Programación 2

Lección 9. Caracterización asintótica de la eficiencia de algoritmos recursivos

1. Eficiencia y ecuaciones recurrentes
2. Resolución de recurrencias lineales de coeficientes constantes
   - Método de resolución y su aplicación a las funciones `factorial(n)`, `fibonacci(n)` y `torresHanoi(n,origen,destino auxiliar)`
3. Resolución de recurrencias mediante cambio de variable
   - Método de resolución y su aplicación a las funciones `buscar(v,n,dato)` y `mcd(a,b)`
4. Análisis comparativo de costes de algoritmos que resuelven un mismo problema
   - La función `raiz(n)`

1. Eficiencia y ecuaciones recurrentes

```
/*
 *  Pre:  n >= 0
 *  Post: factorial(n) = (PROD alfa EN [1,n].alfa)
 */
int factorial (const int n) {
   if (n == 0) {
     // n = 0
     return 1;
     // factorial(n) = 1
   }
   else {
     // n > 0
     return n * factorial(n-1);
     // factorial(n) = (PROD alfa EN [1,n].alfa)
   }
}
```

Coste en memoria

```
/*
 *  Pre:  n >= 0
 *  Post: factorial(n) = (PROD alfa EN [1,n].alfa)
 */
int factorial (const int n) {
   if (n == 0) {                                    //  a
      return 1;                                     //  b
   }
   else {
      return n * factorial(n-1);                    //  c
   }
}
```

La función que determina el coste de la memoria, **mem(n)**, necesaria para ejecutar una invocación **factorial(n)** es la solución del sistema:

- $n = 0 \rightarrow mem(0) = mem_{inv.} + 2 \times mem_{int}$
- $n > 0 \rightarrow mem(n) = mem_{inv.} + 2 \times mem_{int} + mem(n-1)$

La solución del sistema:

- $n = 0 \rightarrow mem(0) = mem_{inv.} + 2 \times mem_{int}$
- $n > 0 \rightarrow mem(n) = mem_{inv.} + 2 \times mem_{int} + mem(n-1)$

es la siguiente:

$$n \geq 0 \rightarrow mem(n) = (mem_{inv.} + 2 \times mem_{int}) \times (n + 1)$$

y, por lo tanto:

$$\boldsymbol{O(mem(n))} = \boldsymbol{O}((mem_{inv.} + 2 \times mem_{int}) \times (n + 1)) = \boldsymbol{O(n)}$$

Es decir, la función de coste en memoria de una invocación **`factorial(n)`** es asintóticamente **lineal** en el valor de ***n***.

Veremos que, para llegar a esta caracterización asintótica de la función de coste, **no es necesario conocer la solución exacta** del sistema de ecuaciones de partida.

Coste en tiempo

```
/*
 *  Pre:  n >= 0
 *  Post: factorial(n) = (PROD alfa EN [1,n].alfa)
 */
int factorial (const int n) {
   if (n == 0) {                                    //  a
      return 1;                                     //  b
   }
   else {
      return n * factorial(n-1);                    //  c
   }
}
```

La función que determina el coste en tiempo, **t(n)**, necesario para ejecutar una invocación **factorial(n)** es la solución del sistema:

- $n = 0 \rightarrow t(0) = t_{inv.} + t_a + t_b$
- $n > 0 \rightarrow t(n) = t_{inv.} + t_a + t_c + t(n-1)$

La solución del sistema:

- $n = 0 \rightarrow t(0) = t_{inv.} + t_a + t_b$
- $n > 0 \rightarrow t(n) = t_{inv.} + t_a + t_c + t(n-1)$

es la siguiente:

- $n \geq 0 \rightarrow t(n) = (t_{inv.} + t_a + t_c) \times n + t_{inv.} + t_a + t_b$

y, por lo tanto:

- $O(t(n)) = O((t_{inv.} + t_a + t_c) \times n + t_{inv.} + t_a + t_b) = O(n)$

Es decir, la función de coste en tiempo de una invocación **`factorial(n)`** es asintóticamente **lineal** en el valor de **n**.

Veremos que, para llegar a esta caracterización asintótica de la función de coste, **no es necesario conocer la solución exacta** del sistema de ecuaciones de partida.

2. Resolución de recurrencias lineales de coeficientes constantes

Partimos de una recurrencia de coeficientes constantes de la forma:

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

que forma parte de un sistema de ecuaciones que define la función de coste del algoritmo, **t(n)**, que depende de un parámetro, "**n**" en este caso.

Nuestro objetivo no es la resolución del sistema, sino analizar la complejidad del algoritmo caracterizando el orden $O(t(n))$.

Como paso previo, presentaremos un método para resolver la recurrencia anterior en el caso de que sea **homogénea**, es decir, cuando **f(n) = 0**

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

Y explicaremos cómo generalizar el método cuando **f(n)** no sea una función nula.

