Programación 2

Análisis del coste de algoritmos: deducción de su función de coste

Problemas 07

Problema 1. Deducir el coste en tiempo y memoria

Deducir el coste, en tiempo y memoria, del siguiente código C++, en función del valor del dato variable **n**. Denominaremos t_a, t_b y t_c los costes, en tiempo, de ejecutar cada línea de este código

```
double x = 0;                                  // a
for (int i = 1; i <= n; i++)  {                // b
   x = calcular(x,i);                          // c
}
```

Y se sabe que:

$$t_{calcular(y,j)}(j) = k_1 . j \quad \text{y que} \quad mem_{calcular(y,j)}(j) = k_2$$

donde $\mathbf{k_1}$ y $\mathbf{k_2}$ son dos constantes dependientes del entorno en el que se ejecuta el código

```
double x = 0;                                   // a
for (int i = 1; i <= n; i++)  {                 // b
   x = calcular(x,i);                           // c
}
```

Iteración α	i	Observaciones	Coste iteración α-ésima
1	1		$t_{c'} + k_1.1 + t_b$
2	2		$t_{c'} + k_1.2 + t_b$
3	3		$t_{c'} + k_1.3 + t_b$
… α …	… α …		… $t_{c'} + k_1.\alpha + t_b$ …
k	n	k = n	$t_{c'} + k_1.n + t_b$

```
double x = 0;                              // a
for (int i = 1; i <= n; i++)  {            // b
   x = calcular(x,i);                      // c
}
```

Donde:

$$t_{calcular(y,j)}(j) = k_1.j \quad \Rightarrow \quad t_{calcular(x,i)}(i) = k_1.i$$

Calculemos el coste en función del valor de **n**:

$$t(n) = t_a + t_b + (\Sigma\alpha\in[1,n].t_c + t_b)$$

$$t(n) = t_a + t_b + (\Sigma\alpha\in[1,n].t_{c'} + k_1.i + t_b)$$

$$t(n) = t_a + t_b + (\Sigma\alpha\in[1,n].t_{c'} + k_1.\alpha + t_b)$$ [ya que: **i=α**]

$$t(n) = t_a + t_b + (t_{c'} + t_b)n + k_1(\Sigma\alpha\in[1,n].\alpha)$$

$$t(n) = t_a + t_b + (t_{c'} + t_b)n + k_1/2(1+n).n$$

$$\boxed{t(n) = (k_1/2).n^2 + (t_{c'} + t_b + k_1/2).n + t_a + t_b}$$

Vamos ahora a caracterizar asintóticamente el coste, <u>en uso de memoria</u>, del código C++ considerado anteriormente, en función del valor del dato variable **n**, sabiendo que:

$$\mathtt{mem}_{\mathtt{calcular(y,j)}}\mathtt{(j)} = \mathtt{k}_2$$

```cpp
double x = 0;
for (int i = 1; i <= n; i++)  {
   x = calcular(x,i);
}
```

Caracterización asintótica del coste, <u>en uso de memoria</u>, del código C++ considerado anteriormente, en función del valor del valor de la variable **n** , sabiendo que $\texttt{mem}_{\texttt{calcular(y,j)}}\texttt{(j) = k}_2$

```
double x = 0;
for (int i = 1; i <= n; i++)  {
   x = calcular(x,i);
}
```

Hace uso de las variables **x** de tipo **double** (su coste es $\texttt{mem}_{\texttt{double}}$) e **i** de tipo **int** (su coste es $\texttt{mem}_{\texttt{int}}$) y de la memoria constante, k_2, necesaria para ejecutar la invocación `calcular(x,i)`:

$$\texttt{memoria(n) = mem}_{\texttt{double}} + \texttt{mem}_{\texttt{int}} + \texttt{mem}_{\texttt{calcular(x,i)}}$$

$$\texttt{memoria(n) = mem}_{\texttt{double}} + \texttt{mem}_{\texttt{int}} + \texttt{k}_2$$

