Ejercicio_1.

Usar las relaciones $\subset$ y $=$ para ordenar los órdenes de complejidad, O, Ω, y Θ, de las siguientes funciones:

nlogn, n^2logn, n^8, n^{1+a}, $(1+a)^n$, $(n^2+8n+\log^3 n)^4$, n^2/ logn, 2^n, siendo a una constante real, 0<a<1.

- ### Solución

Como todas las funciones son continuas, las comprobaciones que hay que hacer son:

1. Respecto al orden de complejidad O

- para $O(f(n)) \subset O(g(n))$ hay que ver que $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$
- para $O(f(n)) = O(g(n))$ hay que ver que $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \mathcal{R}^+ \neq 0$

$\Rightarrow$ tenemos que:

$O(n\log n) \subset O(n^{1+a}) \subset O(n^2/\log n) \subset O(n^2\log n) \subset O(n^8) \subset O((n^2+8n+\log^3 n)^4) \subset O((1+a)^n) \subset O(2^n)$

2. Respecto al orden de complejidad Ω

- para $\Omega(f(n)) \subset \Omega(g(n))$ hay que ver que $\lim_{n\to\infty}\left(\frac{g(n)}{f(n)}\right) = 0$
- para $\Omega(f(n)) = \Omega(g(n))$ hay que ver que $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \mathcal{R}^+ \neq 0$

$\Rightarrow$ tenemos que:

$\Omega(n\log n) \supset \Omega(n^{1+a}) \supset \Omega(n^2/\log n) \supset \Omega(n^2\log n) \supset \Omega(n^8) \supset \Omega((n^2+8n+\log^3 n)^4) \supset \Omega((1+a)^n) \supset \Omega(2^n)$

3. Respecto al orden de complejidad Θ al definirse como la intersección de O y Ω sólo se puede asegurar que

$\Theta(n^8) = \Theta((n^2+8n+\log^3 n)^4)$

y el resto de los órdenes **NO** se pueden comparar.

Ejercicio_2.

Usando la definición de notación asintótica Θ demostrar que $512n^2 + 5n \in \Theta(n^2)$.

- Solución

- Para demostrar que $512n^2 + 5n \in O(n^2)$ hay que encontrar un n_0 y una cte $c>0$ tal que $512n^2 + 5n \leq cn^2$ $\forall$ $n \geq n_0$.
  Basta dividir esa inecuación por n^2 para obtener $512 + \frac{5}{n} \leq c$. Por tanto, basta tomar $n_0=5$ y $c=513$
- Para demostrar que $512n^2 + 5n \in \Omega(n^2)$ hay que encontrar un n_0 y una cte $c>0$ tal que $512n^2 + 5n \geq cn^2$ $\forall$ $n \geq n_0$.
  Basta tomar $n_0=0$ y $c=1$

Ejercicio_3.

Usando las definiciones de notación asintótica, demostrar si son verdaderas o falsas las afirmaciones siguientes:

(a) $(n+1)! \in O(3(n!))$

(b) $n^2 \in \Omega((n+1)^2)$

- **Solución**

(a) $(n+1)! \in O(3(n!))$. Es **FALSO**. Demostración por reducción al absurdo:
- Si suponemos que es verdadero, $\exists$ un n_0 y una cte $c>0$ tal que $(n+1)! \leq c3n!$ $\forall$ $n \geq n_0$.
- Y entonces se cumpliría que $(n+1) \leq c3$ $\forall$ $n \geq n_0$ que es IMPOSIBLE!!

(b) $n^2 \in \Omega((n+1)^2)$. Es **VERDADERO**. Demostración definición (acotaciones):

$$n^2 \geq \boldsymbol{c}(n+1)^2 \Rightarrow n^2 \geq \boldsymbol{c}(n^2+2n+1) \Rightarrow$$

$\Rightarrow 1 \geq c + \frac{c2}{n} + \frac{c}{n^2}$ Basta tomar $0< c \leq \frac{1}{4}$ para que se satisfagan las acotaciones $\forall$ $n \geq 1$

Ejercicio_4.

Escribir un algoritmo que, dado un entero positivo $n \geq 1$, verifique si es un número triangular. Analizar el algoritmo implementado.

