

5.1.(Clase) Obtener el tiempo de ejecución $t(n)$, $O(t(n))$, $\Omega(t(n))$ y $o(t(n))$ para el siguiente algoritmo de multiplicación de matrices:

```

for i= 1 to n
  for j= 1 to n
    suma= 0
    for k= 1 to n
      suma= suma + a[i, k]·b[k, j]
    endfor
    c[i, j]= suma
  endfor
endfor

```

¿Cuál es el significado de $t(n)$ en este caso? ¿Dependen $O(t(n))$ y $\Omega(t(n))$ de este significado? ¿Y $o(t(n))$? Justifica las respuestas.

5.2.(TG 7.1) Obtener O y Ω , y Θ del tiempo promedio para el algoritmo de búsqueda binaria.

```

i= 1
j= n
repeat
  m= (i+j) div 2
  if a[m] > x
    j= m-1
  else
    i= m+1
  endif
until i>j or a[m]=x

```

Dadas las características del anterior algoritmo, justifica por qué no es correcto (hablando matemáticamente) definir el tiempo de ejecución $t(n)$ para todos los casos como una función de N en \mathbb{R}^+ . En consecuencia, ¿cómo se debe definir el tiempo para tener funciones de N en \mathbb{R}^+ ?

5.3.Demostrar que siendo $a, b, c \in \mathbb{R}^+$ y $d, n_0 \in \mathbb{N}$, con $d > 0$ y:

$$f(n) = \begin{cases} c & \text{Si } n < n_0 \\ a \cdot f(n-d) + b & \text{En otro caso} \end{cases}$$

Se cumple que:

$$a < 1 \Rightarrow f \in O(1)$$

$$a = 1 \Rightarrow f \in O(n)$$

$$a > 1 \Rightarrow f \in O(a^{n/d})$$

5.4.(EX) Decir si son ciertas o falsas las siguientes afirmaciones, razonando la respuesta:

- $\Theta(\log_2 n) = \Theta(\log_{10} n)$
- $O(2^n) = O(2^{n+3})$
- $O(n^{n+2}) = O(n^{n+3})$
- $O(m \cdot n) \subseteq O(2^n)$

5.5.(EX, Clase) El tiempo de ejecución de un determinado programa se puede expresar con la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} 2n & \text{Si } n \leq 10 \\ 2 \cdot t(\lfloor n/2 \rfloor) + 3 \cdot t(\lfloor n/4 \rfloor) + 2n + 1 & \text{En otro caso} \end{cases}$$

- Estudia el tiempo de ejecución del algoritmo, para los valores de n que sean potencia de 2. A partir del tiempo $t(n)$, expresa la complejidad usando las notaciones O , Ω ó Θ . (No hace falta calcular el valor de las constantes de $t(n)$, se supondrán todas positivas.)
- Muestra las condiciones iniciales que deberían aplicarse para calcular las constantes en la fórmula de $t(n)$. Sólo es necesario indicar los valores de n que son las condiciones iniciales y escribir alguna de las ecuaciones que surgen de esas condiciones. No es necesario resolverlas.
- La afirmación $t(n) \in \Omega(\log n)$ ¿es correcta en este caso?, ¿es una buena cota para el orden de complejidad del programa? Justifica las respuestas.

5.6. Calcular el tiempo de ejecución y el orden exacto del siguiente algoritmo:

```
p = 0
for i = 1 to n
  p = p + i*i
  for j = 1 to p
    write (a[p, j])
  endfor
endfor
```

5.7. Resolver la ecuación recurrente: $t(n) - 2t(n-1) = n + 2^n$, con $n > 0$, y $t(0) = 0$.

5.8.(Clase) Demostrar que siendo $a, b, c, d \in \mathbb{R}^+$ y $e, n_0 \in \mathbb{N}$, con $e > 0$ y:

$$f(n) = \begin{cases} d & \text{Si } n < n_0 \\ a \cdot f(n-e) + bn + c & \text{En otro caso} \end{cases}$$

Se cumple que:

$$a < c \Rightarrow f \in O(n)$$

$$a = c \Rightarrow f \in O(n \cdot \log n)$$

$$a > c \Rightarrow f \in O(n^{\log_e a})$$

Estudiar que pasaría si alguna de las constantes utilizadas (a, b, c, d ó e) fuera negativa.

5.9.(TG 8.2) Calcular el tiempo de ejecución en segundos, en función del valor de n , del siguiente programa Pascal. A partir del tiempo, expresar el orden de complejidad.

procedure Algoritmo (x: array [1..MAX, 1..MAX] of real; n: integer)

var

i, j: integer;

```

    a, b, c: array [1..MAX, 1..MAX] of real;
begin
    if n > 1 then begin
        for i:= 1 to n div 2 do
            for j:= 1 to n/2 do begin
                a[i, j]:= x[i, j];
                b[i, j]:= x[i, j + n div 2];
                c[i, j]:= x[i+n div 2, j];
            end;
            Algoritmo (a, n div 2);
            Algoritmo (b, n div 2);
            Algoritmo (c, n div 2);
        end;
    end;
end;

```

5.10. Hacer una estimación de la máxima cantidad de memoria requerida por el algoritmo anterior, en función del valor de **MAX** y de **n**. Suponer que un valor de tipo real ocupa **r** bytes de memoria.