2.1 Resolución de recurrencias lineales homogéneas

Sea la recurrencia homogénea de coeficientes constantes:

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

Sus soluciones pueden expresarse de la forma $t(n) = x^n$

$$a_0.x^n + a_1.x^{n-1} + \ldots + a_k.x^{n-k} = 0$$

Sacando x^{n-k} como factor común:

$$(a_0.x^k + a_1.x^{k-1} + \ldots + a_k).x^{n-k} = 0$$

Las **n** raíces **x**, soluciones de la ecuación anterior, son:

- (n-k) raíces $x = 0$ que definen las soluciones $t(n) = x^n = 0^n = 0$ que no interesan ya que el coste asintótico de un algoritmo no puede ser nulo.

- k raíces $\{ r_1, r_2, \ldots , r_k \}$ de la **ecuación característica**:
  $a_0.x^k + a_1.x^{k-1} + \ldots + a_k = 0$
  que definen k soluciones $t(n) = r_1^n, \ldots , t(n) = r_k^n$

La ecuación características tiene **k** raíces no nulas:

- Cada una de las **k** raíces no nulas $\{r_1, r_2, \ldots r_k\}$ de la ecuación característica define un solución de la recurencia. Cada raíz $\mathbf{r_i}$ tiene una multiplicidad **m** (número de veces que está repetida), mayor o igual que uno.
- Cada raíz real $\mathbf{r_i}$ con multiplicidad **m** define la siguiente solución, **t(n)**:

$$t(n) = \Sigma_{\alpha \in [1,m]}\ c_\alpha . n^{m-\alpha} . r_i^n$$

$$= c_1 . n^{m-1} . r_i^n + c_2 . n^{m-2} . r_i^n + \ldots + c_m . n^0 . r_i^n$$

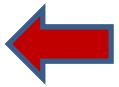
- Forma de las soluciones de las diferentes raíces, según su multiplicidad:
  - Si la raíz $\mathbf{r_i}$ es **real y simple** la solución es de la forma: $t(n) = c_1 . r_i^n$
  - Si la raíz $\mathbf{r_i}$ es **real y doble** entonces quedan definidas soluciones de la forma: $t(n) = c_1 . n . r_i^n + c_2 . r_i^n$
  - Si la raíz $\mathbf{r_i}$ es **real y triple**: $t(n) = c_1 . n^2 . r_i^n + c_2 . n . r_i^n + c_3 . r_i^n$
  - Etc., etc.
- La solución general de la recurrencia se expresa como una combinación lineal de todas las soluciones deducidas de las raíces no nulas de su ecuación característica.

2.2 Resolución de recurrencias lineales no homogéneas

En general, las recurrencias que definen costes de algoritmos no son homogéneas:

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

Así, la recurrencia que define la función de coste, **t(n)**, de una invocación **factorial(n)** tiene la forma:

- $n = 0 \rightarrow \quad t(0) = t_a + t_b$
- $n > 0 \rightarrow \quad \mathbf{t(n) - t(n-1) = t_{inv} + t_a + t_c}$ 

No interesa calcular exactamente la función de coste **t(n)**, ya que es suficiente caracterizarla asintóticamente de la forma ***O*(t(n)**). Para ello basta considerar la recurrencia señalada por la fecha roja.

En el caso general nos encontraremos con recurrencias de coeficientes constantes no homogéneas:

$$\mathbf{a_0.t(n) + a_1.t(n-1) + \ldots + a_k.t(n-k) = f(n)}$$

En el caso general nos encontraremos con recurrencias de coeficientes constantes no homogéneas:

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

En gran parte de los casos que se nos presentan, estas recurrencias tiene una solución sencilla, por ejemplo, si $f(n) = B^n.p(n)$, donde **B** es una constante real y $p(n)$ es un polinomio en n de grado **d**:

$$f(n) = B^n.p(n) = B^n.(a_d.n^d + a_{d-1}.n^{d-1} + \ldots + a_0)$$

En algunos casos, el valor de **B** puede ser 1.0:

$$f(n) = 1.0^n.p(n) = 1.0^n.(a_d.n^d + a_{d-1}.n^{d-1} + \ldots + a_0)$$

$$= a_d.n^d + a_{d-1}.n^{d-1} + \ldots + a_0$$

En tales casos, la recurrencia a resolver es la siguiente:

$$a_0.t(n) + a_1.t(n-1) + \ldots + a_k.t(n-k) = B^n.(a_d.n^d + a_{d-1}.n^{d-1} + \ldots + a_0)$$

Resolución de recurrencias no homogéneas de la forma:

$$a_0.t(n) + a_1.t(n-1) + \ldots + a_k.t(n-k) = B^n.p(n)$$
$$= B^n.(a_d.n^d + a_{d-1}.n^{d-1} + \ldots + a_0)$$

Las soluciones pueden expresarse de la forma $t(n) = x^n$. Para calcularlas debemos escribir la siguiente ecuación característica:

$$(a_0.x^k + a_1.x^{k-1} + \ldots + a_k).(x - B)^{d+1} = 0$$

Se calculan las raíces reales $\mathbf{r_i}$ de esta ecuación y se puede escribir directamente la función de coste **t(n)** como combinación lineal de las soluciones aportadas por cada una de las raíces calculadas, del mismo modo que en la resolución de una recurrencia homogénea.