NOTAS:

- Un número natural $n \geq 1$ es *triangular* si es la suma de una sucesión ascendente no nula de naturales consecutivos que comienza en 1. Por tanto, los cinco primeros números triangulares son:

  1, **3** = 1+2, **6** = 1+2+3, **10** = 1+2+3+4 y **15** = 1 + 2 + 3 + 4 + 5.

- Un posible algoritmo para comprobar que un número natural positivo n *es* triangular consiste en calcular sucesivamente los números triangulares y compararlos con n*.* En cada iteración, se genera el siguiente número triangular, si es igual a n se termina con éxito; en caso contrario, el número generado será mayor que n y se termina con fracaso.

➢ Solución

Un posible algoritmo para comprobar que un número natural positivo *n* es triangular consiste en calcular sucesivamente los números triangulares y compararlos con *n*. En cada iteración, se genera el siguiente número triangular (*numTri*). Si *numTri* = *n*, termina con éxito; si *numTri* > *n*, termina con fracaso.

```
función EsTriangular (n: positivo) return boolean
   numTri ← 1
   UltimoSumando ← 1
   mientras numTri < n hacer
        UltimoSumando ← UltimoSumando + 1
        numTri ← numTri + UltimoSumando
   fmientras
   return (numTri = n)
ffunción
```

- Las operaciones del bucle requieren tiempo constante y las iteraciones se realizan hasta que *numTri* es igual o mayor que *n*. Esto es, el número de veces que se realiza este bucle (por ej k) es igual al número de sumandos que tiene la sucesión ascendente no nula de naturales consecutivos que comienza en 1 y cuya suma es

$$n = \sum_{i=1}^{k} i = \frac{(1+k)k}{2}$$

- Despejando k de la ecuación $k^2 + k - 2n = 0$, se obtiene el orden del número de veces que se ejecuta el bucle ⇒ $k = -(1 \pm \sqrt{1+8n})/2 \in \Theta(\sqrt{n})$, tanto si se sale del bucle con éxito como con fracaso.

  Por tanto, EsTriangular (n) $\in \Theta(\sqrt{n})$.

➢ Observaciones:

- o Si se ha salido del bucle porque *numTri* > *n*, eso significa que en la iteración anterior, *numTri* < *n* y tras realizar una suma más *numTri* > *n*; pero el orden del algoritmo sigue siendo el mismo.

Ejercicio_5.

Dado el algoritmo siguiente, que determina si una cadena C es palíndroma:

```
función PAL (C, i, j) : booleano;
   if i ≥ j then
       return cierto
   else
       if C(i)≠C(j) then
               return falso
       else
               return PAL(C, i+1, j-1)
   ffunción
```

- **Calcular** el tiempo de ejecución para PAL(C, 1, n) en el caso peor y en el caso medio, suponiendo equiprobabilidad de todas las entradas y siendo {a, b} el alfabeto que forma las cadenas

NOTA:

- Para calcular la eficiencia temporal considerar como operación característica el número de comparaciones entre componentes de la cadena (C(i)≠C(j)), siendo n=j-i+1 el tamaño de la cadena.

Solución

Sea n=j-i+1 el tamaño de la cadena. Consideraremos como operación característica el número de comparaciones entre componentes de la cadena (C(i)≠C(j)).

1. Para el **caso peor** el nº de comparaciones vendrá dado por la función:

$$\mathbf{t}(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ 1+\mathbf{t}(n-2) & \text{si } n > 1 \end{cases}$$

   Resolviendo por ec. característica $(x^2-1)(x-1)=0 \Rightarrow r_1=1$ doble $r_2=-1 \Rightarrow t(n)=c_1+c_2n+c_3(-1)^n \Rightarrow$

   $\Rightarrow c_1=-1/4;\ c_2=1/2;\ c_3=1/4 \Rightarrow t(n)= n/2+1/4(-1)^n-1/4= n/2$ (si n par)

   $t(n)=n/2 \in \mathcal{O}(\boldsymbol{n})$

2. Para el **caso medio** la equiprobabilidad de las entradas hace que la probabilidad de que los extremos sean distintos sea 1/2 ya que el alfabeto de entrada es {a, b} y en ese caso el nº de comparaciones que se realizan es 1; en caso que sean iguales (cuya probabilidad es 1/2) se producirán el nº promedio de comparaciones para una cadena de tamaño (n-2) más 1. Es decir:

$$\mathbf{t}(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \frac{1}{2}+\frac{1}{2}(1+\mathbf{t}(n-2)) & \text{si } n > 1 \end{cases}$$

   Resolviendo $\Rightarrow t(n) = 2-\frac{2}{\sqrt{2^n}} \in \mathcal{O}(\mathbf{1})$

Ejercicio_6.

