Programación 2
Lección 7: Coste de un algoritmo

Índice

1. Eficiencia de un algoritmo

- **Eficiencia**: uso razonable de los recursos
- **Recursos**:

  memoria: para almacenar sus datos

  tiempo: invertido en su ejecución
- Analizaremos el coste de nuestros algoritmos:
  - la cantidad de memoria que precisan para almacenar sus datos y
  - el tiempo invertido en su ejecución

2. Coste en memoria y en tiempo

Coste en **memoria**

```
//Pre: uno = X ∧ otro = Y
//Post: uno = Y ∧ otro = X
void permutar(T &uno, T &otro) {
   T auxiliar = uno;
   uno = otro;
   otro = auxiliar;
}
```

2. Coste en memoria y en tiempo

Coste en **memoria**

```
//Pre: uno = X ∧ otro = Y
//Post: uno = Y ∧ otro = X
void permutar(T &uno, T &otro) {
   T auxiliar = uno;
   uno = otro;
   otro = auxiliar;
}
```

$$mem_{permutar(uno,otro)} = mem_{invoc} + 2 \cdot mem_{ref} + mem_T$$

El coste en memoria no depende de ningún parámetro: es constante

2. Coste en memoria y en tiempo

Coste en **tiempo**

```
//Pre: uno = X ∧ otro = Y
//Post: uno = Y ∧ otro = X
void permutar(T &uno, T &otro) {
   T auxiliar = uno;      // a
   uno = otro;            // b
   otro = auxiliar;       // c
}
```

2. Coste en memoria y en tiempo

Coste en **tiempo**

```
//Pre: uno = X ∧ otro = Y
//Post: uno = Y ∧ otro = X
void permutar(T &uno, T &otro) {
   T auxiliar = uno;      // a
   uno = otro;            // b
   otro = auxiliar;       // c
}
```

$$t_{permutar(uno,otro)} = t_{invoc} + t_a + t_b + t_c$$

El coste en tiempo no depende de ningún parámetro: es constante

Análisis de coste de $permutar(uno, otro)$

- **Coste en memoria:** No cabe distinguir casos particulares, ya que hay una caracterización general del coste que es constante

$$mem_{permutar(uno,otro)} = mem_{invoc} + 2 \cdot mem_{ref} + mem_T$$

- **Coste en tiempo:** No cabe distinguir casos particulares, ya que hay una caracterización general del coste que es constante

$$t_{permutar(uno,otro)} = t_{invoc} + t_a + t_b + t_c$$

Análisis de coste de $factorial(n)$

Coste en **memoria**

```
//Pre: 0 ≤ n
//Post: factorial(n) = ∏_{α=1}^{n} .α
int factorial(const int n) {
   int indice = 0, resultado = 1;
   while (indice != n) {
      indice = indice + 1;
      resultado = indice * resultado;
   }
   return resultado;
}
```

$$mem_{factorial(n)}(n) = mem_{invoc} + 4 \cdot mem_{int}$$

Análisis de coste de $factorial(n)$

Coste en **tiempo**

```
//Pre: 0 ≤ n
//Post: factorial(n) = ∏_{α=1}^{n} .α
int factorial(const int n) {
   int indice = 0, resultado = 1;          // a
   while (indice != n) {                   // b
      indice = indice + 1;                 // c
      resultado = indice * resultado;      // d
   }
   return resultado;                       // e
}
```

$$t_{factorial(n)}(n) = t_{invoc} + t_a + t_b + \sum_{\alpha=1}^{n} (t_c + t_d + t_b) + t_e$$
$$= (t_c + t_d + t_b) \cdot n + t_{invoc} + t_a + t_b + t_e$$

Análisis de coste de $factorial(n)$

- **Coste en memoria:** No cabe distinguir casos particulares, ya que hay una caracterización general del coste que es constante

$$mem_{factorial(n)}(n) = mem_{invoc} + 4 \cdot mem_{int}$$

- **Coste en tiempo:** No cabe distinguir casos particulares, ya que hay una caracterización general del coste que depende **linealmente** del valor del parámetro n