Problema 2. Análisis asintótico del coste en tiempo

Deducir la función de coste, en tiempo, de ejecutar la función **esPrimo**(n).

```
// Pre: cierto
// Post: esPrimo(n) = (n >= 2) AND
//                    (PT alfa EN [2,n-1] n % alfa > 0)
bool esPrimo (const int n) {
   if (n == 2) { return true; }
   else if (n < 2 || n % 2 == 0) { return false; }
   else {
      int divisor = 3;
      bool loParece = true;
      while (loParece && divisor * divisor <= n) {
         loParece = n % divisor > 0;  divisor = divisor + 2;
      }
      return loParece;
   }
}
```

```
// Pre: cierto
// Post: esPrimo(n) = (n >= 2) AND
//                     (PT alfa EN [2,n-1] n % alfa > 0)
bool esPrimo (const int n) {
   if (n == 2) {                                              // a
      return true;                                            // b
   }
   else if (n < 2 || n % 2 == 0) {                            // c
      return false;                                           // d
   }
   else {
      int divisor = 3;                                        // e
      bool loParece = true;                                   // f
      while (loParece && divisor * divisor <= n) {            // g
         loParece = n % divisor > 0;  divisor = divisor + 2;  // h
      }
      return loParece;                                        // i
   }
}
```

```
int divisor = 3;                                         // e
bool loParece = true;                                    // f
while (loParece && divisor * divisor <= n) {             // g
   loParece = n % divisor > 0;  divisor = divisor + 2;   // h
}
```

Iteración α	divisor	Observaciones	Coste iteración α
1	3	3^2 <= n	$t_h + t_g$
2	5	5^2 <= n	$t_h + t_g$
3	7	7^2 <= n	$t_h + t_g$
… α …	… $2\alpha+1$ …	… $(2\alpha+1)^2$ <= n …	… $t_h + t_g$ …
k	2k+1	$(2k+1)^2$ <= n $(2k+3)^2$ > n $\mathbf{k} \approx \frac{\sqrt{n}}{2}$	$t_h + t_g$

Análisis asintótico del coste: para **n** positivo o negativo, con **|n|** suficientemente grande.

Casos mejores: cuando **n** es negativo o par:

$$t(n) = t_a + t_c + t_d$$

Casos peores: cuando **n** es primo (no tiene mas divisores que **1** y **n**):

$$t(n) = t_a + t_c + t_e + t_f + t_g + (\Sigma\alpha\in[1,k].\ t_h + t_g) + t_i$$

Y para **n** suficientemente grande, el número de iteraciones **k** $\approx \frac{\sqrt{n}}{2}$

$$t(n) = \frac{1}{2}(t_h + t_g)\sqrt{n} + t_a + t_c + t_e + t_f + t_g + t_i$$

Problema 3. Determinar el coste en tiempo y memoria

```
// Pre: v = Vo AND  n > 0  AND  n <= #v
// Post: esPermutación(v,Vo,0,n-1)  AND  ordenado(v,0,n-1)
template <typename T>
void seleccion (T v[], const int n) {
   // Ordenación del vector v[0,n-1] aplicando el método de selección
   for (int i = 0; i != n - 1; ++i) {
      int iMenor = i, j = i;
      while (j != n - 1) {
          j = j + 1;
          if (v[j] < v[iMenor]) {
             iMenor = j;
          }
      }
      T dato = v[i];
      v[i] = v[iMenor]; v[iMenor] = dato;
   }
}
```

Véase esta función en la lección 16

Sean d1 y d2 dos datos de tipo T. Se sabe que el coste:

$$t_{d1 < d2} = k_1$$
$$t_{d1 = d2} = k_2$$

Donde:

```
esPermutación(v1,v2,desde,hasta) =
    (PT alfa EN [desde,hasta].
           (Núm beta EN [desde,hasta]. v1[beta] = v1[alfa])
            = (Núm beta EN [desde,hasta]. v2[beta] = v1[alfa]) )

ordenado(v,desde,hasta) =
    (PT alfa EN [desde,hasta-1]. v[alfa] <= v[alfa+1])
```

```
// Pre: v = Vo  AND  n > 0  AND  n <= #v
// Post: esPermutación(v,Vo,0,n-1) AND ordenado(v,0,n-1)
template <typename T>
void seleccion (T v[], const int n) {
   // Ordenación del vector v[0,n-1] aplicando el método de selección
   for (int i = 0; i != n - 1; ++i) {                        // a
      int iMenor = i, j = i;                                 // b
      while (j != n - 1) {                                   // c
          j = j + 1;                                         // d
          if (v[j] < v[iMenor]) {                            // e
             iMenor = j;                                     // f
          }
      }
      T dato = v[i];                                         // g
      v[i] = v[iMenor]; v[iMenor] = dato;                    // h
   }
}
```