Para resolver cierto problema se dispone de dos algoritmos, $\mathbf{A_1}$ y $\mathbf{A_2}$, de divide y vencerás:

- $\mathbf{A_1}$ descompone el problema de tamaño **n** en tres subproblemas de tamaño **n/2** y cuatro subproblemas de tamaño **n/4**. La división y combinación requieren $\mathbf{3n^2}$, y el caso base, con **n** menor que 5, es **n!**
- $\mathbf{A_2}$ descompone el problema de tamaño **n** en un subproblema de tamaño **n-3** y dos de tamaño **n-2**. El tiempo de la división y combinación es despreciable, y el caso base, con **n** menor que 5, es de orden constante.
  1. Calcular el orden de complejidad de los dos algoritmos.
  2. Estudiar cuál de los dos algoritmos es más eficiente.

➢ Solución

Según el enunciado, las ecuaciones de recurrencia de los dos algoritmos, que llamaremos t_1 y t_2, son las siguientes:

$$t_1(n)=\begin{cases} n! & \text{Si } n < 5 \\ 3t_1(n/2)+4t_1(n/4)+3n^2 & \text{Si } n > 4 \end{cases}$$

$$t_2(n)=\begin{cases} c & \text{Si } n < 5 \\ 2t_2(n-2)+t_2(n-3) & \text{Si } n > 4 \end{cases}$$

Resolviendo ambas ecuaciones de recurrencia por el método de la ecuación característica, el resultado es:

$$t_1(n) = c_1 n^2 + c_2 n^2 \log_2 n + c_3 (-1)^{\log_2 n} \in O(n^2 \log n)$$

$$t_2(n) = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n + c_3(-1)^n \in O(1{,}62^n)$$

$O(1{,}62^n) \supset O(n^2 \log n) \Rightarrow$ el segundo algoritmo tiene un orden de complejidad mayor por lo que el primero es más eficiente.

Ejercicio_7. Algoritmo Recursivo para la Búsqueda Ternaria.

El algoritmo de “búsqueda ternaria” realiza una búsqueda de un elemento en un vector ordenado.
La función compara el elemento a buscar “clave” con el que ocupa la posición n/3 y si este es menor que el elemento a buscar se vuelve a comparar con el que ocupa la posición 2n/3. En caso de no coincidir ninguno con el elemento buscado se busca recursivamente en el subvector correspondiente de tamaño 1/3 del original.

1. Escribir un algoritmo para la búsqueda ternaria.
2. Calcular la complejidad del algoritmo propuesto.
3. Comparar el algoritmo propuesto con el de búsqueda binaria.

➢ Solución

- **Algoritmo búsqueda ternaria:**

```
funcion BusquedaTernaria(V[1..n];primero,ultimo,clave:int)
    si primero ≥ ultimo  entonces
       devolver V[ultimo]=clave;
    fsi
    tercio ← ((ultimo - primero + 1) / 3);
    si clave = V[primero+tercio] entonces
       devolver cierto;
    sino
       si clave < V[primero+tercio] entonces
              devolver BusquedaTernaria (V, primero, primero+tercio-1, clave);
       sino
              si clave = V[ultimo-tercio] entonces
                     devolver cierto;
              sino
                     si clave < V[ultimo-tercio] entonces
                        devolver BusquedaTernaria (V, primero+tercio+1, ultimo-tercio-1, clave);
                     sino
                            devolver BusquedaTernaria (V, ultimo-tercio+1, ultimo, clave);
                     fsi

              fsi
       fsi
    fsi
ffuncion
```

- Para el análisis del algoritmo en el **caso peor**, cuando la clave es mayor a cualquier elemento de la lista, observamos que en cada llamada recursiva el tamaño del vector es un tercio del tamaño del vector en la llamada anterior. Suponiendo que n es una potencia de 3, la complejidad es:

$$\mathbf{T}(n) = \begin{cases} 4 & \text{si } n = 1 \\ T(n/3) + 18 & \text{si } n > 1 \end{cases}$$