2.3. Aplicación a la caracterización asintótica del coste en tiempo de la invocación *factorial(n)*

$$n > 0 \rightarrow t_{factorial}(n) = t_{inv.} + t_a + t_c + t_{factorial}(n-1)$$

Reordenamos la ecuación e identificamos su miembro derecho:

$$\mathbf{t_{factorial}(n) - t_{factorial}(n-1) = t_{inv.} + t_a + t_c} =$$

$$1^n.(t_{inv.} + t_a + t_c) = B^n.p(n) \rightarrow B = 1 \wedge d = 0$$

Escribimos la ecuación característica:

$$(a_0.x^k + a_1.x^{k-1} + \ldots + a_k).(x-B)^{d+1} = 0$$

$$\mathbf{(x-1).(x-1)^{0+1} = 0}, \quad \text{es decir,} \quad (x-1)^2 = 0$$

Tiene una raíz doble $x = 1$. Por lo tanto, la solución de la recurrencia será:

$$\mathbf{t_{factorial}(n)} = c_1.n^{2-1}.r^n + c_2.n^{2-2}.r^n = c_1.n^1.1^n + c_2.n^0.1^n$$

$$= \mathbf{c_1.n + c_2}$$

Finalmente podemos deducir que el coste del algoritmo, en tiempo, es asintóticamente de lineal en **n**:

$$\boldsymbol{O}(t_{factorial}(n)) = \boldsymbol{O}(c_1.n + c_2) = \boldsymbol{O(n)}$$

Aplicación a la caracterización asintótica de la función de coste en memoria, `mem(n)`, de la invocación `factorial(n)`

$$n > 0 \rightarrow mem(n) = mem_{inv.} + 2 \times mem_{int} + mem(n-1)$$

Reordenamos la ecuación e identificamos su miembro derecho:

$$mem(n) - mem(n-1) = 1^n.(mem_{inv.} + 2 \times mem_{int})$$
$$= B^n.p(n) \rightarrow B = 1 \wedge d = 0$$

Escribimos la ecuación característica:

$$(x-1).(x-1)^{0+1} = 0, \quad \text{es decir,} \quad (x-1)^2 = 0$$

Tiene una raíz doble **x = 1**. Por lo tanto la solución será:

$$mem(n) = c_1.n^{2-1}.r^n + c_2.n^{2-2}.r^n = c_1.n^1.1^n + c_2.n^0.1^n$$
$$= c_1.n + c_2$$

Finalmente podemos deducir que la función de coste del algoritmo, en memoria, es asintóticamente de lineal en **n**:

$$O(mem(n)) = O(c_1.n + c_2) = O(n)$$

2.4 Aplicación a la caracterización asintótica de las funciones de coste en tiempo y memoria de la invocación `fibonacci(n)`

Coste en tiempo

```
/*
 *  Pre:  n >= 1
 *  Post: (n <= 2 -> fibonacci(n) = n - 1)
 *        AND (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1))
 */
int fibonacci (const int n) {
   if (n <= 2)                                                  //  a
   {  return n - 1;  }                                          //  b
   else
   {  return fibonacci(n-2) + fibonacci(n-1);  }                //  c
}
```

¿Cuál es la función de coste, en tiempo, de la invocación **`fibonacci(i)`**?

- $i \leq 2 \rightarrow t(i) = t_{inv.} + t_a + t_b$
- $i > 2 \rightarrow t(i) = t_{inv.} + t_a + t_c + t(i-2) + t(i-1)$

Debemos resolver la recurrencia:

$$t(i) = t_{inv.} + t_a + t_c + t(i-2) + t(i-1)$$

Reordenamos la ecuación e identificamos su miembro derecho:

$$t(i) - t(i-1) - t(i-2) = t_{inv.} + t_a + t_c = 1^i.(t_a + t_c) =$$

$$B^i.p(i) \rightarrow B = 1 \wedge d = 0$$

Escribimos la ecuación característica:

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

es decir,

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

¿Cuáles son las tres raíces de la ecuación característica?

$$t(i) - t(i-1) - t(i-2) = t_{inv.} + t_a + t_c = 1^i.(t_a + t_c) =$$
$$K^i.p(i) \rightarrow K = 1 \wedge d = 0$$

Escribimos la ecuación característica:

$$(x^2 - x - 1).(x-1)^{0+1} = 0, \text{ es decir, } (x^2 - x - 1)(x-1) = 0$$

Tiene tres raíces reales simples $\mathbf{x = 1}$, $\mathbf{x = (1 + \sqrt{5}) / 2}$ y $\mathbf{x = (1 - \sqrt{5}) / 2}$.