5.11. Calcular una buena cota superior de la complejidad del siguiente algoritmo:

```

procedure dos (a: array [1..n] of integer; x, y: integer)
    if x ≠ y
        if 2·a[x] < a[y]
            dos (a, x, y div 2)
        else
            dos (a, 2x, y)
        endif
    endif

```

Suponer que los datos del array **a** están ordenados de manera creciente.

5.12. Hacer un algoritmo recursivo para el cálculo del **n**-ésimo número de Fibonacci. Calcular su tiempo de ejecución y la memoria ocupada. Dar también una fórmula explícita **f(n) = ...**, no recursiva, que devuelva el **n**-ésimo número de Fibonacci.

5.13. La sucesión de Fibonacci está definida sobre los números naturales (1, 1, 2, 3, 5, 8, ...). Justifica por qué es posible que en la fórmula explícita para su cálculo (del ejercicio anterior) aparezcan números fraccionarios o irracionales. ¿Podrían aparecer en algún caso números complejos?

5.14. (EX) Dado el siguiente algoritmo en Pascal:

```

PROCEDURE invierte (A: array [0..n, 0..n] of integer);
var
    i, j: integer;
begin
    for i:= 0 to n-1 do
        for j:= i+1 to n do
            A[i, j]:= 2*A[j, i]+A[i, j];
        end;
    end;

```

- a) Calcular el tiempo de ejecución $t(\mathbf{n})$, y el número de veces que se ejecutará la instrucción de asignación sobre la matriz $\mathbf{f}(\mathbf{n})$. Si utilizas constantes en los cálculos, define el significado de cada una de ellas.
- b) Expresa el orden de complejidad de $t(\mathbf{n})$ con las notaciones Θ y o . El valor del tiempo de ejecución $t(\mathbf{n}) = \mathbf{X}$, ¿en qué unidades está dado?

5.15. (TG 6.3) Dado el programa:

```
i= 1
while i ≤ n
  if a[i] ≥ a[n]
    a[n]= a[i]
  endif
  i= i * 2
endwhile
```

Calcular:

- a) Su orden exacto. En caso de aparecer un orden condicionado eliminar la restricción.
- b) El número promedio de asignaciones en el array.

5.16. (EX) Para cada una de las siguientes afirmaciones, decir justificadamente si son ciertas o falsas:

- $\Omega(f(n)+g(n)) = \Omega(\min(f(n), g(n)))$
- $t(n+1) \in \Theta(t(n))$ para toda función $t: \mathbb{N} \rightarrow \mathbb{R}^+$
- $\Theta(2n^2 + 5n - 1) = \Theta(n^2)$

5.17. (TG 6.4) Dado el siguiente procedimiento:

```
Contar (izq, der: índices): integer
  if izq ≥ der
    return 0
  elsif a[izq] < a[der]
    return (Contar (izq*2, der) + 1)
  else
    return (Contar (izq*2, der))
  endif
```

Donde \mathbf{a} es un array global de valores enteros. Calcular para la llamada inicial de $\text{Contar}(1, n)$, el orden exacto del tiempo de ejecución en el caso más favorable, más desfavorable y promedio. ¿Coinciden los órdenes en los tres casos anteriores? ¿Coinciden los tiempos de ejecución?

5.18. (TG 7.2) Tenemos claves formadas por una serie no limitada de campos, de la forma:

$\text{Clave} = (\text{campo}_1, \text{campo}_2, \text{campo}_3, \dots)$

Sabemos que la probabilidad de que el contenido del campo_i sea igual en dos claves distintas es $1/(1+i)$. Calcular el orden exacto del tiempo promedio de la comparación de dos claves. Expresar también el tiempo de ejecución, para todos los casos, con las notaciones de complejidad vistas en clase. **Sugerencia:** para comparar las claves, primero se comprueba campo_1 , si es distinto se comprueba campo_2 , y así sucesivamente.

