 2 es el "i++".

Ahora hay que quitar sumatorios, desde dentro hacia fuera:

$T(n) = 3 + \sum_{i=1}(8i + 7) \rightarrow T(n) = 3 + \sum_{i=1}(8i) + \sum_{i=1}(7) \rightarrow$

$\sum_{i=1}(8i) = 8 \cdot \sum_{i=1}(i) \rightarrow 8 \cdot \frac{n(n-1)}{2} \rightarrow 4n^2 - 4n$

$\sum_{i=1}(7) = 7 \cdot \sum_{i=1}(1) \rightarrow 7 \cdot (n - 1 + 1) \rightarrow 7n$

$T(n) = 3 + 4n^2 - 4n + 7n \rightarrow$ **$T(n) = 4n^2 + 3n \in O(n^2)$.**

Conteo por operación característica

Hay ocasiones en las que no es necesario conocer el número de operaciones elementales que realiza el algoritmo, sino sólo su complejidad, la cual se puede conocer sin necesidad de contar todas las operaciones. Por ejemplo, en una búsqueda lineal, puedes conocer que pertenece a la cota superior O(n) sin tener que contar todas las OEs, con sólo contar cuantas comparaciones realiza el algoritmo también llegas a una función que pertenece al mismo orden (por ejemplo, será 2n en lugar de 14n + 21, lo que interesa es saber que es O(n), no es necesario saber que es 14n + 21 exactamente).

La operación característica será entonces aquella que sea utilizada como control de la ejecución, normalmente es una comparación o una llamada a una función.

Análisis de Algoritmos Recursivos

Un algoritmo recursivo es aquel que se invoca a sí mismo durante su ejecución, pasándole el problema dado, pero con un tamaño menor. Esto hace que no pueda ser analizado con tan sólo el conteo de operaciones como se acaba de ver.

Los algoritmos recursivos se componen de:

- Caso base. Es aquel en el que el problema es tan pequeño, que su solución es directa, no hay que hacer llamadas recursivas.
- Caso general. Es aquel en el que, para resolver el problema, es necesario descomponerlo a una forma más sencilla (más pequeña) y realizar una llamada a sí mismo con el problema reducido. Se realizan llamadas recursivas hasta llegar al caso base que da la solución para el subproblema que ha recibido. Esta solución irá subiendo desde el caso base hasta la llamada original de la función, completándose la solución en cada llamada que se ha hecho.

El ejemplo más sencillo de un algoritmo recursivo es el cálculo de la factorial de un número:

```
funcion factorial(n : entero)
    si (n == 0) entonces
        fact ← 1
    sino
        fact ← n * factorial(n-1)
    fsi
    devuelve fact
ffuncion
```

La factorial de un número es igual que multiplicar el número por la factorial del anterior: n! = n·(n-1)! siempre que n no sea 0, que entonces la factorial vale 1.

Como se puede ver, según el tamaño del problema, el algoritmo se ejecutará de una manera o de otra, haciendo que la función de complejidad quede como:

$T(n) = \begin{cases} 3 & si\ n = 0 \\ T(n-1) + 6 & si\ n > 0 \end{cases}$ Según sea el caso base o el caso general, será una función u otra.

Existen tres formas de analizar un algoritmo recursivo:

Método de Ecuación Característica:

Consiste en transformar la función complejidad del caso general en un polinomio, que una vez resuelto se podrá obtener la función complejidad del algoritmo:

T(n) = T(n – 1) + 6 → Hay que pasar todas las llamadas a la izquierda, quedando:

T(n) – T(n – 1) = 6 → Como 6 no es 0, es No Homogénea, por tanto, es necesario pasarlo a la izquierda, pero como a la izquierda sólo pueden ir llamadas, es necesario realizar una pequeña conversión.

$6 = b^n \cdot p(n)^d \rightarrow 6 = 1^n \cdot 6 \cdot n^0 \rightarrow b = 1; d = 0$, el cual pasa a la izquierda como $(x - b)^{d+1}$

Mientras, en la izquierda, los k elementos que hay se transforman en potencias de x que van desde k hasta 0: T(n) – T(n – 1) → $x^k - x^{k-1}$ → k-1 = 0 → x – 1.