   - Resolviendo $T(n)=18\boldsymbol{log_3}\, n+4 \in \Theta(\boldsymbol{log\ n})$

- **Algoritmo búsqueda binaria:**

```
funcion BusquedaBinaria(V[1..n];primero,ultimo,clave:int)
    si primero ≥ ultimo  entonces
       devolver V[ultimo]=clave;
    fsi
    mitad ← ((ultimo - primero + 1) / 2);
    si clave = V[mitad] entonces
       devolver cierto;
    sino
       si clave < V[mitad] entonces
              devolver BusquedaBinaria (V, primero, mitad-1, clave);
       sino
              si clave > V[mitad] entonces
                 devolver BusquedaBinaria (V, mitad+1, ultimo, clave);
              fsi
       fsi
    fsi
ffuncion
```

- Para el análisis del algoritmo en el **caso peor**, cuando la clave es mayor a cualquier elemento de la lista, observamos que en cada llamada recursiva el tamaño del vector es la mitad del tamaño del vector en la llamada anterior. Suponiendo que n es una potencia de 2, la complejidad es:

$$\mathbf{T}(n) = \begin{cases} 4 & \text{si } n = 1 \\ T(n/2) + 13 & \text{si } n > 1 \end{cases}$$

  - Resolviendo $T(n)=13\boldsymbol{log}_2 n+4 \in \Theta(\boldsymbol{log\ n})$

- **Comparación algoritmos**

  Como $13\boldsymbol{log}_2 n+4 < 18\boldsymbol{log}_3 n+4 \Rightarrow$ realiza menos operaciones la búsqueda binaria aunque los dos sean $\Theta(\boldsymbol{log\ n})$

Ejercicio_8.

Escribir un algoritmo que dados un vector de **n** enteros y un entero **X**, determine si existen en el vector dos números cuya suma sea X.
El tiempo del algoritmo debe ser de O(n**log* n). Analiza el algoritmo y demuestra que es así.

➢ Solución:

1. Un posible **algoritmo** es:

```
funcion buscarN1N2 (V[1..n]; X: int)
   N1,N2: int;
   Vord: [1..n]; /* Vord almacena el vector V ordenado */
empezar
   MergeSort(V, 1, n);
   Vord ← V ;
   para  ind = 1 hasta n-1 hacer
       N1 ← Vord(ind);
       N2 ← X-N1;
       pos ← BusquedaBinaria (Vord, ind+1, n, N2);
       si pos ≠ 0 entonces
           devolver (Vord(ind), Vord(pos))
       fsi;
   fpara;
   devolver (0,0) /*suponiendo 0 los valores devuelto cuando no se encuentre el número y X≠0 */
ffuncion
```

2. **Análisis** del orden del algoritmo:

1. La ordenación de n enteros con MergeSort es **Θ(n *log* n)**
2. El bucle se repetirá a lo sumo ***n*** veces con coste temporal:
   - Dos asignaciones + sentencia ***si*** requieren O(**1**)
   - La búsqueda dicotómica es logarítmica en el número de elementos: O(*log* i) (y concretamente $\forall i (i \leq n \Rightarrow O(\log i) \subseteq O(\log n))$)
   - ➢ El bucle por tanto es O (**n *log* n**)
3. Por último, aplicando la regla de la suma, **la solución** propuesta tiene **orden**: **Θ(n *log* n)**

Ejercicio_9.

Estudiar la complejidad del algoritmo de ordenación por Selección por la llamada al procedimiento, especificado a continuación, `Selection (a,1,n)`.