5.19. Encontrar O y Ω del algoritmo:

```

max = 0
for i = 1 to n
    cont = 1
    j = i + 1
    while a[i] ≤ a[j]
        j = j + 1
        cont = cont + 1
    endwhile
    if cont > max
        max = cont
    endif
endfor

```

5.20. (TG 7.3) Calcular la O pequeña del número promedio de asignaciones del algoritmo:

```

max = a[1]
for i = 2 to n
    if a[i] > max
        max = a[i]
    endif
endfor

```

5.21. (EX) El tiempo de ejecución de un programa viene dado por la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} n! & \text{Si } n \leq 5 \\ 4 * t(\lfloor n/4 \rfloor) + 3 \lfloor \log_2 n \rfloor & \text{En otro caso} \end{cases}$$

- Estudia el tiempo de ejecución, para los valores de n que sean potencia de 2. A partir del tiempo $t(n)$, expresa la complejidad. (No hace falta calcular el valor de las constantes de $t(n)$, se supondrán todas positivas.)
- Indica cuántas condiciones iniciales deberían aplicarse para calcular las constantes en la fórmula de $t(n)$, y para qué valores de n . Sólo es necesario indicar las ecuaciones que deberían resolverse, no es necesario resolverlas.
- ¿Qué ocurre con el tiempo de ejecución y con el orden de complejidad (decir si se modifican o no, y por qué) si cambiamos:
 - en lugar de $n!$, una función $g(n)$ cualquiera,
 - en lugar del primer 4 (en $4 * t(\lfloor n/4 \rfloor)$) ponemos un valor b positivo cualquiera,
 - en lugar del 3 (en $3 \lfloor \log_2 n \rfloor$) ponemos un valor c positivo cualquiera,
 - en lugar del 2 (en $3 \lfloor \log_2 n \rfloor$) ponemos un valor d positivo cualquiera?

5.22. Resolver la siguiente ecuación recurrente, indicando las condiciones iniciales que se deberían aplicar:

$$t(n) = \begin{cases} 1 & \text{Si } n < n_0 \\ 8 \cdot t(n-3) + 2^n + 1 & \text{En otro caso} \end{cases}$$

5.23. Calcular O , Ω y Θ para el siguiente programa:

```

i = 1
j = 1
while i ≤ n and j ≤ n
    if a[i, j + 1] < a[i + 1, j]
        j = j + 1
    else
        i = i + 1
    endif
endwhile

```

5.24. El tiempo de ejecución de determinado algoritmo está dado por la siguiente ecuación de recurrencia:

$$t(n) = n \cdot t^2(n-1) + t(n-2)$$

Siendo las condiciones iniciales: $t(n) = 0$, para $n \leq 0$. Calcular $t(n)$ de forma explícita y expresarlo usando las notaciones de complejidad vistas en clase. **Nota:** Se puede probar usando transformación de la imagen, inducción constructiva o expansión de recurrencias.

5.25. Dado el programa:

```

max = -∞
for i = 1 to n
    for j = 1 to m
        if a[i, j] > max
            max = a[i, j]
        endif
    endfor
endfor

```

Calcular la cota inferior y superior del número de asignaciones a la variable **max** y el **o** (o pequeña) del número promedio de instrucciones que se ejecutan.

5.26. Los tiempos de ejecución de determinados algoritmos han sido expresados mediante las ecuaciones de recurrencia siguientes. Calcular los valores de $t(n)$, calculando los valores de las constantes (indicando las condiciones iniciales que se aplican) y expresar su complejidad.

$$a) \quad t(n) = \begin{cases} n & \text{Si } n \leq 3 \\ 3 \cdot t(\lfloor n/2 \rfloor) + 2n \cdot \lfloor \log_2 n \rfloor & \text{Si } n > 3 \end{cases}$$

$$b) \quad t(1) = 3; t(2) = 4; t(3) = 2; \text{ y para } n > 3 \quad t(n) = 2t(n-1) + 3t(n-2) + 1$$

$$c) \quad t(n) = \begin{cases} n & \text{Si } n \leq 3 \\ 2 \cdot t(n-1) - 5 \cdot t(n-2) & \text{Si } n > 3 \end{cases}$$

d) Para cada uno de los resultados anteriores, discute si crees que es posible que correspondan a algoritmos reales.

5.27. (TG 7.4) Dado el siguiente programa:

Ejercicios

```

i = 1
j = n
while i < j
    if a[i] < a[j]
        i = i * 2
    else
        j = j / 2
    endif
endwhile

```

- Calcular su orden exacto. Habrá que hacerlo considerando el tamaño potencia de dos y quitando posteriormente esta restricción.
- Calcular el orden exacto en el caso en que se sustituyan las sentencias “ $i = i * 2$ ” y “ $j = j / 2$ ” por “ $i = i + 1$ ” y “ $j = j - 1$ ”, respectivamente.

5.28. (Clase) Para el siguiente programa, calcular el orden exacto del número promedio de veces que se ejecuta la instrucción “ $cont = cont + 1$ ”. Se supondrá que los valores del array **a** están desordenados y tienen una distribución aleatoria uniforme.

```

cont = 0
for i = 1 to n do
    for j = 1 to i-1 do
        if a[i] < a[j] then
            cont = cont + 1
        endif
    endfor
endfor

```

5.29. Dado el siguiente programa:

```

encontrado = false
for i = 1 to n-1 do
    if x[i] < y[1]
        if x[i+1] = y[2]
            encontrado = true
            lugar = i
        endif
    endif
    i = i + 1
endfor

```

Calcular su tiempo promedio de ejecución, suponiendo una probabilidad **p** de que dos elementos cualesquiera (de **x** o de **y**) sean iguales.

