Programación 2

Lección 8. Caracterización asintótica de la eficiencia de un algoritmo

1. **Funciones de coste de un algoritmo**
2. **Caracterización asintótica del coste**
   - **Notación *O***
   - **Consecuencias prácticas**
   - **Coste y principio de invariancia**
   - **Reglas para caracterizar el orden de la función de coste**
3. **Ejemplos de caracterización asintótica de la eficiencia de un algoritmo**
   - **Coste asintótico de *factorial(n)***
   - **Coste asintótico de *sumaPares(v,n)***
   - **Coste asintótico de *buscar(v,n,dato)* de búsqueda binaria**
   - **Coste asintótico de *buscar(v,n,dato)* de búsqueda secuencial**

1. Funciones de coste de un algoritmo

Coste de un algoritmo de búsqueda binaria:

- ✓ $mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 6 \times mem_{int}$
- ✓ $t_{buscar(v,n,dato)}(n) = (t_c + t_d + t_e + t_b) \times \log_2 n + t_{invocación} + t_a + t_b + t_g$

Búsqueda binaria

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

$$t_{buscar(v,n,dato)}(n) = (t_c + t_d + t_e + t_b) \times \log_2 n + t_{invocación} + t_a + t_b + t_g$$

Coste en tiempo de un algoritmo de búsqueda binaria:

$$t_{buscar(v,n,dato)}(n) = (t_c + t_d + t_e + t_b) \times \log_2 n + t_{invoc.} + t_a + t_b + t_g$$

Haciendo:

$$k_1 = t_c + t_d + t_e + t_b \quad y$$

$$k_2 = t_{invoc.} + t_a + t_b + t_g$$

entonces, la función de coste del algoritmo es:

$$t_{buscar(v,n,dato)}(n) = k_1 \times \log_2 n + k_2$$

Importante: k_1 y k_2 dependen del computador y su entorno

Funciones de coste de un algoritmo de búsqueda binaria:

- ✓ $\mathbf{mem_{buscar(v,n,dato)}(n)} = mem_{invocación} + mem_{ref} + 6 \times mem_{int}$
  $= \mathbf{k_m}$
- ✓ $\mathbf{t_{buscar(v,n,dato)}(n)} = (t_c + t_d + t_e + t_b) \times \log_2 n +$
  $t_{invoc.} + t_a + t_b + t_g$
  $= \mathbf{k_1 \times \log_2 n + k_2}$

Donde:

- ❑ $k_m = mem_{invocación} + mem_{ref} + 6 \times mem_{int}$
- ❑ $k_1 = t_c + t_d + t_e + t_b$ y
  $k_2 = t_{invoc.} + t_a + t_b + t_g$

Búsqueda secuencial: caso mejor

```
/*
 *  Pre: n <= #v  AND  (EX alfa EN [0,n-1].v[alfa] = dato)
 *  Post: v[buscar(v,n,dato)] = dato
 */
int buscar (const int v[], const int n, const int dato) {
   int indice = 0;                                  //  a
   while (v[indice] != dato) {                      //  b
      indice = indice + 1;                          //  c
   }
   return indice;                                   //  d
}
```

- ✓ $mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$
- ✓ $t_{buscar(v,n,dato)}(n) = t_{invoc.} + t_a + t_b + t_d$

Coste en tiempo de un algoritmo de búsqueda secuencial en el caso mejor:

$$t_{buscar(v,n,dato)} = t_{invoc.} + t_a + t_b + t_d$$

La función de coste es una función constante:

$$t_{buscar(v,n,dato)}(n) = t_{invoc.} + t_a + t_b + t_d$$

Haciendo $k_3 = t_{invoc.} + t_a + t_b + t_d$ entonces, la función de coste del algoritmo es:

$$\mathbf{t_{buscar(v,n,dato)}(n) = k_3}$$

Búsqueda secuencial: caso peor

```
/*
 *  Pre: n <= #v  AND  (EX alfa EN [0,n-1].v[alfa]=dato)
 *  Post: v[buscar(v,n,dato)] = dato
 */
int buscar (int v[], int n, int dato) {
   int indice = 0;                                  //  a
   while (v[indice] != dato) {                      //  b
      indice = indice + 1;                          //  c
   }
   return indice;                                   //  d
}
```