Por lo tanto la función de coste, en tiempo, de **`fibonacci(i)`** será:

$$\mathbf{t(i) = c_1.1^i + c_2.[(1 + \sqrt{5}) / 2]^i + c_3.[(1 - \sqrt{5}) / 2]^i}$$

Y, por lo tanto, su caracterización asintótica será:

$$\mathbf{O(t(i))} = \mathbf{O}(c_1.1^i + c_2.[(1 + \sqrt{5}) / 2]^i + c_3.[(1 - \sqrt{5}) / 2]^i)$$
$$= \mathbf{O}([(1 + \sqrt{5}) / 2]^i) = \mathbf{O}(1.618^i)$$

¡¡¡ Nos encontramos ante un algoritmo cuyo coste, en tiempo, es asintóticamente **exponencial** en `i`. Entra en la categoría de algoritmos que calificamos como **intratables** !!!

Coste en memoria

```
/*
 *  Pre:  n >= 1
 *  Post: (n <= 2 -> fibonacci(n) = n - 1)
 *        AND (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1))
 */
int fibonacci (const int n) {
   if (n <= 2) {  return n-1;  }
   else {  return fibonacci(n-2) + fibonacci(n-1);  }
}
```

¿Cuál es la función de coste, en memoria, de una invocación **`fibonacci(i)`**?

- $i \leq 2 \rightarrow mem(i) = mem_{inv.} + 2 \times mem_{int}$
- $i > 2 \rightarrow mem(i) = mem_{inv.} + 2 \times mem_{int} + mem(i-1)$

Recurrencia análoga a la resuelta al analizar la función de coste, en tiempo, de ***`factorial(n)`***:

$$O(mem_{fibonacci}(i)) = O(c_1.i + c_2) = O(i)$$

2.5 Aplicación a la caracterización asintótica del coste en tiempo y memoria de una invocación a la función *torresHanoi(n,origen,destino,auxiliar)*

```
/*
 *  Pre:  n >= 0
 *  Post: Presenta por pantalla un listado con la secuencia de
 *        movimientos de discos a realizar para resolver el problema
 *        de las torres de Hanoi para trasladar una pirámide con [n]
 *        discos desde la torre [origen] hasta la torre [destino],
 *        utilizando como auxiliar la torre [auxiliar]
 */
void torresHanoi (const int n, const char origen[], const char destino[],
                  const char auxiliar[]) {
   if (n > 0) {
      torresHanoi(n-1, origen, auxiliar, destino);
      cout << "Mover el disco superior de la torre " << origen
           << " y apilarlo en la cima de la torre " << destino
           << endl;
      torresHanoi(n-1, auxiliar, destino, origen);
   }
}
```

Coste en tiempo

```
/*
 *  Pre:  n >= 0
 *  Post: Presenta por pantalla un listado con la secuencia ...
 */
void torresHanoi (const int n, const char origen[], const char destino[],
                  const char auxiliar[]) {
   if (n > 0) {                                                    //  a
      torresHanoi(n-1, origen, auxiliar, destino);                 //  b
      cout << "Mover el disco superior de la torre " << origen
           << " y apilarlo en la cima de la torre " << destino
           << endl;                                                //  c
      torresHanoi(n-1, auxiliar, destino, origen);                 //  d
   }
}
```

¿Cuál es el coste de una invocación **torresHanoi(nD, “Sol”, “Luna”, “Tierra”)**?

- $nD = 0 \rightarrow t(nD) = t_{inv.} + t_a$
- $nD > 0 \rightarrow t(nD) = t_{inv.} + t_a + t(nD-1) + t_c + t(nD-1)$

Debemos resolver la recurrencia:

$$\mathbf{t(nD) = t_{inv.} + t_a + t(nD-1) + t_c + t(nD-1)}$$

Agrupamos y reordenamos la ecuación e identificamos su miembro derecho:

$$t(nD) - 2 \times t(nD-1) = t_{inv.} + t_a + t_c = 1^{nD}.(t_a + t_c) = B^{nD}.p(nD)$$

$$\rightarrow \quad B = 1 \wedge d = 0$$

Escribimos la ecuación característica:

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

es decir,

$$\mathbf{(x-2).(x-1) = 0}$$

Tiene dos raíces reales simples **x = 2** y **x = 1**. Por lo tanto la solución será:

$$t(nD) = c_1.2^{nD} + c_2.1^{nD} = c_1.2^{nD} + c_2$$

Y, por lo tanto:

$$\boldsymbol{O}(t(nD)) = \boldsymbol{O}(c_1.2^{nD} + c_2) = \boldsymbol{O}(2^{nD})$$