Quedaría en la izquierda: $(x - 1)(x - 1)^{0+1}$, que sería igual a 0, → raiz (r) = 1 doble

La función complejidad quedaría como:

$T(n) = c_0 1^n n^0 + c_1 1^n n^1 = c_0 + c_1 n$

Es necesario calcular cuanto valen las constantes c_0 y c_1, para ello, se emplean valores pequeños que se puedan calcular:

Si T(0) = 3, T(1) = T(0) + 6 = 9, T(2) = T(1) + 6 = 15, por tanto:

Quedaría: $\begin{cases} c0 + c1 = 9 \\ c0 + 2c1 = 15 \end{cases}$ $c_0 = 3$, $c_1 = 6$ → **T(n) = 6n + 3 ∈ O(n).**

Método de Expansión de Recurrencia:

Sería deshacer la recurrencia tomando la función del caso general, profundizando hasta el caso base:

T(n) = T(n – 1) + 6 Sería lo mismo que → T(n) = (T(n – 2) + 6) + 6

T(n) = T(n – 2) + 12 → T(n) = (T(n – 3) + 6) + 12

T(n) = T(n – 3) + 18

…

Generalizando:

T(n) = T(n – i) + 6·i

Esto continuaría hasta llegar al caso base, desde el cual no se puede realizar ninguna llamada más, por tanto, esto es cuando n – i es igual a 0, o lo que es lo mismo, cuando i = n.

Eso nos deja con: T(n) = T(n – n) + 6·n → **T(n) = 6n + 3 ∈ O(n).**

Método del Teorema Maestro por Sutracción:

A partir de la función complejidad dada como: $T(n) = aT(n - b) + p(n)^k$ (polinomio de n de grado k)

Viendo la función del algoritmo: T(n) = T(n – 1) + 6 → a = 1, b = 1, k = 0

Observando los órdenes de complejidad:

$$\mathbf{T(n)} \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/a}) & \text{si } a > 1 \end{cases}$$

Como a = 1, entonces, **T(n) ∈ O(n).**

Ejemplo de algoritmo recursivo que reduce el problema a un subproblema de tamaño mitad:

```
funcion ejemplo2(n : entero)
    si (n == 1)                          1 OE
        devuelve 1                        1 OE
    sino
        devuelve ejemplo2(n/2) + 1        3 OEs + T(n/2)
ffuncion
```

$$T(n) = \begin{cases} 2 & si\ n = 1 \\ T\left(\frac{n}{2}\right) + 3 & si\ n > 1 \end{cases}$$

Método de Ecuación Característica:

Consiste en transformar la función complejidad en un polinomio:

T(n) = T(n/2) + 3 → Como la llamada recursiva se reduce en tamaño medio, es No lineal

Como no es lineal, hay que aplicar un cambio de base para que sí sea lineal: $n = 2^k$; k = log(n)

$T(2^k) = T(2^{k-1}) + 3$ → $T(2^k) - T(2^{k-1}) = 3$ → Como 3 es distinto de 0, es No homogénea

Para poder pasar a la izquierda el 3, hay que convertirlo primero en un polinomio tal que:

$3 = b^k \cdot p(k)^d$ → $3 = 1^k \cdot 3 \cdot k^0$ → b = 1; d = 0, el cual pasa a la izquierda como $(x - b)^{d+1}$

$(x - 1)(x - 1)^{0+1} = 0$ → r = 1 doble

$T(2^k) = c_0 1^k k^0 + c_1 1^k k^1 = c_0 + c_1 k$ → Deshaciendo el cambio de base:

$T(n) = c_0 + c_1 \log(n)$

Hay que calcular las constantes, para ello, se emplearán valores a partir del caso base:

Si T(1) = 2, T(2) = T(1) + 3 = 5, T(4) = T(2) + 3 = 8, por tanto:

Quedaría: $\begin{cases} c0 + c1 = 5 \\ c0 + 2c1 = 8 \end{cases}$ $c_0 = 2$, $c_1 = 3$ → **T(n) = 3·log(n) + 2 ∈ O(log(n)).**

Método de Expansión de Recurrencia:

Consiste en sustituir la llamada recursiva de la función complejidad por la función correspondiente a dicho tamaño del problema:

T(n) = T(n/2) + 3 → Sería lo mismo que poner: T(n) = (T(n/4) + 3) + 3 = T(n/4) + 6

T(n) = T(n/4) + 6 → T(n) = (T(n/8) + 3) + 6 = T(n/8) + 9

…

$T(n) = T(n/2^i) + 3i$

Hasta que $n / 2^i$ sea igual a 1, que sería el caso base, por tanto, se harían i llamadas recursivas. Esto haría que $2^i = n$, por tanto, $i = \log_2(n)$.

T(n) = T(n/n) + 3·log(n) → **T(n) = 3·log(n) + 2 ∈ O(log(n)).**

Método del Teorema Maestro por División:

A partir de la función complejidad dada como: $T(n) = aT(n/b) + O(n^k \log^p(n))$

En este caso, con T(n) = T(n/2) + 3: a = 1, b = 2, k = 0, p = 0, por tanto, viendo los órdenes:

$$T(n) \in \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \cdot \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \cdot \log^p n) & \text{si } a < b^k \end{cases}$$

Como $a = b^k$, $1 = 2^0$, el orden de complejidad sería: $O(n^0 \cdot \log^{0+1}(n))$, por tanto: **T(n) ∈ O(log(n)).**

Función complejidad de los algoritmos más comunes

- QuickSort:
  - Caso peor: Partition divide el vector en dos subvectores de tamaño 1 y n-1 siempre:
    $T(n) = \begin{cases} c1 & si\ n = 1 \\ T(n-1) + T(1) + c2n & si\ n > 1 \end{cases}$
  - Caso mejor: Partition divide el vector en dos subvectores de tamaño mitad siempre:
    $T(n) = \begin{cases} c1 & si\ n = 1 \\ 2T\left(\frac{n}{2}\right) + c2n & si\ n > 1 \end{cases}$
- MergeSort:
  - Para todos los casos siempre se divide en dos subvectores de tamaño mitad:
    $T(n) = \begin{cases} c1 & si\ n = 1 \\ 2T\left(\frac{n}{2}\right) + c2n & si\ n > 1 \end{cases}$
- K-esimo menor elemento:
  - Caso peor: Partition divide el vector en dos subvectores de tamaño 1 y n-1 siempre:
    $T(n) = \begin{cases} c1 & si\ n = 1 \\ T(n-1) + c2n & si\ n > 1 \end{cases}$
  - Caso mejor: Partition divide el vector tomando como pivote justamente el k-esimo elemento, por lo que sólo se realiza una única llamada a Partition, no hay llamadas recursivas:

    $T(n) = T(n)_{Partition} \in O(n)$.
- Búsqueda binaria:
  - Caso peor: el elemento buscado no está, se hacen todas las llamadas recursivas:
    $T(n) = \begin{cases} c1 & si\ n = 1 \\ T\left(\frac{n}{2}\right) + c2 & si\ n > 1 \end{cases}$
  - Caso mejor: el elemento buscado está en la posición mitad:
    $T(n) = c1 \in O(1)$.

Ejemplos de análisis de algoritmos

Una vez visto cómo se tendrían que analizar los diferentes tipos de algoritmos, vamos a ver unos cuantos ejemplos de cómo se analizan los siguientes algoritmos.