- El procedimiento Selección puede ser implementado como sigue:

```
procedimiento  Selection (a:vector; primero,ultimo: int);
    para i=primero hasta ultimo-1 hacer
            posmin = PosMinimo(a,i,ultimo);
            Intercambia(a, i, posmin);
    fpara
fprocedimiento Selection
```

- En el algoritmo anterior se utiliza una función **`PosMinimo`** que calcula la posición del elemento mínimo de un subvector :

```
Int función PosMinimo (a:vector;primero,ultimo:int);
/* devuelve la posición del mínimo elemento de a[primero..ultimo] */
    pmin=primero;
    para i=primero+1 hasta ultimo hacer
            si a[i] < a[pmin] entonces
                    pmin = i
            fsi;
    fpara
    return pmin;
ffuncion PosMinimo;
```

- También se utiliza el procedimiento **`Intercambia`** para intercambiar dos elementos de un vector:

```
función Intercambia (a:vector ; i , j :int );
/* intercambia a[i]  con a[j]  */
    aux = a[i] ;
    a[i] = a[j] ;
    a[j] = aux;
ffuncion Intercambia;
```

- **Solución:**

Calculamos primero los tiempos de ejecución de las funciones **`Intercambia`** y **`PosMinimo.`** El tiempo de ejecución de **`PosMinimo`** depende de la ordenación inicial además del tamaño del subvector de entrada. El estudio por casos es:

1) función **`PosMinimo`**

- **Caso peor**: la condición es siempre verdadera, por tanto:

$$T(n) = T(ultimo - primero + 1) = 1 + 2 + \sum_{i=primero+1}^{ultimo} (3 + 1 + 2 + 1 + 1) + 1 + 1 = 5 + 8(ultimo - primero)$$

- **Caso mejor**: la condición es siempre falsa, por tanto:

$$T(n) = T(ultimo - primero + 1) = 1 + 2 + \sum_{i=primero+1}^{ultimo} (3 + 2 + 1 + 1) + 1 + 1 = 5 + 7(ultimo - primero)$$

- **Caso medio**: la condición es verdadera la mitad de los casos, por tanto:

$$T(n) = T(ultimo - primero + 1) = 1 + 2 + \sum_{i=primero+1}^{ultimo} \left(3 + \frac{1}{2} + 2 + 1 + 1\right) + 1 + 1 = 5 + \frac{15}{2}(ultimo - primero)$$

2) función **`Intercambia`**

- $T(n) = 7$

3) procedimiento **`Selection`**

La complejidad para los distintos casos coincide con los de la función **`PosMinimo:`**

- **Caso peor**:

$$T(n) = 1 + 2 + \sum_{i=1}^{n-1} ((5 + 8(n - i) + 1 + 7 + 2 + 1 + 2) + 1 = 4 + \sum_{i=1}^{n-1} (18 + 8n - 8i) =$$
$$4 + 18(n - 1) + 8n(n - 1) - 8\frac{n^2}{2} = 4n^2 + 10n - 14 \in \Theta(n^2)$$

- **Caso mejor**: la condición es siempre falsa, por tanto:

$$T(n) = 1 + 2 + \sum_{i=1}^{n-1} ((5 + 7(n - i) + 1 + 7 + 2 + 1 + 2) + 1 = 4 + \sum_{i=1}^{n-1} (18 + 7n - 7i) =$$
$$= 4 + 18(n - 1) + 7n(n - 1) - 7\frac{n^2}{2} = \frac{7}{2}n^2 + 11n - 14 \in \Theta(n^2)$$

- **Caso medio**: la condición es verdadera la mitad de los casos, por tanto:

$$T(n) = 1 + 2 + \sum_{i=1}^{n-1} \left((5 + \frac{15}{2}(n - i) + 1 + 7 + 2 + 1 + 2 \right) + 1 = 4 + \sum_{i=1}^{n-1} \left(18 + \frac{15}{2}n - \frac{15}{2}i \right)$$
$$= 4 + 18(n - 1) + \frac{15}{2}n(n - 1) - \frac{15}{2}\frac{n^2}{2} = \frac{15}{4}n^2 + \frac{21}{2}n - 14 \in \Theta(n^2)$$

Ejercicio_10.

Para resolver cierto problema se dispone de un algoritmo trivial cuyo tiempo de ejecución t(n) (para problemas de tamaño n) es cuadrático ($t(n) \in \Theta(n^2)$). Se ha encontrado una estrategia Divide y Vencerás para resolver el mismo problema; dicha estrategia realiza ***D(n)= n log n*** operaciones para dividir el problema en dos subproblemas de tamaño mitad y ***C(n)= n log n*** operaciones para componer una solución del original con la solución de dichos subproblemas.