La función de **coste** en tiempo es asintóticamente **exponencial** en **nD**

Coste en memoria

```
/*
 *  Pre:  n >= 0
 *  Post: Presenta por pantalla un listado con la secuencia ...
 */
void torresHanoi (const int n, const char origen[], const char destino[],
                  const char auxiliar[]) {
   if (n > 0) {
      torresHanoi(n-1, origen, auxiliar, destino);
      cout << "Mover el disco superior de la torre " << origen
           << " y apilarlo en la cima de la torre " << destino
           << endl;
      torresHanoi(n-1, auxiliar, destino, origen);
   }
}
```

¿Cuál es el coste de una invocación **torresHanoi(nD, “Sol”, “Luna”, “Tierra”)**?

- $nD = 0 \rightarrow mem(nD) = mem_{inv.} + mem_{int} + 3 \times mem_{ref}$
- $nD > 0 \rightarrow mem(nD) = mem_{inv.} + mem_{int} + 3 \times mem_{ref} + mem(nD-1)$

Debemos resolver la recurrencia:

$$mem(nD) = mem_{inv.} + mem_{int} + 3 \times mem_{ref} + mem(nD-1)$$

Idéntica a la resuelta al analizar el coste de **factorial(n)**. Por lo tanto:

$$O(mem(nD)) = O(c_1.nD + c_2) = O(nD)$$

La función de **coste** , en memoria, de una invocación:

```
torresHanoi(nD, "Sol", "Luna", "Tierra")
```

es asintóticamente **lineal** en **nD**

3. Resolución de recurrencias mediante cambio de variable

En ocasiones se nos presentan recurrencias del tipo:

$$t_{función}(n) = t_{función}(n / k) + f(n)$$

que no pueden resolverse aplicando directamente el método expuesto anteriormente.

Se pueden resolver aplicando el **cambio de variable**:

$$m = \log_k n \leftrightarrow n = k^m \wedge n / k = k^{m-1} \wedge \log_k(n / k) = m - 1$$

Que nos facilita la siguiente recurrencia:

$$t_{función}(m) - t_{función}(m-1) = f'(m)$$

Donde $f'(m)$ es la función resultante de sustituir en $f(n)$ cada ocurrencia de n por k^m.

A menudo, esta última recurrencia puede ser resuelta aplicando el método presentado anteriormente. Tras ello deberemos deshacer el cambio de variable.

a) Aplicación al caso de la función de búsqueda binaria *buscar(v,izdo,dcho,dato)*

```
/*
 *  Pre: izdo > 0 AND izdo <= dcho AND dcho < #v
 *       AND (PT alfa EN [izdo,dcho-1]. v[alfa] <= v[alfa+1])
 *  Post: ((EX alfa EN [izdo,dcho].v[alfa] = dato)
 *                          -> v[buscar(v,izdo,dcho,dato)] = dato) AND
 *        (PT alfa EN [izdo,dcho].v[alfa] != v[alfa+1]
 *                          -> buscar(v,izdo,dcho,dato) < 0)
 */
int buscar (const int v[], const int dcho, const int izdo,
            const int dato) {
   if (izdo == dcho)
      if (v[izdo] == dato) { return izdo; }
      else { return -1; }
   else {
      int medio = (izdo + dcho) / 2;
      if (v[medio] < dato) { return buscar(v, medio + 1, dcho, dato); }
      else { return buscar(v, izdo, medio, dato); }
   }
}
```

```
// Pre: izdo > 0  AND  izdo <= dcho  AND  dcho < #v  AND  …
// Post: ((EX alfa EN [izdo,dcho]. v[alfa] = dato) …
int buscar (const int v[], const int dcho, const int izdo,
            const int dato) {
   if (izdo == dcho)                                        //  a
      if (v[izdo] == dato)                                  //  b
      { return izdo; }                                      //  c
      else { return -1; }                                   //  d
   else {
      int medio = (izdo + dcho) / 2;                        //  e
      if (v[medio] < dato)                                  //  f
      { return buscar(v, medio+1, dcho, dato); }            //  g
      else { return buscar(v, izdo, medio, dato); }         //  h
   }
}
```

Coste en tiempo

¿Cuál es el coste en tiempo de una invocación **buscar(v,0,n-1,dato)**?

Donde **n** es igual al número de datos sobre los que se plantea la búsqueda.