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

Análisis de coste de $sumarPares(v, n)$

Coste en **memoria**

```
//Pre: 0 ≤ n ≤ #v
//Post: sumarPares(v,n) 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 < n; i++) {
      if (v[i] % 2 == 0) {
         suma = suma + v[i];
      }
   }
   return suma;
}
```

$$mem_{sumarPares(v,n)}(n) = mem_{invoc} + mem_{ref} + 4 \cdot mem_{int}$$

Análisis de coste de $sumarPares(v, n)$

Coste en **tiempo**: depende de los datos

```
//Pre: 0 ≤ n ≤ #v
//Post: sumarPares(v,n) 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
}
```

$$t_{sumarPares(v,n)}(n) = t_{invoc} + t_a + t_b + \sum_{\alpha=1}^{n} (t_c + \text{¿}t_d\text{?} + t_b) + t_e$$

Análisis de coste de $sumarPares(v, n)$

Coste en **tiempo caso mejor**

```
//Pre: 0 ≤ n ≤ #v
//Post: sumarPares(v,n) 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
}
```

$$t_{sumarPares(v,n)}(n) = t_{invoc} + t_a + t_b + (t_c + t_b) \cdot n + t_e$$

Análisis de coste de $sumarPares(v, n)$

Coste en **tiempo caso peor**

```
//Pre: 0 ≤ n ≤ #v
//Post: sumarPares(v,n) 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
}
```

$$t_{sumarPares(v,n)}(n) = t_{invoc} + t_a + t_b + (t_c + t_d + t_b) \cdot n + t_e$$

Análisis de coste de $sumarPares(v, n)$

Caso mejor: en ninguna iteración se ejecuta "d"

$$t_{sumarPares(v,n)}(n) = (t_c + t_b) \cdot n + t_{invoc} + t_a + t_b + t_e$$

Caso peor: en todas las iteraciones se ejecuta "d"

$$t_{sumarPares(v,n)}(n) = (t_c + t_d + t_b) \cdot n + t_{invoc} + t_a + t_b + t_e$$

Caso general: en algunas iteraciones se ejecuta "d", en otras no. El coste será un valor entre el peor y el mejor caso.

Análisis de coste de $buscarPar(v, n)$

Coste en **memoria**

```
//Pre: 0 ≤ n ≤ #v
//Post: buscarPar(v,n) devuelve un número par almacenado en
//      v[0,n-1] y, si no hay ninguno, devuelve -1
int buscarPar(const int v[], const int n) {
   int i = 0;
   bool encontrado = false;
   while (!encontrado && i < n) {
      if (v[i] % 2 == 0) { encontrado = true; }
      else { i = i + 1; }
   }
   if (encontrado) { return v[i]; }
   else  { return -1; }
}
```

$$mem_{buscarPar}(v, n) = mem_{invoc} + mem_{ref} + 3 \cdot mem_{int} + mem_{bool}$$

Análisis de coste de $buscarPar(v, n)$

Coste en **tiempo**: **caso peor**

```
//Pre: 0 ≤ n ≤ #v
//Post: buscarPar(v,n) devuelve un número par almacenado en
//      v[0,n-1] y, si no hay ninguno, devuelve -1
int buscarPar(const int v[], const int n) {
   int i = 0;                                  // a
   bool encontrado = false;                    // b
   while (!encontrado && i < n) {              // c
      if (v[i] % 2 == 0)                       // d
         { encontrado = true; }                // e
      else
         { i = i + 1;}                         // f
   }
   if (encontrado)                             // g
      { return v[i]; }                         // h
   else
      { return -1;}                            // i
}
```

Análisis de coste de $buscarPar(v, n)$

Caso peor: los datos $v[0]$ a $v[n-1]$ son impares.

$$\begin{aligned} t_{buscarPar(v,n)}(n) &= t_{invoc} + t_a + t_b + t_c + \sum_{\alpha=1}^{n} (t_d + t_f + t_c) + t_g + t_i \\ &= (t_d + t_f + t_c) \cdot n + t_{invoc} + t_a + t_b + t_c + t_g + t_i \end{aligned}$$

Análisis de coste de $buscarPar(v, n)$

Coste en **tiempo**: **caso mejor**

```
//Pre: 0 ≤ n ≤ #v
//Post: buscarPar(v,n) devuelve un número par almacenado en
//      v[0,n-1] y, si no hay ninguno, devuelve -1
int buscarPar(const int v[], const int n) {
   int i = 0;                                  // a
   bool encontrado = false;                    // b
   while (!encontrado && i < n) {              // c
      if (v[i] % 2 == 0)                       // d
         { encontrado = true; }                // e
      else
         { i = i + 1;}                         // f
   }
   if (encontrado)                             // g
      { return v[i]; }                         // h
   else
      { return -1;}                            // i
}
```