1. Calcular la eficiencia para el algoritmo Divide y Vencerás por el método de la **ecuación característica**
2. Corroborar el resultado anterior aplicando el teorema maestro.
3. Estudiar cuál de los dos algoritmos es más eficiente.

NOTA:

Teorema : La solución a la ecuación $T(n) = aT(n/b) + \Theta(n^k \log^p n)$, con $a \geq 1$, $b > 1$ y $p \geq 0$, es:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log^{p+1} n) & \text{si } a = b^k \\ O(n^k \log^p n) & \text{si } a < b^k \end{cases}$$

➢ Solución:

El tiempo de ejecución para el algoritmo nuevo viene dado por el sistema recurrente:

$$T(n) = \begin{cases} c & \text{si } n \leq 1 \\ 2T(n/2) + 2\, n \cdot \log n & \text{si } n > 1 \end{cases}$$

1. Por la ecuación característica. $n=2^k \Rightarrow k=\log n \Rightarrow (x-2)(x-2)^2=0 \Rightarrow r=2$ triple $\Rightarrow$

   $T(n)=c_1 n+c_2 n\log n + c_3 n\log^2 n \Rightarrow T(n) \in \Theta(n \cdot \log^2 n)$

2. Aplicando el teorema maestro [$T(n) = aT(n/b) + \Theta(n^k \mathrm{long}^p n)$; a=b=2, k=1, p=1 $\Rightarrow a = b^k$]

   $T(n) \in O(n^k \cdot \log^{p+1} n) \Rightarrow T(n) \in \Theta(n \cdot \log^2 n)$

3. Aplicando la regla del límite:

$$\boxed{\lim_{n \to \infty} \frac{n(\lg n)^2}{n^2} = \lim_{n \to \infty} \frac{(\lg n)^2}{n} = \lim_{n \to \infty} \frac{2(\lg n)}{n \ln 2} = 0}$$

$\Rightarrow \Theta(n \cdot \log^2 n) \subseteq \Theta(n^2)$ y **la versión DyV es más eficiente**.

Ejercicio_11.

Se realiza una variante de los números de Fibonacci que denominaremos "**Nacci**" cuya ecuación recurrente es:

$$\textbf{Nacci}(n) = \begin{cases} 1 & \text{Si } n = 1 \\ 3 & \text{Si } n = 2 \\ 3/2\ \textbf{Nacci}(n-1) + \textbf{Nacci}(n-2) & \text{En otro caso} \end{cases}$$

1. Escribir tres posibles implementaciones, **simples y cortas**, para el cálculo del n-ésimo número de **f** con las siguientes estrategias:
   a. divide y vencerás recursivo.
   b. Procedimiento iterativo.
   c. procedimiento directo, mediante una simple operación aritmética.
2. Realizar una estimación del orden de complejidad de los tres algoritmos del apartado anterior. Comparar los órdenes de complejidad obtenidos, estableciendo una relación de orden entre los mismos.

➢ Solución

- **Resolviendo la recurrencia** tenemos: $f(n) = 7/10*2^n+4/5(-1/2)^n$

1. Posibles implementaciones
   a.
```
function fDV (n: integer) : real;
      si n=1 entonces return 1
      sino
            si n=2 entonces return 3
            sino
                  return 3/2*fDV(n-1) +fDV(n-2);
            fsi
      fsi
ffunction_fDV
```
   b.
```
function fIt (n: integer) : real;
var
  T: array [1..n] de real;
  i: integer;
  T[1]:= 1;
  T[2]:= 3;
  para i:= 3 hasta n hacer
     T[i]:= 3/2*T[i-1] + T[i-2] fpara
  return T[n];
ffunction_fPD
```
   c.
```
function fDirecto (n: integer) : real;
return 7/10*2^n + 4/5*(-1/2)^n;
ffunction_fDirecto
```

2. Orden de complejidad de los tres algoritmos y comparación entre ellos
   ➢ Llamamos:
   - **t_{DV}(n)** el tiempo de **fDV(n)**,
   - **t_{IT}(n)** el tiempo de **fIT(n)** y
   - **t_{DIR}(n)** el tiempo de **fDirecto(n)**.