Y se sabe que: $t_{d1 < d2} = k_1$ y $t_{d1 = d2} = k_2$

```
// Pre: v = Vo  AND  n > 0  AND  n <= #v
// Post: esPermutación(v,Vo,0,n-1) AND ordenado(v,0,n-1)
template <typename T>
void seleccion (T v[], const int n) {
   // Ordenación del vector v[0,n-1] aplicando el método de selección
   for (int i = 0; i != n - 1; ++i) {                       // a
      int iMenor = i, j = i;                                // b
      while (j != n - 1) {                                  // c
         j = j + 1;                                         // d
         if (v[j] < v[iMenor]) {                            // e
            iMenor = j;                                     // f
         }
      }
      T dato = v[i];                                        // g
      v[i] = v[iMenor]; v[iMenor] = dato;                   // h
   }
}
```

CASO PEOR (se ejecuta todo el código)

Y se sabe que: $t_{d1 < d2} = k_1$ y $t_{d1 = d2} = k_2$

```
for (int i = 0; i != n - 1; ++i) {                   // a
    int iMenor = i, j = i;                           // b
    while (j != n - 1) { … }                         // bucle while
    T dato = v[i];                                   // g
    v[i] = v[iMenor]; v[iMenor] = dato;              // h
}
```

Iteración α	i	observaciones	Coste iteración α
1	0		$t_a + t_b + t_c + t_{while(1)} + t_g + t_h$
2	1		$t_a + t_b + t_c + t_{while(2)} + t_g + t_h$
3	2		$t_a + t_b + t_c + t_{while(3)} + t_g + t_h$
… α …	… α-1 …	i = α - 1	… $t_a + t_b + t_c + t_{while(\alpha)} + t_g + t_h$ …
k	n-2	k = n - 1	$t_a + t_b + t_c + t_{while(n-1)} + t_g + t_h$

```
j = i;
while (j != n - 1) {                          // c
     j = j + 1;                               // d
     if (v[j] < v[iMenor]) {                  // e
        iMenor = j;                           // f
     }
}
```

Iteración β	j	observaciones	Coste iteración β
1	i		$t_d + t_e + t_f + t_c$
2	i + 1		$t_d + t_e + t_f + t_c$
3	i + 2		$t_d + t_e + t_f + t_c$
… β …	… $i + \beta - 1$ …	$j = i + \beta - 1$	… $t_d + t_e + t_f + t_c$ …
k'	n - 2	$k' = n - i - 1$	$t_d + t_e + t_f + t_c$

Función de coste de la función:

$$t(n) = (\Sigma\alpha\in[1,n-1].t_a+t_b+t_c+t_{while}+t_g+t_h) \quad \text{siendo: } \alpha = i + 1$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (\Sigma\alpha\in[1,n-1].t_{while})$$

Caso peor. Cuando siempre se satisface que `(v[j] < v[iMenor])`:

$$t_{while} = (\Sigma\beta\in[i,n-2].t_d+t_e+t_f+t_c) = (t_d+t_e+t_f+t_c)(n-1-i)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_f+t_c)(\Sigma\alpha\in[1,n-1].n-1-i)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_f+t_c)(\Sigma\alpha\in[1,n-1].n-\alpha)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_f+t_c)\tfrac{1}{2}n(n-1)$$

$$\boxed{t(n) = \tfrac{1}{2}(t_d+t_e+t_f+t_c)n(n-1) + (t_a+t_b+t_c+t_g+t_h)(n-1)}$$

```
// Pre: v = Vo  AND  n > 0  AND  n <= #v
//  Post: esPermutación(v,Vo,0,n-1) AND ordenado(v,0,n-1)
template <typename T>
void seleccion (T v[], const int n) {
   // Ordenación del vector v[0,n-1] aplicando el método de selección
   for (int i = 0; i != n - 1; ++i) {                              // a
      int iMenor = i, j = i;                                       // b
      while (j != n - 1) {                                         // c
          j = j + 1;                                               // d
          if (v[j] < v[iMenor]) {                                  // e
             iMenor = j;                                           // f
          }
      }
      T dato = v[i];                                               // g
      v[i] = v[iMenor]; v[iMenor] = dato;                          // h
   }
}
```