5.30. (EX) Suponer el siguiente algoritmo, que utiliza la técnica divide y vencerás.

```

PROCEDURE DivVenc (i, j: integer)
    if (i = j) then
        Combina (i, j);
    else begin
        m:= ( i+j ) / 2;
        a:= ( j - i ) / 4;
    end

```

```

        DivVenc (i, m);
        DivVenc (i+a, m+a);
        DivVenc (m+1, j);
        Combina (i, j);
    end;
```

```

PROCEDURE Combina (i, j: integer)
    for k:= i to j do
        for q:= i to j do
            Operacion(i, j, k, q);
```

- Calcular el número de veces que se ejecuta la instrucción *Operacion*, suponiendo que la llamada inicial es **DivVenc (1, n)**. Para el resultado obtenido, expresar la complejidad con las notaciones O , Ω , Θ y o (o pequeña). (Se deben calcular todas las constantes que aparezcan en la fórmula.)
- Para los resultados del punto anterior, si se ha hecho alguna suposición acerca del valor de **n**, eliminarla. Es decir, probar que el resultado es válido para cualquier valor de **n**.

5.31. (EX) Dada la siguiente ecuación de recurrencia, siendo $a, b, c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$:

$$f(n) = \begin{cases} c & \text{Si } n \leq n_0 \\ a \cdot f(n-1) + d \cdot b^n & \text{En otro caso} \end{cases}$$

Encontrar el orden de complejidad de $f(n)$, en función de la relación entre **a** y **b**. ¿Influyen los valores de **c** y **d** en el orden de complejidad? ¿Influyen en el valor de $f(n)$? Justifica las respuestas anteriores de manera razonada.

5.32. (EX) Considera el siguiente algoritmo, implementado en notación Pascal:

```

PROCEDURE Cosa (n: integer; A: array [0..n, 0..n] of integer);
```

```

var
```

```

    i, j: integer;
```

```

begin
```

```

    i:= 1;
```

```

    repeat
```

```

        A[i, 0]:= 0;
```

```

        for j:= 1 to i do begin
```

```

            A[i, j]:= 0;
```

```

            Rec (i);
```

```

        end;
```

```

        i:= i+1;
```

```

    until i>=n;
```

```

end;
```

```

PROCEDURE Rec (n: integer);
```

```

begin
```

```

    if n=1 then
```

```

        A[n, n]:= A[n, n]+1;
```

```

    else begin
        Rec (n-1);
        Rec (n-1);
    end;
end;

```

- Estudia el número de veces $f(n)$ que se realiza la asignación “ $A[n, n] := A[n, n]+1$ ”, dentro del procedimiento **Rec**, para la ejecución de la llamada inicial **Cosa (n, A)**.
- Calcula el tiempo de ejecución $t(n)$ del procedimiento **Cosa**, en número de instrucciones. Expresa la complejidad de $t(n)$ usando las notaciones O , Ω ó Θ , y con la notación o .
- Suponiendo que calculamos el tiempo de ejecución $g(n)$ en unidades de tiempo (por ejemplo, milisegundos) ¿qué relación existe entre $t(n)$ y $g(n)$? ¿Por qué?

5.33. (EX) Calcular una cota inferior y superior para el orden de complejidad del siguiente algoritmo:

```

PROCEDURE Div23 (k: integer);
begin
    if  $k \leq 1$  then
        write (“Acaba la recursividad”)
    else if EsPrimo( $2^k-1$ ) then
        Div23 ( $\lfloor k/2 \rfloor$ )
    else begin
        Div23 ( $\lfloor k/3 \rfloor$ );
        Div23 ( $\lceil k/3 \rceil$ );
    end;
end;

```

Suponer que la operación **EsPrimo(x)** requiere un tiempo constante.

5.34. (EX) Dado el siguiente algoritmo de suavizado uniforme de una imagen en escala de grises:

```

PROCEDURE Suavizado (Imagen: array [1..n, 1..n] of integer; m: integer);
var
    i, j: integer;
begin
    i:= m+1;
    repeat
        for j:= m+1 to n-m do
            mascara (Imagen, i, j, m);
        i:= i+1;
    until i > n-m;
end;

```

```

PROCEDURE mascara (A: array [1..n, 1..n] of integer; x, y, tam: integer);
var

```

```

    i, j, acum: integer;
begin
    acum:= 0;
    for i:= y-tam to y+tam do
        for j:= x-tam to x+tam do
            acum:= acum + A[i, j];
        A[x, y]:= acum/(4*tam*tam);
    end;
end;

```

- ¿En función de qué parámetros está el tiempo de ejecución del procedimiento **mascara**? ¿Y el procedimiento **Suavizado**?
- Calcula el tiempo de ejecución del procedimiento **mascara(A, x, y, tam)**.
- Calcula el tiempo de ejecución de **Suavizado(I, m)**, en función del tamaño de la entrada. Expresa la complejidad del tiempo usando las notaciones Θ y o-pequeña.