Análisis de coste de $buscarPar(v, n)$

Caso mejor: $v[0]$ almacena un número par.

$$t_{buscarPar(v,n)}(n) = t_{invoc} + t_a + t_b + t_c + t_d + t_e + t_c + t_g + t_h$$

Análisis de coste de $buscarPar(v, n)$

Caso mejor: $v[0]$ almacena un número par.

$$t_{buscarPar(v,n)}(n) = t_{invoc} + t_a + t_b + t_c + t_d + t_e + t_c + t_g + t_h$$

Caso peor: los datos $v[0]$ a $v[n-1]$ son impares.

$$\begin{aligned} t_{buscarPar(v,n)}(n) &= t_{invoc} + t_a + t_b + t_c + \sum_{\alpha=1}^{n} (t_d + t_f + t_c) + t_g + t_i \\ &= (t_d + t_f + t_c) \cdot n + t_{invoc} + t_a + t_b + t_c + t_g + t_i \end{aligned}$$

Casos restantes: hay un dato par en $v[i]$, con $i > 0$. Coste intermedio entre el caso mejor (coste mínimo) y el caso peor (coste máximo).

3. Cálculo del coste en tiempo

3. Cálculo del coste en tiempo

Coste de instrucciones simples

- $expresión;$
- $variable = expresión;$

- $t_{eval\_expresión} = t_{calcula\_operandos} + t_{ejecuta\_operaciones}$
- $t_{asignación} = t_{eval\_expresión} + t_{operación\_asignación}$

3. Cálculo del coste en tiempo

Coste de instrucciones simples (cont.)

- $nombreFuncion(lista\_de\_argumentos);$
- $return\ expresión;$
- $;$

- $t_{llamada\_función} = t_{eval\_argumentos} + t_{ejecuta\_código}$
- $t_{return} = t_{eval\_expresión}$
- $t_{intrucción\_nula} = K_{intrucción\_nula}$

Coste de un bloque de instrucciones

```
{
    instr_A;          // A
    instr_B;          // B
    . . .             // ...
    instr_K;          // K
}
```

$$t_{bloque} = t_A + t_B + ... + t_K$$

Coste de la instrucción condicional

```
if (c_A)
    { bloque_A }
else if (c_B)
    { bloque_B }
   . . .
else if (c_K)
    { bloque_K }
else
    { bloque_else }
```

$$
\begin{aligned}
& c_A \rightarrow t_{cond} = t_{c_A} + t_A \\
& \neg c_A \wedge c_B \rightarrow t_{cond} = t_{c_A} + t_{c_B} + t_B \\
& \ldots \\
& \neg c_A \wedge \neg c_B \wedge \ldots \wedge \neg c_{K-1} \wedge c_K \rightarrow t_{cond} = t_{c_A} + t_{c_B} + \ldots + t_{c_K} + t_K \\
& \neg c_A \wedge \neg c_B \wedge \ldots \wedge \neg c_K \rightarrow t_{cond} = t_{c_A} + t_{c_B} + \ldots + t_{c_K} + t_{else}
\end{aligned}
$$

Coste de un bucle $while$

```
while (cond) {
   bloque
}
```

$$t_{while} = t_{Cond(0)} + \sum_{\alpha=1}^{k} (t_{bloque(\alpha)} + t_{cond(\alpha)})$$

- El número de iteraciones del bucle, k, depende de la satisfacción de la condición del bucle
- El coste de evaluación de la condición del bucle y de ejecución del bloque a iterar pueden variar en cada iteración

Coste de un bucle for

```
for (inicialización; condición; incremento) {
   bloque
}
```

$$t_{for} = t_{gestión(0)} + \sum_{\alpha=1}^{k} (t_{bloque(\alpha)} + t_{gestión(\alpha)})$$

- El número de iteraciones del bucle, k, depende de la satisfacción de la condición del bucle
- El coste de gestión del bucle y de ejecución del bloque a iterar pueden variar en cada iteración.