- ✓ $mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$
- ✓ $t_{buscar(v,n,dato)}(n) = t_{invoc.} + t_a + t_b + (\Sigma\alpha\in[1,n-1].t_c + t_b) + t_d$

Coste en tiempo de un algoritmo de búsqueda secuencial en el caso peor:

La función de coste es una función lineal en **n**:

$$t_{buscar(v,n,dato)}(n) = (t_c + t_b) \times n + t_{invoc.} + t_a - t_c + t_d$$

Haciendo: $k_4 = t_c + t_b$ y

$$k_5 = t_{invoc.} + t_a - t_c + t_d$$

entonces, la función de coste del algoritmo es:

$$t_{buscar(v,n,dato)}(n) = k_4 \times n + k_5$$

Función de coste en memoria de una búsqueda secuencial:

- ✓ $mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$
  $= k_m$

Función de coste en tiempo de una búsqueda secuencial:

- ✓ **En el caso mejor:**
  - o $t_{buscar(v,n,dato)}(n) = k_3$
- ✓ **En el caso peor:**
  - o $t_{buscar(v,n,dato)}(n) = k_4 \times n + k_5$

Donde:

- o $k_m = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$
- o $k_3 = t_{invoc.} + t_a + t_b + t_d$,
  $k_4 = t_c + t_b$ y $k_5 = t_{invoc.} + t_a - t_c + t_d$

2. Caracterización asintótica del coste

La notación O

$O(f(n))$ define un conjunto de funciones, aquellas que son acotables superior y asintóticamente (para $n \to \infty$) por $k.f(n)$, siendo k una constante real positiva.

Ejemplos:

$O(1) = \{ 1.0, 45.56, 100.0, 10000.0, 98752223.0, \ldots \}$

$O(n) = \{ 1.5, 500000.0, 3 \times n^{0.5}, 100 \times \log_2 n, 9999 \times \log_{10} n,$
$2500 \times n + 10000000.0, \ldots \}$

$O(n^2) = \{ 1.5, 500000.0, 3.n^{0.5}, 100 \times \log_2 n, 25.n,$
$n.\log_2 n, n^2, 3\,n^2, 1000\,n^2, 5 \times n^2 + 100 \times n, \ldots \}$

Consecuencias prácticas

Sea $t_A(n)$ la función de coste de un algoritmo A que verifica:

$$O(t_A(n)) = O(f(n))$$

Entonces la función de coste $t_A(n)$ se puede aproximar asintóticamente (para $n \to \infty$) a:

$$\lim_{n \to \infty} t_A(n) = k.f(n)$$ donde **k** es una constante real

Si, para un valor de $n = X$ 'suficientemente grande', medimos experimentalmente el coste del algoritmo, podemos deducir **k**:

$$t_A(X) = t_{medido} \approx k.f(X) \quad \rightarrow \quad k = t_{medido} / f(X)$$

Y, por lo tanto, la función de coste será:

$$t_A(n) \approx k.f(n) = (t_{medido} / f(X)).f(n)$$

Ilustración de las consecuencias prácticas

Sea un algoritmo cuya función de coste en la máquina M sea:

$$t(n) = 2.n^2 + 10.n + 5$$

En tal caso:

$$O(t(n)) = O(2.n^2 + 10.n + 5) = O(n^2)$$

Eso significa que:

$$t(n) \approx k.f(n) = k.n^2$$

Ahora podemos deducir el valor de **k**:

$$k = \lim_{n \to \infty} \frac{t(n)}{f(n)} = \lim_{n \to \infty} \frac{2.n^2 + 10.n + 5}{n^2} = 2$$

Y esta es la aproximación calculada:

$$t(n) \approx k.f(n) = k.n^2 = 2.n^2$$

Ilustración de las consecuencias prácticas (cont.)