   ➢ Los Órdenes de complejidad son:
   - $t_{DV}(n) \in O(1+\sqrt{5}/2)^n \approx O(1.62^n)$
   - $t_{IT}(n) \in O(n)$
   - $t_{DIR}(n) \in O(1)$

   ➢ La relación entre los órdenes es:
   $O(t_{DIR}) \subset O(t_{IT}) \subset O(t_{DV})$

Ejercicio_12.

Escribir un algoritmo voraz para entregar billetes en un cajero automático que suministra la cantidad de billetes solicitada de forma que el número total de billetes sea **mínimo**. Se supone que el cajero dispone de suficientes billetes de todas las cantidades consideradas. Explicar el funcionamiento del algoritmo: cuál es el conjunto de candidatos, la función de selección, la función para añadir un elemento a la solución, el criterio de finalización, el criterio de coste, etc. Suponer billetes de 10, 20 y 50 €.
Aplicar el algoritmo para el caso que se solicite la cantidad de **570 €.**

➢ Solución

Similar al implementado en los apuntes del tema 7.

Ejercicio_13. (Ex_Sept 16)

Resolver las siguientes ecuaciones de recurrencia y calcular el orden temporal (1 punto cada apartado):

1. (1 punto)

```
function total(n:positivo)
        if n=1 then 1 else total(n-1) + 2 * parcial(n-1)
```

- siendo

```
function parcial (m:positivo)
        if m=1 then 1 else 2 * parcial(m-1)
```

2. (1 punto)

```
function total(n,m:positivo)
        if n=1 then m else    m + total (n-1, 2 * m)
```

3. (1 punto)

$$T(n) = \begin{cases} a & \text{si } n=1 \\ 2T\left(\frac{n}{4}\right) + \lg n & \text{si } n>1 \end{cases}$$

➢ **Solución**

1. a) $T_{parcial}(m) = k_2 + T_{parcial}(m-1)$ => Resolviendo por ec. característica $(x-1)(x-1)=0 \Rightarrow r_1=1$ doble $\Rightarrow$

$T_{parcial}(m) = c_1+c_2\,m \Rightarrow$ $\mathbf{T_{parcial}(m) \in \Theta\,(m)}$

b) $T_{total}(n) = k_1 + T_{total}(n-1) + T_{parcial}(n-1) = k_1 + c_1+c_2(n-1) + T_{total}(n-1) = K_3+k_4n + T_{total}(n-1)$ =>

Resolviendo por ec. característica $(x-1)(x-1)^2=0 \Rightarrow r_1=1$ triple $\Rightarrow T_{total}(n)=c_1+c_2n+c_3n^2 \Rightarrow$

$\mathbf{T_{total}(n) \in \Theta\,(n^2)}$

2. El tamaño del argumento m no afecta si consideramos elemental la operación producto.

$$Ttotal(n) = \begin{cases} k_1 & \text{si } n=1 \\ k_2 + Ttotal(n-1) & \text{si } n>1 \end{cases}$$

Resolviendo por ec. característica $(x-1)(x-1)=0 \Rightarrow r_1=1$ doble $\Rightarrow t(n)=c_1+c_2n \Rightarrow \mathbf{T_{total}(n)} \in O(n)$

3. $T(n) = 2\,T(n/4)+\lg n$

- Si $n=4^k$ $[=(2^k)^2 \Rightarrow \sqrt{n}=2^k]$ podemos escribir la ecuación como:

  $T(4^k) = 2T(4^{k-1}) + 2k$
- Haciendo el cambio de variable $t_k = T(4^k)$, tenemos

  $t_k = 2\,t_{k-1} + 2k$ $[\,p(x)=(x-2)(x-1)^2 \Rightarrow r_1=2\ ;\ r_2=1$ doble $]$
- Esta es una recurrencia no homogénea, con solución

  $t_k = c_1 2^k + c_2 1^k + c_3 k 1^k$
- Deshacemos el cambio de variable:

  $T(n) = c_1\sqrt{n} + c_2 + c_3 \lg n \in \Theta(\sqrt{n})$.

Ejercicio_14. (Ex_Sept 16)

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En un instante T=1, 2, ... podemos ejecutar únicamente una tarea. La tarea i produce unos beneficios b_i ($b_i > 0$) sólo en el caso en el que sea ejecutada en un instante anterior o igual a d_i.