CASO MEJOR (nunca se ejecuta el código de f)

Y se sabe que: $t_{d1 < d2} = k_1$ y $t_{d1 = d2} = k_2$

```
for (int i = 0; i != n - 1; ++i) {                // a
    int iMenor = i, j = i;                        // b
    while (j != n - 1) { … }                      // bucle while
    T dato = v[i];                                // g
    v[i] = v[iMenor]; v[iMenor] = dato;           // h
}
```

Iteración α	i	Observaciones	Coste iteración α
1	0		$t_a + t_b + t_c + t_{while(1)} + t_g + t_h$
2	1		$t_a + t_b + t_c + t_{while(2)} + t_g + t_h$
3	2		$t_a + t_b + t_c + t_{while(3)} + t_g + t_h$
… α …	… $\alpha - 1$ …	$i = \alpha - 1$	… $t_a + t_b + t_c + t_{while(\alpha)} + t_g + t_h$ …
k	n - 2	k = n - 1	$t_a + t_b + t_c + t_{while(n-1)} + t_g + t_h$

```
j = i;
while (j != n - 1) {                          // c
     j = j + 1;                               // d
     if (v[j] < v[iMenor]) {                  // e
        iMenor = j;                           // f
     }
}
```

Iteración β	j	observaciones	Coste iteración α
1	i		$t_d + t_e + t_c$
2	i + 1		$t_d + t_e + t_c$
3	i + 2		$t_d + t_e + t_c$
… β …	… $i + \beta - 1$ …	$j = i + \beta - 1$	$t_d + t_e + t_c$
k'	n - 2	k' = n - i - 1	$t_d + t_e + t_c$

Función de coste de la función:

$$t(n) = (\Sigma\alpha\in[1,n-1].t_a+t_b+t_c+t_{while}+t_g+t_h) \quad \text{siendo: } \alpha = i + 1$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (\Sigma\alpha\in[1,n-1].t_{while})$$

Caso mejor. Cuando nunca se satisface `(v[j] < v[iMenor])`:

$$t_{while} = (\Sigma\beta\in[i,n-2].t_d+t_e+t_c) = (t_d+t_e+t_c)(n-1-i)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_c)(\Sigma\alpha\in[1,n-1].n-1-i)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_c)(\Sigma\alpha\in[1,n-1].n-\alpha)$$

$$t(n) = (t_a+t_b+t_c+t_g+t_h)(n-1) + (t_d+t_e+t_c)\,\tfrac{1}{2}n(n-1)$$

$$\boxed{t(n) = \tfrac{1}{2}(t_d+t_e+t_c)\,n(n-1) + (t_a+t_b+t_c+t_g+t_h)(n-1)}$$

Caracterización del coste, en uso de memoria

```
// Pre: v = Vo  AND  n > 0  AND  n <= #v
// Post: esPermutación(v,Vo,0,n-1) AND ordenado(v,0,n-1)
template <typename T>
void seleccion (T v[], const int n) {  // Ordenación por selección
   for (int i = 0; i != n - 1; ++i) {
      int iMenor = i, j = i;
      while (j != n-1) {
          j = j + 1;
          if (v[j] < v[iMenor]) { iMenor = j; }
      }
      T dato = v[i]; v[i] = v[iMenor]; v[iMenor] = dato;
   }
}
```

Hace uso de las variables **v** (referencia cuyo coste es m_{ref}), **n** , **i** , **iMenor** y **j** de tipo **int** (su coste es m_{int}) y **dato** (su coste es m_T) :

$$mem(n) = mem_{invocación} + mem_{ref} + 4 \times mem_{int} + mem_T$$