Función de coste del algoritmo en la máquina M sea:

$$t(n) = 2.n^2 + 10.n + 5$$

¿Es acertada la aproximación calculada?

$$t(n) \approx 2.n^2$$

Así se va a comprobar el grado de acierto:

- Calcularemos los valores exactos y aproximados del coste para n = 10, n = 100, n = 1.000, n = 10.000, etc.
- Calcularemos los errores porcentuales de la aproximación en cada caso

Ilustración de las consecuencias prácticas

Función de coste exacta en la máquina M:

$$t(n) = 2.n^2 + 10.n + 5$$

Función de coste aproximada en la máquina M:

$$t(n) \approx 2.n^2$$

¿Es acertada esta aproximación?

parámetro	valor de t(n)		diferencia
n	exacto	aproximado	error
10	305	200	-50%
100	21.005	20.000	-5%
1.000	2.010.005	2.000.000	-0.5%
10.000	…	…	-0.05%

La notación *O* y la complejidad de un algoritmo

Si $O(t_A(n))$ equivale a …	Diremos que el crecimiento asintótico de la función de coste es … en n	Y que la complejidad del algoritmo es … en n
$O(1)$	constante	constante
$O(\log n)$	logarítmico	logarítmica
$O(n)$	lineal	lineal
$O(n.\log n)$	enelogene	enelogene
$O(n^2)$	cuadrático	cuadrática
$O(n^3)$	cúbico	cúbica
$O(n^a)$	polinómico	polinómica (con $a \geq 1$)
$O(a^n)$	exponencial	exponencial (con $a > 1$)
$O(n!)$	factorial	factorial

Relaciones de inclusión entre los órdenes anteriores:

- ✓ Cadena de inclusiones:

$$O(1) \subset O(lg\, n) \subset O(n) \subset O(n.lg\, n) \subset O(n^2) \subset O(n^3) \subset \cdots \subset O(a^n) \subset O(n!)$$

- ✓ Los dos últimos órdenes corresponden a algoritmos «**intratables**»

Significado de la notación *O(n)*:

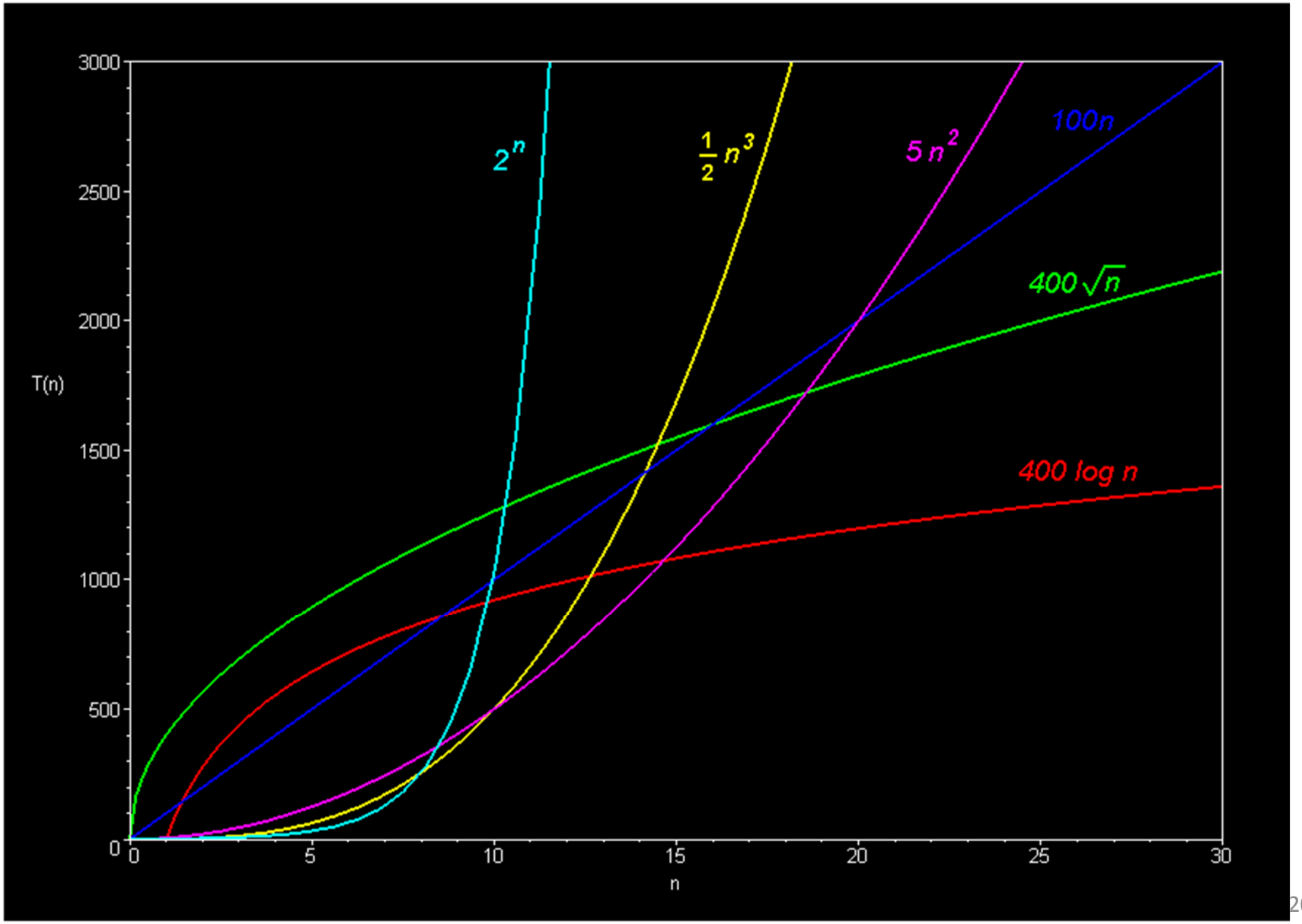
Sea $O(t_A(n))$ el orden de la función de coste $t_A(n)$ de un algoritmo A. Según que sea:

- $O(t_A(n)) = O(n)$ o bien
- $O(t_A(n)) = O(\log n)$ o bien
- $O(t_A(n)) = O(n \times \log n)$ o bien
- $O(t_A(n)) = O(n^2)$ o bien
- Etc., etc., etc.

Nos proporciona :

- Información muy precisa sobre la evolución asintótica del coste del algoritmo en función del parámetro **n**, es decir, del modo en que crece asintóticamente la función de coste
- Una aproximación asintótica de los valores que toma la función de coste es tanto más precisa cuanto mayor es el valor del parámetro **n** ($k.n$ o $k.\log n$ o $k.n^2$ o …)

Significado de la notación *O(n)*:



Aplicación a un caso

Sea $t_A(DIM)$ la función de coste de un algoritmo A que verifica:

$$O(t_A(DIM)) = O(DIM^2)$$

Entonces, la función de coste se puede aproximar asintóticamente (para **DIM** "suficientemente grande") a:

$t_A(DIM) \approx k.DIM^2$ donde **k** es una constante real

Medimos que, para **DIM = 1000**, el coste del algoritmo es de **0.25 ms**. Podemos ahora deducir **k**:

$$t_A(1000) = 0.25 \text{ ms} \approx k.1000^2 \rightarrow k = 0.25.10^{-6} \text{ ms}$$

Y, por lo tanto, la función de coste será:

$$t_A(DIM) \approx k.DIM^2 = 0.25.10^{-6}.DIM^2 \text{ ms}$$

Coste y principio de invariancia

Sea $t_A(n)$ la función de coste de un algoritmo que verifica:

$$O(t_A(n)) = O(f(n)) \rightarrow \lim_{n\to\infty} t_A(n) = k.f(n)$$

La constante **k** depende de la velocidad de la máquina donde se ejecuta el algoritmo. Supongamos que en un "Computador 1":

- ✓ **Computador 1** $\rightarrow \lim_{n\to\infty} t_A(n) = k_1.f(n)$

Si lo ejecutamos en un "Computador 2", 2 veces más rápido:

- ✓ **Computador 2** $\rightarrow \lim_{n\to\infty} t_A(n) = (k_1/2).f(n)$

Y si lo ejecutamos en un "Computador 3", 10 veces más rápido:

- ✓ **Computador 3** $\rightarrow \lim_{n\to\infty} t_A(n) = (k_1/10).f(n)$

Principio de invariancia: la función **f(n)** que caracteriza asintóticamente el coste **no varía**, sólo varía la constante multiplicativa **k**, que depende del entorno de ejecución

Reglas para caracterizar el orden de la función de coste

- ✓ Regla de las **constantes multiplicativas**:

  $O(k.f(n)) = O(f(n))$ para $k > 0$

- ✓ Regla de la **suma de funciones**:

  $O(f_1(n) + ... + f_K(n)) = O(Máx(f_1(n),..., f_K(n)))$

- ✓ Regla de la **sustitución de funciones**:

  $O(f(n)) = O(f'(n)) \rightarrow$

  $O(f(n)+g(n)) = O(f'(n)+g(n))$

  $O(f(n)) = O(f'(n)) \rightarrow$

  $O(f(n).g(n)) = O(f'(n).g(n))$

3. Ejemplos de caracterización asintótica de la eficiencia de un algoritmo

1. De la función **`factorial`**`(n)` (algoritmo iterativo de cálculo)
2. De la función **`sumarPares`**`(v,n)` (algoritmo de recorrido de un vector)
3. De la función **`buscar`**`(v,n,dato)` (algoritmo de búsqueda binaria en un vector ordenado)
4. De la función **`buscar`**`(v,n,dato)` (algoritmo de búsqueda secuencial en un vector)

Coste en memoria

```
/*
 *  Pre:  n >= 0
 *  Post: factorial(n) = (PROD alfa EN [1,n].alfa)
 */
int factorial (const int n) {
   int indice = 0, resultado = 1;
   while (indice != n) {
      ++indice;  resultado = indice * resultado;
   }
   return resultado;
}
```

$$mem_{factorial(n)}(n) = mem_{invocación} + 4 \times mem_{int}$$

$$O(mem_{factorial(n)}(n)) = O(mem_{invocación} + 4 \times mem_{int})$$
$$= O(1)$$

Coste en tiempo

```
/*
 *  Pre:  n >= 0
 *  Post: factorial(n) = (PROD alfa EN [1,n].alfa)
 */
int factorial (const int n) {
   int indice = 0, resultado = 1;                    //  a
   while (indice != n) {                             //  b
      ++indice;  resultado = indice * resultado;     //  c
   }
   return resultado;                                 //  d
}
```

$$t_{factorial(n)}(n) = (t_c + t_b) \times n + t_{invoc.} + t_a + t_b + t_d$$

$$O(t_{factorial(n)}(n)) = O((t_c + t_b) \times n + t_{invoc.} + t_a + t_b + t_d) = O((t_c + t_b) \times n) = O(n)$$

Coste en memoria

```
/*
 *  Pre:  n >= 0  AND  n <= #v
 *  Post: devuelve la suma de todos los datos
 *        con valor par almacenados en v[0,n-1]
 */
int sumarPares (const int v[], const int n) {
   int suma = 0;
   for (int i = 0; i < v; i++) {
      if (v[i] % 2 == 0) { suma = suma + v[i]; }
   }
   return suma;
}
```

$$mem_{sumaPares(v,n)}() = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$$

$$O(mem_{sumaPares(v,n)}()) = O(mem_{invocación} + mem_{ref} + 4 \times mem_{int}) = O(1)$$

Coste en tiempo

```
/*
 *  Pre:  n >= 0  AND  n <= #v
 *  Post: devuelve la suma de todos los datos
 *        con valor par almacenados en v[0,n-1]
 */
int sumarPares (const int v[], const int n) {
   int suma = 0;                                    //  a
   for (int i = 0; i < n; i++)                      //  b
      if (v[i] % 2 == 0) {                          //  c
         suma = suma + v[i];                        //  d
      }
      return suma;                                  // e
}
```