- $n = 1 \rightarrow t(n) = t_{inv.} + t_a + t_b + [t_c \mid t_d]$
- $n > 1 \rightarrow t(n) = t_{inv.} + t_a + t_e + t_f + t(n/2)$

Debemos resolver la recurrencia:

$$\mathbf{t(n) = t_{inv.} + t_a + t_e + t_f + t(n/2)}$$

Reordenamos la ecuación:

$$t(n) - t(n/2) = t_{inv.} + t_a + t_e + t_f$$

La recurrencia no tiene la forma adecuada. Podemos llegar a una recurrencia adecuada haciendo el cambio de variable;

$$m = \log_2 n \;\rightarrow\; n = 2^m \;\wedge\; n/2 = 2^{m-1} \;\wedge\; \log_2(n/2) = m - 1$$

Podemos reescribir la ecuación recurrente en función de la variable **m**:

$$\mathbf{t(m) - t(m\text{-}1) = t_{inv.} + t_a + t_e + t_f} =$$

$$1^m.(t_{inv.} + t_a + t_e + t_f) = B^m.p(m) \;\rightarrow\; B = 1 \wedge d = 0$$

La ecuación característica será:

$$\mathbf{(x - 1).(x - 1)^{0+1} = 0}$$

Reescribimos la ecuación característica:

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

Tiene una raíz real doble, **x=1**, por lo tanto la solución de la recurrencia, en **m**, proporciona la siguiente función de coste:

$$t(m) = c_1.m^1.1^m + c_2.m^0.1^m = c_1.m + c_2$$

Deshacemos el cambio de variable $[m = \log_2 n]$:

$$\mathbf{t\,(n) = c_1.\log_2 n + c_2}$$

Por lo tanto:

$$\boldsymbol{O}(t(n)) = \boldsymbol{O}(c_1.\log_2 n + c_2) = \boldsymbol{O}(\log_2 n) = \boldsymbol{O}(\log n)$$

Nos encontramos ante un algoritmo cuyo coste en tiempo es asintóticamente **logarítmico** en el número de datos, **n**, sobre los que se plantea la búsqueda.

```
// Pre: izdo > 0  AND  izdo <= dcho  AND  dcho < #v  AND  …
// Post: ((EX alfa EN [izdo,dcho]. v[alfa] = dato) …
int buscar (const int v[], const int dcho, const int izdo,
            const int dato) {
   if (izdo == dcho)
      if (v[izdo] == dato) { return izdo; }
      else { return -1; }
   else {
      int medio = (izdo + dcho) / 2;
      if (v[medio] < dato) { return buscar(v, medio+1, dcho, dato); }
      else { return buscar(v, izdo, medio, dato); }
   }
}
```

Coste en memoria

¿Cuál es el coste en memoria de una invocación **buscar(T,0,n-1,dato)**?

Donde **n** es igual al número de datos sobre los que se plantea la búsqueda.

- $n = 1 \rightarrow mem(n) = mem_{inv.} + mem_{ref.} + 4 \times mem_{int}$
- $n > 1 \rightarrow mem(n) = mem_{inv.} + mem_{ref.} + 5 \times mem_{int} + mem(n/2)$

Debemos resolver la recurrencia [sea n = dcho – izdo + 1]:

$$\mathbf{mem(n) = mem_{inv.} + mem_{ref.} + 5 \times mem_{int} + mem(n/2)}$$

Que es análoga a la recurrencia que acaba de ser resuelta:

$$t(n) - t(n/2) = t_{inv.} + t_a + t_e + t_f$$

Por lo tanto, la función de coste en memoria será:

$$\boldsymbol{O(mem(n))} = \boldsymbol{O}(c_1.\log_2 n + c_2) = O(\log_2 n) = \boldsymbol{O(\log n)}$$

Nos encontramos ante un algoritmo cuyo coste en memoria es asintóticamente **logarítmico** en el número de datos, **n**, sobre los que se plantea la búsqueda

b) Aplicación al caso de la función *mcd(a,b)*

```
/*
 *  Pre:  a >= 0 AND b >= 0 AND (a != 0 OR b != 0)
 *  Post: a % mcd(a,b) = 0 AND b % mcd(a,b) = 0 AND mcd(a,b) >= 1
 *        AND (PT alfa EN [2,mcd(a,b)-1].a%alfa!=0 OR b%alfa!=0)
 */
int mcd (const int a, const int b) {
   if (b == 0)                                          //  a
   {  return a;  }                                      //  b
   else
   {  return mcd(b, a % b);  }                          //  c
}
```

El resto **a % b** es un valor del intervalo **[0,b-1]**

→ **Casos más favorables**: el valor **a % b** está próximo a **0** ó a **b - 1**

→ **Casos más desfavorables**: el valor **a % b** está próximo a **b / 2**

Para caracterizar la función de coste en el **caso peor** de una invocación ***mcd(a,b)*** debemos resolver la recurrencia:

$$t_{mcd}(b) = t_{inv.} + t_a + t_c + t_{mcd}(b/2)$$

Reordenamos la ecuación:

$$t_{mcd}(b) - t_{mcd}(b/2) = t_{inv.} + t_a + t_c$$

Esta recurrencia es análoga a la del problema anterior, por lo tanto podemos escribir directamente su solución:

$$O(t_{mcd}(b)) = O(\log_2 b)$$

Nos encontramos ante un algoritmo cuyo coste asintótico **en el caso peor** es **logarítmico** en el valor de su segundo parámetro.