Aplicación a un algoritmo de búsqueda binaria

```
//Pre:  n ≤ #v ∧  ∃α ∈ [0, n − 1].(v[α] = dato)
//              ∧  ∀α ∈ [0, n − 2].(v[α] ≤ v[α + 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;
}
```

Coste en memoria

```
//Pre:  n ≤ #v ∧  ∃α ∈ [0, n − 1].(v[α] = dato)
//              ∧  ∀α ∈ [0, n − 2].(v[α] ≤ v[α + 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;
}
```

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

Coste en tiempo

```
//Pre: n ≤ #v ∧  ∃α ∈ [0, n − 1].(v[α] = dato)
//            ∧  ∀α ∈ [0, n − 2].(v[α] ≤ v[α + 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
}
```

$$t_{buscar(v,n,dato)}(n) = t_{invoc} + t_a + t_{while} + t_g$$

Coste en tiempo

El número de iteraciones es aproximadamente igual a $log_2(n)$, por lo tanto:

$$t_{buscar(v,n,dato)}(n) = t_{invoc} + t_a + t_{while} + t_g$$

$$= t_{invoc} + t_a + t_b + \sum_{\alpha=1}^{log_2 n} (t_c + t_d + (t_e \mid t_f) + t_b) + t_g$$

$$= (t_c + t_d + t_f + t_b) \cdot log_2 n + t_{invoc} + t_a + t_b + t_g$$

- Ya que podemos asumir que $t_e \approx t_f$
- El coste calculado es para cualquier caso (caso general) ya que no depende de los datos almacenados en v[0,n-1]

Análisis de coste de $buscar(v, n, dato)$

Coste en memoria: No cabe distinguir casos particulares, ya que hay una caracterización general del coste que es constante

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

Coste en tiempo: No cabe distinguir casos particulares, hay un caso general cuyo coste es logarítmico en el valor del parámetro n

$$t_{buscarPar}(v, n) = (t_c + t_d + t_f + t_b) \cdot log_2 n + t_{invoc} + t_a + t_b + t_g$$

Aplicación a un algoritmo de búsqueda secuencial

```
//Pre: n ≤ #v ∧ ∃α ∈ [0, n − 1].(v[α] = 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;
}
```

Coste en memoria

```
//Pre: n ≤ #v ∧ ∃α ∈ [0, n − 1].(v[α] = 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;
}
```

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

Coste en tiempo

```
//Pre: n ≤ #v ∧ ∃α ∈ [0, n − 1].(v[α] = 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
}
```

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

Coste en tiempo: caso mejor

```
//Pre: n ≤ #v ∧ ∃α ∈ [0, n − 1].(v[α] = 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
}
```

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

Coste en tiempo: caso peor

```
//Pre: n ≤ #v ∧ ∃α ∈ [0, n − 1].(v[α] = 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
}
```

$$t_{buscar(v,n,dato)}(n) = t_{invoc} + t_a + t_{while} + t_d$$
$$= t_{invoc} + t_a + t_b + \sum_{\alpha=1}^{n-1} (t_c + t_b) + t_d$$

Análisis de coste de $buscar(v, n, dato)$

Caso mejor: $v[0] = dato$

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

Caso peor: $\forall \alpha \in [0, n-2].(v[\alpha] \neq dato) \wedge v[n-1] = dato$

$$\begin{aligned} t_{buscar(v,n,dato)}(n) &= t_{invoc} + t_a + t_{while} + t_d \\ &= t_{invoc} + t_a + t_b + \sum_{\alpha=1}^{n-1} (t_c + t_b) + t_d \\ &= t_{invoc} + t_a + t_b + (t_c + t_b) \cdot (n-1) + t_d \\ &= (t_c + t_b) \cdot n + t_{invoc} + t_a + t_b - t_c - t_b + t_d \\ &= (t_c + t_b) \cdot n + t_{invoc} + t_a - t_c + t_d \end{aligned}$$

Análisis de coste de $buscar(v, n, dato)$

Coste en memoria: No cabe distinguir casos particulares, ya que hay una caracterización general del coste que es constante

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

Coste en tiempo: Cabe distinguir casos particulares, en función a la primera aparición de $dato$ en v[0,n-1]

- **Caso mejor:**

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

- **Caso peor:**

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