5.35. (EX) Resolver la siguiente ecuación de recurrencia. Encontrar el valor de $t(n)$ y el orden de complejidad. No es necesario calcular las constantes de $t(n)$.

$$t(n) = \begin{cases} 5 \log_2 n & \text{Si } n \leq 4 \\ 6t(n-2) - 4t(n-3) - 3n & \text{En otro caso} \end{cases}$$

5.36. (EX) Decir si las siguientes afirmaciones son ciertas o falsas, justificando la respuesta muy brevemente:

- $4n^2 + 2n + \log_2 n \in O(n^2 + \log n)$
- $(2n)! \in \Omega(n!)$
- $\Omega(f(n) \mid n=2^k) \subset \Omega(f(n) \mid n=2^k)$

5.37. (EX) Dado el siguiente algoritmo en sintaxis Pascal:

PROCEDURE Acumula (P: array [1..n] of integer; m: integer);

```

var
    i, acum: integer;
begin
    for i:= 1 to n do
        if P[i] < 1 then P[i]:= 1;
    i:= 1;
    acum:= 0;
    while true do begin
        acum:= acum + P[i];
        i:= i+1;
        if i > n then i:= 1;
        if acum >= m then exit;
    end;
end;

```

- ¿En función de qué parámetros está el tiempo de ejecución del procedimiento **Acumula**? ¿El tiempo de ejecución depende del contenido de los datos de entrada o sólo del tamaño de la entrada?

- b) Calcula el tiempo de ejecución del procedimiento **Acumula**. Expresa la complejidad del tiempo resultante usando las notaciones asintóticas que conozcas.

- 5.38. (EX) Encontrar una fórmula (no recursiva) para calcular el resultado que devuelve la siguiente función, para la llamada **Rec(n)**. Calcular también el orden de complejidad del número de veces que se ejecuta la instrucción de asignación sobre la variable **a**.

FUNCTION Rec (i: integer): real;

var

 a: real;

begin

 if $i \leq 4$ then return i

 else begin

 a:= 4*Rec (i-1);

 a:= a - 3*Rec (i-2);

 return a;

 end;

end;

- 5.39. (EX) Decir si las siguientes afirmaciones son ciertas o falsas, justificando la respuesta muy brevemente:

i. $\Omega(n) \subseteq O(n^2)$

ii. $f(n+1) \in O(f(n)), \forall f: \mathbb{N} \rightarrow \mathbb{R}^+$

- iii. Los algoritmos que usan divide y vencerás tienen siempre órdenes de complejidad exponenciales.

- 5.40. (EX M01) ¿Cuál es el orden $O(\cdot)$ del número de nodos de un árbol AVL en el peor caso (máximo desbalanceo permitido) en función de la altura **h**? ¿Coincide con el mejor caso de árbol AVL (esto es, $O(2^h)$), es mayor o menor? Teniendo en cuenta lo anterior, ¿cuál es el orden del procedimiento de búsqueda de un elemento en el peor caso de árbol AVL? ¿Es igual, mayor o menor que en el mejor caso de árbol AVL?

- 5.41. (EX M01) Resolver la siguiente ecuación de recurrencia.

$$t(n) = \begin{cases} 0 & \text{Si } n \leq 3 \\ \sqrt{3t^2(\lfloor n/2 \rfloor) - 2t^2(\lfloor n/4 \rfloor) + n} & \text{En otro caso} \end{cases}$$

- a) Encontrar una fórmula para el valor de **t(n)**. Expresa la complejidad de la función **t(n)** usando las notaciones asintóticas adecuadas.
- b) Si has utilizado alguna condición sobre el tamaño de la entrada para el cálculo anterior, comprobar que la condición puede ser eliminada. En particular, ¿qué resultado es válido para todos los valores de **n**: el valor de **t(n)**, el valor del orden de complejidad, los dos o ninguno?

- c) ¿Crees que una ecuación de recurrencia como la anterior puede surgir tras el conteo de instrucciones de un programa? ¿Por qué? Justifica la respuesta brevemente.