Algoritmo de ordenación Burbuja

```
procedimiento Burbuja(v [], n : enteros)
    para i = 1 hasta n hacer
        para j = n hasta i hacer
            si v[j] < v[j-1] entonces
                tmp ← v[j]
                v[j] ← v[j-1]
                v[j-1] ← tmp
            fsi
        fpara
    fpara
fprocedimiento
```

```
Ini: 2, Cond: 1, Inc: 2, Slt: 1
| Ini: 2, Cond: 1, Inc: 2, Slt: 1
| | Cond: 4 OEs
| |   CuerpoSi: 2
| |             4
| |             3 → 9 OEs
| Este bucle va de n hasta i
Este bucle va de 1 hasta n
```

Viendo que la condición de parada de los bucles no depende de nada que haga que pueda modificarla, estos bucles ejecutarán siempre todas sus iteraciones, sea el caso que sea.

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

$T(n) = 4 + \sum_{i=1}^{n}(7 + \sum_{j=n}^{i}(4 + Tsi))$

$Tsi = TCond + (TCuerpo\ o\ 0)$ → Según el resultado de la condición, el cuerpo del "si" se ejecutará o no, aquí es donde se pueden dar los casos mejor y peor.

Caso mejor: nunca se cumple la condición

$T(n) = 4 + \sum_{i=1}^{n}(7 + \sum_{j=n}^{i}(4 + 4)) \rightarrow T(n) = 4 + \sum_{i=1}^{n}(7 + \sum_{j=n}^{i}(8)) \rightarrow$

$T(n) = 4 + \sum_{i=1}^{n}(7 + 8(n - i + 1)) \rightarrow T(n) = 4 + \sum_{i=1}^{n}(15 + 8n - 8i) \rightarrow$

$T(n) = 4 + \sum_{i=1}^{n}(15) + \sum_{i=1}^{n}(8n) + \sum_{i=1}^{n}(8i) \rightarrow T(n) = 4 + 15\sum_{i=1}^{n}(1) + 8n\sum_{i=1}^{n}(1) + 8\sum_{i=1}^{n}(i) \rightarrow$

$T(n) = 4 + 15(n - 1 + 1) + 8n(n - 1 + 1) + 8\frac{n(n-1)}{2} \rightarrow$

$T(n) = 12n^2 + 11n + 4 \in O(n^2)$.

Caso peor: la condición siempre es cierta, por tanto, siempre se ejecuta el bloque si

El análisis sería exactamente igual, pero sustituyendo Tsi por (4 + 9).

Algoritmo búsqueda binaria recursivo – Caso peor

Para este algoritmo, es necesario que el vector se encuentre previamente ordenado.

```
funcion binaria(v [], ini, fin, e : enteros)
  si (ini > fin) entonces                          1 OE
    devuelve -1 // No está                         1 OE ← Caso base (no está)
  sino
    mitad ← (ini + fin) / 2                        3 OEs
    si (v[mitad] = e) entonces                     2 OEs
      devuelve mitad                               1 OE
    sino
      si (v[mitad] > e) entonces                   2 OEs
        devuelve binaria(v, ini, mitad, e)         2 OEs
      sino
        devuelve binaria(v, mitad+1, fin, e)       3 OEs ← Opción caso peor
      fsi
    fsi
  fsi
ffuncion
```

Nos quedaría la siguiente función complejidad:

$$T(n) = \begin{cases} 2 & si\ n < 1 \\ T\left(\frac{n}{2}\right) + 11 & si\ n \geq 1 \end{cases}$$

Por ecuación característica:

T(n) – T(n/2) = 11 → 11 ≠ 0 → No homogénea y como se reduce a tamaño mitad (división) → No lineal

Cambio de base: $n = 2^k$

$T(2^k) - T(2^k/2) = 11 \rightarrow T(2^k) - T(2^{k-1}) = 11;\ 11 \rightarrow b^k p(k)^d \rightarrow 11 = 1^k \cdot 11 \cdot k^0 \rightarrow b = 1, d = 0$

$(x - 1)(x - 1)^{0+1} = 0 \rightarrow r = 1$ doble

$T(2^k) = c_0 1^k k^0 + c_1 1^k k^1$ → Deshacer cambio de base → $T(n) = c_0 + c_1 \log(n)$

Para el cálculo de las constantes: T(1) = T(0) + 11 = 13; T(2) = T(1) + 11 = 24

$\begin{cases} c0 + c1\log(1) = 13 \\ c0 + c1\log(2) = 24 \end{cases} \rightarrow \begin{cases} c0 = 13 \\ c0 + c1 = 24 \end{cases}$ → $c_0 = 13$, $c_1 = 11$ → **T(n) = 11·log(n) + 13 ∈ O(log(n)).**

Por expansión de recurrencia:

T(n) = T(n/2) + 11 → T(n) = (T(n/4) + 11) + 11 →

T(n) = T(n/4) + 2·11

...