Caso mejor:
no hay ningún par

Coste en tiempo

```
/*
 *  Pre:  n >= 0  AND  n <= #v
 *  Post: devuelve la suma de todos los datos
 *        con valor par almacenados en v[0,n-1]
 */
int sumarPares (const int v[], const int n) {
   int suma = 0;                                    //  a
   for (int i = 0; i < n; i++)                      //  b
      if (v[i] % 2 == 0) {                          //  c
         suma = suma + v[i];                        //  d
      }
   return suma;                                     //  e
}
```

Caso peor:
no hay ningún impar

Coste en tiempo

CASO MEJOR: no hay ningún par en v[0,n-1]

$$O(t_{sumaPares(v,n)}(n)) = O((t_c + t_b) \times n + t_{invoc.} + t_a + t_b + t_e) = O((t_c + t_b) \times n) = O(n)$$

CASO PEOR: no hay ningún impar en v[0,n-1]

$$O(t_{sumaPares(v,n)}(n)) = O((t_c + t_d + t_b) \times n + t_{invoc.} + t_a + t_b + t_e) = O((t_c + t_d + t_b) \times n) = O(n)$$

Coste en memoria

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

$$mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 6 \times mem_{int}$$

$$O(mem_{buscar(v,n,dato)}(n)) = O(mem_{invocación}+mem_{ref}+6\times mem_{int}) = O(1)$$

Coste en tiempo

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

Coste en tiempo

El coste en tiempo del algoritmo, en cualquiera de los casos, es siempre:

$$t_{buscar(v,n,dato)}(n) = (t_c + t_d + t_e + t_b) \times \log_2 n + t_{invoc.} + t_a + t_b + t_g$$

$$O(t_{buscar(v,n,dato)}(n)) = O((t_c + t_d + t_e + t_b) \times \log_2 n + t_{invoc.} + t_a + t_b + t_g)$$

$$= O((t_c + t_d + t_e + t_b) \times \log_2 n)$$

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

Coste en memoria

```
/*
 *  Pre: n <= #v  AND  (EX alfa EN [0,n-1]. v[alfa] = dato)
 *  Post: v[buscar(v,n,dato)] = dato
 */
int buscar (const int v[], const int n, const int dato) {
   int indice = 0;
   while (v[indice] != dato) {
      indice = indice + 1;
   }
   return indice;
}
```

$$mem_{buscar(v,n,dato)}(n) = mem_{invocación} + mem_{ref} + 4 \times mem_{int}$$

$$O(mem_{buscar(v,n,dato)}(n)) = O(mem_{invocación}+mem_{ref}+4\times mem_{int}) = O(1)$$

Coste en tiempo

```
/*
 *  Pre: n <= #v  AND  (EX alfa EN [0,n-1]. v[alfa] = dato)
 *  Post: v[buscar(v,n,dato)] = dato
 */
int buscar (const int v, const int n, const int dato) {
   int indice = 0;                                  //  a
   while (v[indice] != dato) {                      //  b
      indice = indice + 1;                          //  c
   }
   return indice;                                   //  d
}
```

Caso mejor: v[0]=dato

Coste en tiempo

```
/*
 *  Pre: n <= #v  AND  (EX alfa EN [0,n-1].v[alfa] = dato)
 *  Post: v[buscar(v,n,dato)] = dato
 */
int buscar (const int v[], const int n, const int dato) {
   int indice = 0;                                  // a
   while (v[indice] != dato) {                      // b
      indice = indice + 1;                          // c
   }
   return indice;                                   // d
}
```

Caso peor:
dato sólo coincide con v[n-1]

CASO MEJOR (v[0] = dato):

Coste en tiempo

$$t_{buscar(v,n,dato)}(n) = t_{invoc.} + t_a + t_b + t_d$$

$$O(t_{buscar(v,n,dato)})(n) = O(t_{invoc.} + t_a + t_b + t_d) = O(1)$$

CASO PEOR ((∀α∈[0,n-2]. v[α]≠dato) ∧ v[n-1] = dato):

$$t_{buscar(v,n,dato)}(n) = (t_c + t_b) \times n + t_{invoc.} + t_a - t_c + t_d$$

$$O(t_{buscar(v,n,dato)})(n) = O((t_c + t_b) \times n + t_{invoc.} + t_a - t_c + t_d) = O((t_c + t_b) \times n) = O(n)$$