Curiosidad: El ***teorema de Lamé*** afirma que el caso peor para este algoritmo se presenta cuando se le invoca para calcular el máximo común divisor de dos números consecutivos de la sucesión de Fibonacci.

4. Análisis comparativo de costes en tiempo de algoritmos que resuelven un mismo problema

Análisis del coste en tiempo de cada uno de los diseños del método ***raiz(n)***:

- Diseñado por inmersión mediante fortalecimiento de su precondición (alternativa 1)
- Diseñado por inmersión mediante fortalecimiento de su precondición (alternativa 2)
- Diseñado por inmersión mediante debilitamiento de su postcondición (alternativa 3)

Análisis del coste del primer diseño: *raiz01*

```
/*
 *  Pre:  n >= 0
 *  Post: raiz01(n) = R AND R^2 <= n AND (R+1)^2 > n
 */
int raiz01 (const int n) {  return raiz1(n,0);  }          //  a
```

```
/*
 *  Pre:  n >= 0 AND candidato^2 <= n
 *  Post: raiz1(n,candidato) = R AND R^2 <= n AND (R+1)^2 > n
 */
int raiz1 (const int n, const int candidato) {
   if ((candidato + 1)*(candidato + 1) > n)                 //  b
   {  return candidato;  }                                  //  c
   else
   {  return raiz1(n, candidato + 1);  }                    //  d
}
```

Deseamos analizar el coste en tiempo de una invocación **raiz01(n)**:

$$t_{raiz01(n)} = t_{inv.} + t_a + t_{raiz1(n,0)}$$

Las funciones de coste $t_{raiz01(n)}$ y de $t_{raiz1(n,0)}$ son dependientes del valor de **n**.

$$t_{raiz1(n)}(n) = t_{inv.} + t_a + t_{raiz1(n,0)}(n)$$

La secuencia de invocaciones recursivas de **raiz1(n,candidato)** es:

$$raiz1(n,0) \rightarrow raiz1(n,1) \rightarrow \ldots \rightarrow raiz1(n,\sqrt{n}\,)$$

Haciendo **m** = $\lfloor\sqrt{n}\,\rfloor$ - **candidato** podemos escribir una recurrencia para determinar la función de coste de **raiz1(n,0)**, la función **t(m)**, dependiente de la nueva variable **m**:

$$t(m) = t_{inv.} + t_b + t_d + t(m-1)$$

Es decir:

$$t(m) - t(m-1) = t_{inv.} + t_b + t_d$$

Que ya hemos resuelto anteriormente

La recurrencia:

$$t(m) - t(m-1) = t_{inv.} + t_b + t_d$$

La hemos resuelto anteriormente. Podemos escribir directamente su solución:

$$t(m) = c_1 + c_2.m$$

Es decir:

$$t_{raiz1(n,0)}(m) = c_1 + c_2.m$$

Es momento de deshacer el cambio $\mathbf{m = \sqrt{n} - candidato}$:

$$t_{raiz1(n,0)}(m) = c_1 + c_2.m \rightarrow t_{raiz1(n,0)}(n) = c_1 + c_2.(\sqrt{n} - 0)$$

Y, por lo tanto:

$$O(t_{raiz01(n)}(n)) = O(t_a + t_{raiz1(n,0)}(n))$$

$$= O(t_a + c_1 + c_2.\sqrt{n}) = O(\sqrt{n})$$

Análisis del coste del segundo diseño: *raiz02(n)*

```
/*
 *  Pre:  n >= 0
 *  Post: raiz02(n) = R AND R^2 <= n AND (R+1)^2 > n
 */
int raiz02 (const int n) { return raiz2(n,n); }              //  a
```

```
/*
 *  Pre:  n >= 0 AND (candidato+1)^2 > n
 *  Post: raiz2(n,candidato) = R AND R^2 <= n AND (R+1)^2 > n
 */
int raiz2 (const int n, const int candidato) {
   if (candidato*candidato <= n)                             //  b
   {  return candidato;  }                                   //  c
   else
   {  return raiz2(n, candidato-1);  }                       //  d
}
```

Deseamos analizar el coste en tiempo de una invocación **raiz02(n)**:

$$t_{raiz02(n)} = t_{inv.} + t_a + t_{raiz2(n,n)}$$

Las funciones de coste $t_{raiz02(n)}$ y de $t_{raiz2(n,n)}$ son dependientes del valor de **n**.

$$t_{raiz02(n)}(n) = t_{inv.} + t_a + t_{raiz2(n,n)}(n)$$

La secuencia de invocaciones recursivas de **raiz2(n,candidato)** es:

raiz2(n,n) → raiz2(n,n-1) → ... → raiz2(n, $\sqrt{n}$)