$T(n) = T(n/2^i) + 11i$

Hasta que llega al caso base, T(0), por tanto, $n = 2^i$ → i = log(n)

T(n) = T(0) + 11·log(n) → **T(n) = 11·log(n) + 2 ∈ O(log(n)).**

Algoritmo MergeSort

Para este algoritmo, sólo se sabe que la función Merge tiene O(n), pero no se sabe su función exacta, por lo tanto, se tomará $T(n)_{merge}$ como k·n (porque al ser O(n), existe una k que hace de cota superior).

```
funcion mergeSort(v [], ini, fin : enteros)
  si (fin > ini) entonces                          1 OE → Caso base, tamaño 1
    mitad ← (ini + fin) / 2                        3 OEs
    mergeSort(v, ini, mitad)                       1 OE
    mergeSort(v, mitad+1, fin)                     2 OEs
    merge(v, ini, fin) // mezcla las mitades       Kn OEs
  fsi
ffuncion
```

Siempre se realizan particiones y llamadas recursivas, esté como esté el vector:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + kn + 7 & \text{si } n > 1 \end{cases}$$

Se sabe que 7 es un valor constante (O(1)), por tanto, siempre valdrá igual. Entonces, puede existir una k' que k'n > kn + 7, luego:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T\left(\frac{n}{2}\right) + k'n & \text{si } n > 1 \end{cases}$$ Quedando más reducida

Por ecuación característica:

T(n) – 2T(n/2) = k'n → Se reduce a tamaño mitad → No lineal

Cambio de base: $n = 2^q$

$T(2^q) - 2T(n^{q-1}) = k' \cdot 2^q$ → No homogénea → $k' \cdot 2^q = b^q \cdot p(q)^d$ → b = 2, d = 0

$(x - 2)(x - 2)^{0+1} = 0$ → r = 2 doble

$T(2^q) = c_0 2^q q^0 + c_1 2^q q^1$ → Deshacer cambio de base → **$T(n) = c_0 n + c_1 n \cdot \log(n) \in O(n \cdot \log(n))$.**

Dado que no se puede conocer cuantas OEs realiza merge en cada llamada, no se pueden calcular las constantes.

Por expansión de recurrencia:

T(n) = 2T(n/2) + k'n → T(n) = 2(2T(n/4) + k'(n/2)) + k'n →

T(n) = 4T(n/4) + 2·k'n

...

$T(n) = 2^iT(n/2^i) + i \cdot k'n$

Hasta llegar al caso base, donde $T(1) = 1 \rightarrow n/2^i = 1 \rightarrow n = 2^i$

$T(n) = n \cdot T(1) + \log(n) \cdot k'n \rightarrow$ **$T(n) = k'n \cdot \log(n) + n \in O(n \cdot \log(n))$.**

Como se puede ver son siempre los mismos pasos para realizar el análisis de cualquier algoritmo.

Algoritmos Voraces

Suelen utilizarse para resolver problemas de optimización. A partir de un conjunto de candidatos, seleccionará los más propensos a alcanzar una solución, que no tiene porqué ser la solución óptima.

Su estructura general es la siguiente:

```
funcion voraz(candidatos)
    Solucion ← conjunto vacío
    mientras (no solución) hacer // Bucle voraz
        a ← candidatoMasPrometedor(candidatos) // Selecciona el más prometedor
                                               // según un criterio
        añadir(Solucion, a)
    fmientras
    devuelve Solucion
ffuncion
```

En función del criterio utilizado, el algoritmo puede dar una solución u otra para el mismo problema, que no tiene porqué ser la óptima, o incluso puede no llegar a dar una solución (aunque exista).