- 5.42. (EX M01) Escribir una función que devuelva siempre el mismo resultado que la siguiente función, pero tardando **un tiempo de ejecución constante**. Se supone que **m** es una variable global.

```

FUNCTION Simple (i: integer): integer;
var
  i, acum: integer;
begin
  if i = m then
    return m
  else if i >= m+1 then
    return 1
  else
    return 2*Simple (i+1) - Simple (i+2);
  end;
end;

```

- 5.43. (EX S01) La ecuación de recurrencia de abajo surge del análisis de cierto algoritmo que usa divide y vencerás.

$$f(n) = \begin{cases} 4n & \text{Si } n \leq 3 \\ n^2 + 5 & \text{Si } 3 < n \leq 10 \\ 2f(\lfloor n/2 \rfloor) + 3f(\lfloor n/4 \rfloor) & \text{En otro caso} \end{cases}$$

- a) Encontrar una fórmula no recursiva para el valor de **f(n)**, calculando también el valor de las constantes que puedan aparecer. Expresa la complejidad de **f(n)** usando las notaciones asintóticas O , Θ , Ω y o .
- b) Escribe un procedimiento en notación Pascal (incluida la cabecera del mismo) cuyo análisis de complejidad pueda dar como resultado la anterior ecuación. ¿Cuál es la unidad medida por **f(n)** (número de instrucciones, segundos, ...)? **Sugerencia:** intenta que el procedimiento sea lo más simple posible.
- c) Suponer que en el caso de abajo cambiamos $n/4$ por $n/3$, teniendo: $f'(n) = 2f'(\lfloor n/2 \rfloor) + 3f'(\lfloor n/3 \rfloor)$. Demostrar por inducción que $O(f(n)) \subseteq O(f'(n))$.

- 5.44. (EX D01) Calcular el tiempo que tarda la ejecución del siguiente trozo de código.

```

(1) va:= 0;
(2) for i:= 1 to p do begin
(3)   for j:= 1 to p do begin
(4)     va:= va + 1;
(5)     Delay (1000.0/va);
(6)   end;
(7) end;

```

Tener en cuenta que la instrucción **Delay (x : real)** tarda siempre **x** milisegundos. El tiempo de las demás instrucciones depende de la máquina donde se ejecute.

5.45. (EX D01) Considerar las dos siguientes operaciones.

```
procedure Primero (n: integer);
var
  i: integer;
  acum: real;
begin
  if n>0 then begin
    acum:= 0;
    for i:= 1 to 2 do
      acum:= acum + Segundo (n);
    write (acum);
  end;
end;
```

```
function Segundo (x: integer): real;
var
  j, k: integer;
begin
  k:= 0;
  for i:= 1 to x do
    k:= k+random(100);
  Primero (x div 2);
  Primero (x div 2);
  Segundo:= k/x;
end;
```

- Hacer una estimación del tiempo de ejecución y el orden de complejidad del procedimiento **Primero**, para la llamada **Primero(n)**. No es necesario calcular el valor de las constantes que puedan aparecer. Si aparece alguna condición en el orden, elimínala si es posible.
- Calcular el número exacto de veces que se ejecuta la instrucción “write(acum)”, para la llamada **Primero(m)**. En este caso, ¿es posible eliminar la restricción? ¿Cuántas veces se ejecutará la instrucción “write(acum)” para **Primero(1025)**? ¿Y para **Primero(1026)**?

5.46. (EX M02) Considera la variante de los números de Fibonacci, que denominaremos “números de **cuatrinacci**”, definida a continuación. El n -ésimo número de cuatrinacci es igual a 3 veces el número $(n-1)$ -ésimo, más 2 veces el $(n-2)$ -ésimo, menos el n -ésimo número de cuatrinacci. El primer y el segundo números de cuatrinacci valen 1 y 3, respectivamente. Se pide lo siguiente.

- Escribe tres posibles implementaciones para el cálculo del n -ésimo número de cuatrinacci usando: un método descendente de resolución de problemas (por ejemplo, un algoritmo de divide y vencerás), un método ascendente (por ejemplo, de programación dinámica), y un procedimiento que devuelva el resultado de forma directa, mediante una simple operación aritmética. **Ojo:** las implementaciones deben ser muy simples y cortas.
- Haz una estimación del orden de complejidad de los tres algoritmos del apartado anterior. Compara los órdenes de complejidad obtenidos, estableciendo una relación de orden entre los mismos.

5.47. (EX M02) Calcula el número de instrucciones que ejecuta el siguiente trozo de código, donde todas las variables son de tipo real y toman valores próximos a 0.

```
(1) act:= 1/10;
(2) acum:= sqrt(act);
(3) while act >= mini do begin
```

- (4) act:= act * 1/10;
- (5) acum:= acum + sqrt(act);
- (6) end;
- (7) return acum;

Sugerencia: estudiar el número de veces que se ejecuta el bucle para ciertos valores de **mini**, por ejemplo, para 10^{-3} , para 10^{-7} , etc.

¿Qué pasa si en las líneas (1) y (4) sustituimos 1/10 por 1/d, siendo **d** un natural cualquiera? ¿Cuál sería el tiempo de ejecución?

- 5.48. (EX S02) Dadas las dos siguientes operaciones, hacer una estimación del tiempo de ejecución del procedimiento **Ping**, para la llamada **Ping(n)**. Indica claramente el significado de todas las constantes que uses. Expresa el orden de complejidad usando la notación que creas más adecuada y la o-pequeña. ¿Qué cambios ocurren en el tiempo de ejecución y en el orden de complejidad si en la función **Pong**, cambiamos la llamada **Ping(0)** por **Ping(1)**?