- Diseñar un algoritmo **voraz** para resolver el problema aunque no se garantice la solución óptima que nos permita seleccionar el conjunto de tareas a realizar de forma que nos aseguremos que tenemos el **mayor beneficio posible**.
- Detallar :

a. (1,5 puntos) Las estructuras y/o variables necesarias para representar la información del problema y el método voraz utilizado (El procedimiento o función que implemente el algoritmo). Es necesario marcar en el seudocódigo propuesto a que corresponde cada parte en el esquema general de un algoritmo voraz. Si hay más de un criterio posible elegir uno razonadamente y discutir los otros. Indicar razonadamente el orden de dicho algoritmo.

b. (0,5 puntos) Aplicar el algoritmo implementado en el apartado anterior a la siguiente instancia:

i	1	2	3	4
b_i	50	10	15	30
d_i	2	1	2	1

- Lista de candidatos C: el conjunto de las tareas.
- Lista de seleccionados S: las tareas seleccionadas hasta el momento.
- Función solución: cuando no queden más tarea factibles.
- Función Objetivo: Beneficios acumulados por las tareas en S.

$$Beneficio = \sum_{i \in S} \mathrm{b_i}$$

- Función de **factibilidad**: $(S \cup \{x\})$ es factible si todas las tareas $i \in (S \cup \{x\})$ pueden ser ejecutadas en un **instante anterior o igual a d_i**
- Función de selección: la mejor opción es coger las **tareas con mayor beneficio**.
    - o Para otras alternativas lógicas se pueden encontrar contraejemplos; Por ejemplo coger primero las que deban ejecutarse antes, contraejemplo:

i	1	2	3	4
b_i	10	5	50	**30**
d_i	2	2	3	**3**

Ésta ya no entraría, sin embargo es mejor que las dos primeras

- Algoritmo detallado:

  La parte complicada del ejercicio está en poder comprobar de manera eficiente si incluir una tarea genera **conflicto**. Para ello se puede mantener el vector solución ***S* ordenado según d_i** , tratando de insertar la nueva tarea en su posición correspondiente. Después es fácil comprobar si alguna de las tareas desplazadas o la propia tarea insertada se encuentra en una posición mayor a su correspondiente d_i , en cuyo caso **NO es factible** añadir la tarea.

- **Implementación.**
- Usamos una variable local "num_tareas" para acumular el **número de tareas** insertadas hasta este punto.
- Usamos una estructura "***datos***" con los valores $\boldsymbol{i}$, $\boldsymbol{b_i}$ y $\boldsymbol{d_i}$

```
función Selector_Actividades (C(1..n):datos;  var  S(1..m) :datos)
   // return conjunto_de_actividades (S(1..m), m ≤ n))
   OrdenarPorBeneficios(C,n)
   // C(1..n) ordenados según el criterio ( i < j ⇒ C[i].b ≥ C[j].b )
   S[1] ← C[1]
   num_tareas ← 1    // num_tareas representa  la cantidad de actividades seleccionada
   para i desde 1 hasta n hacer
         factible ← 1;
         K← num_tareas
         mientras k >1 && C[i].d < S[k − 1].d  && factible  hacer
               si S[k − 1].d > k entonces // mantener el vector solución S ordenado según d_i
                     factible ← 0;
               fsi
               K← k+1;
         fmientras
         posicion ← k;
         si  factible &&  C[i].d.≤ posicion entonces // comprobación factibilidad
         // Inserción
               para k desde num_tareas hasta posicion-1 dec 1 (k--) hacer
                     S[k]← S[k − 1];     //desplazamiento
               fpara
               S[posicion]← C[i]          // S ← S ∪ {i}
               num_tareas ← num_tareas + 1
         fsi
   fpara
   return S
ffunción
```

- **Instancia**

i	1	2	3	4
b_i	50	10	15	30
d_i	2	1	2	1

Ordenación C ⇒

i	1	4	3	2
b_i	50	30	15	10
d_i	2	1	2	1

- **Iteración 0**

S=

	1
i	1
b_i	50
d_i	2

num_tareas=1

- **Iteración 1**

S=

	1	2
i	4	1
b_i	30	50
d_i	1	2

num_tareas=2

- **Iteración 2 y 3:** Ningún otro elemento puede ser insertado. **S=solución óptima con Beneficio=80**