Los tres tipos de problemas son:

- Mochila 0-1. Meter en una mochila con capacidad máxima M, una serie de artículos con un peso y un beneficio, con la condición de no superar la capacidad máxima, y obtener el máximo beneficio. Los artículos no pueden ser fragmentados. Un posible algoritmo sería:

```
funcion mochila(P(1...N), B(1...N) : entero, M : entero)
    S(1...N) ← (0, ... 0) // Conjunto solución
    pesoAct ← 0
    ordenados(1...N) ← ordenar(P, B) // Ordena los artículos en función del
    beneficio/peso, devolviendo un array con los índices de las posiciones
    i ← 1
    mientras (i <= N Y pesoAct <= M) hacer
        pos ← ordenados(i) // articulo con mayor b/p en cada iteración
        peso ← P(pos)
        si (pesoAct + peso <= M) entonces
            S(pos) ← 1
            pesoAct ← pesoAct + peso
        fsi
        i ← i + 1
    fmientras
    devuelve S
ffuncion
```

  Ejemplo de traza del algoritmo:

  Para un conjunto de artículos con peso y beneficio: P = (4, 2, 6) y B = (5, 3, 6), y capacidad M = 10:
  5/4 = 1.2; 3/2 = 1.5; 6/6 = 1

ordenados = (2, 1, 3) // de mayor a menor beneficio/peso, esos son los índices de sus posiciones reales
pesoAct = 0
seleccionado articulo 2, ¿pesoAct + P(2) <= M? → SI →se coge → S = (0, 1, 0)
pesoAct = 2
seleccionado articulo 1, ¿pesoAct + P(1) <= M? → SI → se coge → S = (1, 1, 0)
pesoAct = 6
seleccionado articulo 3, ¿pesoAct + P(3) <= M? → NO → no se coge
ya no quedan artículos
S = (1, 1, 0) → Beneficio = 5 + 3 = 8
¿Es la solución óptima, corresponde con el máximo beneficio posible? → NO
S = (1, 0, 1) → Beneficio = 5 + 6 = **11**, peso = 4 + 6 = 10 <= M
Queda demostrado así que el algoritmo funciona, pero no siempre dará la solución óptima.

- Cambio de moneda. Dada una cantidad y una serie de monedas, devolver la misma cantidad utilizando el menor número de monedas posible. Un posible algoritmo sería:

```
funcion moneda(cantidad : entero)
    C ← (x1, x2, ... xn) // Monedas disponibles de mayor a menor (candidatos)
    S(1...N) ← (0, ... 0) // Conjunto solución
    i ← 1
    actual ← 0
    mientras (i <= N Y actual != cantidad) hacer
        j ← C(i) // moneda con mayor valor
        numeroMonedasJ ← (cantidad – actual)/C(j) // Cociente de la división
        S(j) ← numeroMonedasJ
        actual ← actual + numeroMonedasJ * C(j)
    fmientras
    si (actual != cantidad) entonces
        devuelve "No existe solución"
    sino
        devuelve S
    fsi
ffuncion
```

- Selección de actividades. Se tienen n actividades con unos momentos de inicio y fin establecidos, y se deben compaginar el mayor número de actividades posibles sin que ninguna se solape. Un posible algoritmo sería:

```
funcion selectorAct(C(1...N), F(1...N) : entero)
    S(1...N) ← {1} // Conjunto solución (N es el número máximo posible)
    // C(1...N) son los instantes de comienzo de la actividad i-esima
    // F(1...N) son los instantes de finalización de la actividad i-esima
    // Tanto C como F están ordenados según el criterio: (i<j ⇒ f(i)≤f(j))
    z ← 1 // Última actividad seleccionada, inicialmente es la 1
    para i = 2 hasta n (inc 1) hacer
        si (C(i) >= F(z)) entonces
            S ← S U {i}
            z ← i
        fsi
    fpara
    devuelve S
ffuncion
```