<pre> procedure Ping (n: integer); var i, j: integer; valor: real; begin (1) valor:= 1.0; (2) for j:= 1 to 2*n do (3) for i:= 1 to j do (4) valor:= 2*Pong (i); (5) write (valor); end;</pre>	<pre> function Pong (m: integer): real; var tmp: real; begin (6) tmp:= sin(m*3.1416/180.0); (7) tmp:= tmp*tmp + 1.0; (8) Ping (0); (9) Pong:= tmp/m; end;</pre>
---	--

- 5.49. (EX S02) Resuelve la siguiente ecuación de recurrencia. Si aparecen más de 3 constantes en el tiempo de ejecución, no es necesario calcularlas pero sí escribir las ecuaciones que habría que resolver para obtener su valor.

$$f(n) = \begin{cases} 0 & \text{Si } n \leq 2 \\ 1 & \text{Si } 2 < n \leq 4 \\ 4n + 3 f(n-1) + 4 f(n-2) & \text{En otro caso} \end{cases}$$

- 5.50. (EX M03) Calcula el tiempo de ejecución y el orden de complejidad del siguiente algoritmo, para los valores de **n** que sean múltiplos de 2. Si aparecen más de 3 constantes en el tiempo no es necesario calcularlas. ¿Es válido el resultado para los valores de **n** impares?

```

procedure Erude (n, m: entero; var r: real);
(1) if n<1 then
(2)   r:= r+n+m;
(3) else if n=2 then
(4)   r:= r*n*m;
(5) else begin
(6)   m:= m*2;
(7)   Erude (n-1, m-1, r);
(8)   if n mod 2 = 0 then begin
```

Ejercicios

```

(9)     for i:= 1 to 2 do
(10)        Erude (n-3, m-i, r);
(11)    end;
(12)    else begin
(13)        for i:= 1 to n do
(14)            r:= r*m*i;
(15)        end;
(16)    end;

```

5.51. (EX S03) El tiempo de ejecución de determinado algoritmo viene dado por la siguiente ecuación de recurrencia:

$$t(n) = \begin{cases} 2n^2 + 8 & \text{Si } n \leq 4 \\ 4t(n-1) - 5t(n-2) + 2t(n-3) + \log_2 n + 5 & \text{Si } n > 4 \end{cases}$$

Encontrar el orden exacto, Θ , del tiempo de ejecución del algoritmo. **Sugerencia:** en caso de no poder despejar el tiempo, probar buscando cotas superiores e inferiores del mismo.

5.52. (EX S03) Considerar el siguiente algoritmo para generar todas las combinaciones de n elementos de 4 en 4. Se supone que **marca** y **conv** son arrays globales de tamaño n , inicializados con valor 0. La llamada inicial al procedimiento es: **Combinaciones(1)**.

procedure Combinaciones (k: integer)

```

(1) for i:= 1 to n do begin
(2)   if marca[i] = 0 then begin
(3)     conv[k]:= i
(4)     if k = 4 then
(5)       escribir (conv[1], conv[2], conv[3], i)
(6)     else begin
(7)       marca[i]:= 1
(8)       Combinaciones(k+1)
(9)       marca[i]:= 0
(10)    end
(11)  end
(12) end

```

Calcular el tiempo de ejecución del algoritmo. Expresar el orden de complejidad del algoritmo usando la notación asintótica que consideres más adecuada.

5.53. (EX J04) Calcular el tiempo de ejecución, el orden de complejidad y el valor devuelto por el siguiente algoritmo.

operación Petiva (n: entero): entero

```

j:= 1
p:= 0
para i:= 1, ..., n hacer
    si Impar(i) entonces
        j:= j*2

```

```

        sino
            para k:= 1, ..., j hacer
                p:= p + 1
            finpara
        finsi
    finpara
    devolver p

```

5.54. (EX) Considerar el siguiente algoritmo recursivo.

operación Recu (var a: array [1..MAX, 1..MAX] de entero; n: entero)
var tmp: array [1..MAX, 1..MAX] de entero

```

    si n>1 entonces
        Recu (tmp, n/2)
    finsi
    para i:= 1, ..., n hacer
        para j:= i+1, ..., n hacer
            tmp[i, j]:= a[i, j] * tmp[i, j]
        finpara
    finpara
    si n>1 entonces
        Recu (a, n/2)
    finsi

```

- Estimar el tiempo de ejecución y el orden de complejidad del algoritmo.
- Calcular la o-pequeña del número de multiplicaciones que se ejecutan.
- Calcular la memoria que necesita el algoritmo para ejecutarse. Suponer que cada entero ocupa k bytes, y que el paso por variable no necesita reservar memoria.