Podemos escribir una recurrencia para determinar la función de coste de **raiz2(n,n)**, la función **t(n)**, dependiente de **n**:

$$t(n) = t_{inv.} + t_b + t_d + t(n-1)$$

Es decir:

$$t(n) - t(n-1) = t_{inv.} + t_b + t_d$$

Que ya hemos resuelto anteriormente y cuya solución es:

Ya podemos determinar la función de coste en tiempo de una invocación **raiz2(n)**:

$$t_{raiz02(n)}(n) = t_{inv.} + t_a + t_{raiz2(n,n)}(n)$$

$$= t_{inv.} + t_a + c_1 + c_2.n$$

Y, por lo tanto:

$$O(t_{raiz02(n)}(n)) = O(t_{inv.} + t_a + c_1 + c_2.n) = O(n)$$

Análisis del coste del tercer diseño: *raiz03(n)*

```
/*
 *  Pre:  n >= 0
 *  Post: raiz03(n) = R AND R^2 <= n AND (R+1)^2 > n
 */
int raiz03 (const int n) { return raiz3(n,1);  }                    //  a
```

```
/*
 *  Pre:  n >= 0 AND a >= 1
 *  Post: raiz3(n,a) = R AND R^2 <= n AND (R+a)^2 > n
 */
int raiz3 (const int n, const int a) {
   if (a * a > n)                                                   //  b
   { return 0; }                                                    //  c
   else {
      int candidato = raiz3(n,2*a);                                 //  d
      if ((candidato + a) * (candidato + a) > n)                    //  e
      { return candidato; }                                         //  f
      else  { return candidato + a; }                               //  g
   }
}
```

Deseamos analizar el coste en tiempo de una invocación **raiz03(n)**:

$$t_{raiz03(n)} = t_{inv.} + t_a + t_{raiz3(n,1)}$$

Los coste de $t_{raiz03(n)}$ y de $t_{raiz3(n,1)}$ son funciones dependientes del valor de **n.**

$$t_{raiz03(n)}(n) = t_{inv.} + t_a + t_{raiz3(n,1)}(n)$$

La secuencia de invocaciones recursivas a **raiz03(n,a)** es:

$$raiz3(n,1) \rightarrow raiz3(n,2) \rightarrow \ldots \rightarrow raiz3(n,2^{\lfloor \log_2 n \rfloor})$$

Haciendo $m = 2^{\lfloor \log_2 n \rfloor}/a \approx n/a$ podemos escribir la siguiente recurrencia de la función de coste de la invocación **raiz03(n,1)**, como una función **t(m)** dependiente de **m**::

$$t(m) = t_{inv.} + t_b + t_d + t(m/2) + t_e + t_{f|g}$$

Es decir:

$$t(m) - t(m/2) = t_{inv.} + t_b + t_d + t_e + t_{f|g}$$

Que ya hemos resuelto anteriormente y cuya solución es:

$$t(m) = t_{raiz3(n,1)}(m) = c_1 + c_2.\log_2 m$$

Ahora debemos determinar la función de coste en tiempo de una invocación **raiz03(n)**:

$$t_{raiz03(n)} = t + t_a + t_{raiz3(n,1)}$$

Deshacemos el cambio de variable $m = n/a$ teniendo en cuanta que, para la invocación **raiz03(n,1)** , el valor de **a = 1**:

$$t_{raiz3(n,1)}(m) = c_1 + c_2.\log_2 m \rightarrow$$

$$t_{raiz3(n,1)}(n) = c_1 + c_2.\log_2(n/1)$$

Ahora debemos determinar la función de coste en tiempo de una invocación **raiz03(n)**:

$$t_{raiz03(n)} = t_{inv.} + t_a + t_{raiz3(n,1)} =$$

$$t_{inv.} + t_a + c_1 + c_2.\log_2(n/1)$$

Y, por lo tanto:

$$O(t_{raiz03(n)}(n)) = O(t_{inv.} + t_a + t_{raiz3(n,1)}(n)) =$$

$$O(t_{inv.} + t_a + c_1 + c_2.\log_2 n) = O(c_2.\log_2 n) =$$

$$O(\log_2 n) = O(\log n)$$

Comparación de los costes de las tres funciones

Análisis comparativo de costes asintóticos			
Invocación del la función	`raiz01(n)`	`raiz02(n)`	`raiz03(n)`
Orden de la función de coste en tiempo	$O(\sqrt{n})$	$O(n)$	$O(\log n)$
Eficiencia asintótica relativa en tiempo	**	*	***
Orden de la función de coste en memoria	$O(\sqrt{n})$	$O(n)$	$O(\log n)$
Eficiencia asintótica relativa en memoria	**	*	***