5.55. (EX J05) Dada la función:

```

operación Pares (a: array [1,..,n] de entero; izq, der: entero) : entero
var m: entero
m:= (der+izq)/2
si m ≠ izq entonces
    si par(a[m]) entonces
        devolver Pares(a, izq, m) + 1
    fin si
    devolver 1
sino
    devolver 0
fin si

```

Obtener razonando y justificando las respuestas:

- Las entradas en los casos más favorable y más desfavorable.
- Unos buenos valores para Ω y O del tiempo de ejecución. ¿Qué ocurre con Θ ?
- o y Θ del tiempo promedio.
- Una fórmula de la ocupación máxima de memoria. Se supone que todos los parámetros se pasan por valor.

5.56. (EX J05) Obtener la fórmula del tiempo de ejecución, en función del tamaño del problema y del caso base, de un algoritmo recursivo cuyo tiempo tiene la siguiente forma:

$$t(n) = 3 t(n-6) + 2, \text{ si } n > b ; \quad t(n) = n^2, \text{ si } n \leq b .$$

Ojo, se pide la fórmula del tiempo, no sólo la notación asintótica.

5.57. (EX J05) Hacer una estimación, lo más precisa posible, del tiempo de ejecución del siguiente algoritmo, y obtener Ω , O , Θ y la o pequeña:

```

sum:=0
para i:=1 hasta n hacer
  para j:=1 hasta i2 hacer
    si j mod i == 0 entonces
      para k:=1 hasta j hacer
        sum:= sum+1
      finpara
    finpara
  finpara
finpara

```

5.58. (EX) Ordena las siguientes cotas de complejidad de menor a mayor, indicando también las que son iguales: $O(\log_{10}n)$, $O(n^n)$, $O(2n^2)$, $O(n!)$, $O(3 \cdot 2^n)$, $O(n \cdot \log \log n)$, $O(20)$, $O(3^n)$, $O(n \cdot \log n)$, $O(2 \log_2 n)$, $O(n(n + \log n))$, $O(1)$, $O(n2^n)$, $O(n+1)$, $O(2^n + 3^n)$, $O(n^{n+1})$. Justifica brevemente los casos que no sean triviales. Indica por lo menos 8 ejemplos de algoritmos conocidos que tengan algunos de esos órdenes de complejidad. Por ejemplo, $O(n^3) \rightarrow$ algoritmo clásico de multiplicación de matrices cuadradas de tamaño n .

5.59. (EX) Dada una matriz **a** de tamaño **nxn**, y la función:

```

operación Nada
  para i= 1,2,...,n/2
    tratar1(i)
  finpara
  para i= n/2+1,n/2+2,...,n
    tratar2(i)
  finpara

```

donde **tratar1** y **tratar2** son:

operación tratar1 (pos: entero)

```

cont= 1
para j= pos-1,pos-2,...,1
  si a[pos][pos]<a[j][pos+cont]
    a[pos][pos]= a[j][pos+cont]
  finpara
  cont++
finpara
cont= 1
para j= pos-1,pos-2,...,1
  si a[pos][pos]<a[pos+cont][j]
    a[pos][pos]= a[pos+cont][j]
  finpara
  cont++
finpara

```

operación tratar2 (pos: entero)

```

cont= 1
para j= pos+1,pos+2,...,n
  si a[pos][pos]<a[pos-cont][j]
    a[pos][pos]= a[pos-cont][j]
  finpara
  cont++
finpara
cont= 1
para j= pos+1,pos+2,...,n
  si a[pos][pos]<a[j][pos-cont]
    a[pos][pos]= a[j][pos-cont]
  finpara
  cont++
finpara

```

Obtener (razonando y justificando la respuesta) unos buenos valores de Ω y O del tiempo de ejecución, y θ del tiempo promedio de ejecución.

5.60. (EX) Un tiempo se obtiene con la siguiente recurrencia $t(n)=3*t(n/2)+2n^2+n$, si $n>2$, y $t(n)=n^3$, si $n\leq 2$, obtener el orden exacto y la o-pequeña del tiempo de ejecución.

5.61. (EX J06) Considerar las siguientes recurrencias:

1. $t(n)= t(n/2)+1$	2. $t(n)= 2t(n/2)+1$	3. $t(n)= 2t(n/2)+n$	4. $t(n)= 3t(n/2)+n$
5. $t(n)= 4t(n/2)+n$	6. $t(n)= 7t(n/2)+n^2$	7. $t(n)= 8t(n/2)+n^2$	

Responder a una de las dos siguientes preguntas, sin dar por supuestas las fórmulas maestras vistas en clase:

- a) (100%) Deducir a partir de dichas recurrencias una fórmula maestra que las incluya a todas, resolverla de manera general y aplicarla de manera particular a cada una para obtener su orden exacto. Para cada una de las recurrencias, indicar un algoritmo conocido cuyo tiempo de ejecución se corresponda con ella.
- b) (70%) Resolver individualmente cada recurrencia y obtener su orden exacto. Para cada una de las recurrencias, indicar un algoritmo conocido cuyo tiempo de ejecución se corresponda con ella.