2.1 Un programa que utiliza la técnica divide y vencerás divide un problema de tamaño **n** en **a** subproblemas de tamaño **n/b**. El tiempo g(n) de la resolución directa (caso base) se considerará constante. El tiempo f(n) de combinar los resultados es $O(n^p)$; para simplificar consideraremos que $f(n) = d \cdot n^p$. Obtener el orden total de ejecución del algoritmo t(n), en función de los valores **a** y **b**.

2.2 Comprobar que los resultados anteriores son aplicables a cualquier valor de g(n) y para valores de f(n) de la forma $f(n) = a_p n^p + a_{p-1} n^{p-1} + ... + a_1 n + a_0$.

2.3 Usando la técnica divide y vencerás, resolver el problema de calcular la media de un array de **n** enteros, siendo n potencia de 2. Modificar el algoritmo para permitir valores de n que no sean potencia de 2. ¿Qué mejora se obtiene respecto a una simple versión iterativa? ¿Existe alguna situación en la que sea mejor la solución con divide y vencerás?

2.4 Supongamos que un problema puede resolverse de forma directa en un tiempo g(n) o bien, usando divide y vencerás, puede ser dividido en varios subproblemas de igual tamaño, necesitando en la división y en la combinación de resultados un tiempo f(n).

a) Demostrar que si $O(g(n)) \subseteq O(f(n))$ entonces la solución usando divide y vencerás es siempre peor o igual (en cuanto a orden de complejidad) que resolver el problema usando el método directo. ¿Qué significa lo anterior y qué consecuencias tiene en la construcción de algoritmos con divide y vencerás?
b) Demostrar que si $O(f(n)) \subseteq O(g(n))$, no siempre tiene que ser mejor la solución usando divide y vencerás. Sugerencia: la demostración se puede hacer dando un contraejemplo con valores de f(n) y g(n) adecuados.

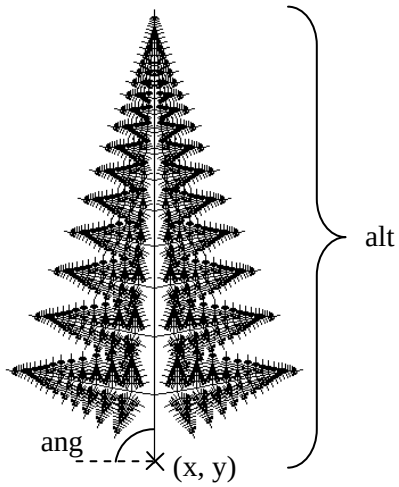
2.5 Un algoritmo divide y vencerás divide un problema de tamaño n en 2 subproblemas de tamaño n-1. El algoritmo aplica una solución directa cuando n = 2, que tarda un tiempo de 3 ms. Si el tiempo de realizar la combinación es f(n) = 1,2n ms, calcular el tiempo de ejecución exacto del algoritmo t(n).

2.6 Supongamos el problema de calcular todos los caminos mínimos en un grafo no dirigido. En general, ¿es posible descomponer fácilmente el problema en subproblemas, de forma que sea posible aplicar divide y vencerás? ¿En qué casos sí sería posible aplicar esta descomposición de forma sencilla? ¿Qué mejora se obtendría en tal caso?

2.7 Un programa para ordenar cadenas utiliza el método de ordenación por mezcla. La resolución para problemas de tamaño pequeño tarda un tiempo de $g(n) = 2n^2$, mientras que la mezcla requiere f(n) = 10n. Calcular cuál debería ser el tamaño del caso base para optimizar el tiempo de ejecución total t(n).

2.8 Diseñar un algoritmo para calcular el mayor y el segundo mayor elemento de un array de enteros, utilizando la técnica divide y vencerás. Calcular el número de comparaciones realizadas en el mejor y en el peor caso, suponiendo n potencia de 2. Comprobar si es posible aplicar los resultados obtenidos (en cuanto a órdenes de ejecución) a valores de n que no sean potencia de 2.

2.9 La figura de abajo representa el "helecho", una curva fractal que puede ser generada mediante una variedad de procedimientos. En su versión más simple, es posible utilizar un procedimiento de divide y vencerás para dibujarlo. Suponer que la cabecera del procedimiento es de la forma **Pinta_Helecho(x, y: integer; ang, alt: real)**, que pinta un helecho de altura **alt**, cuyo tallo principal tiene la base en **(x, y)** y tiene ángulo **ang** respecto a la horizontal. Escribir un procedimiento para dibujar el helecho, a partir de los parámetros anteriores, utilizando la técnica divide y vencerás.



2.10 En un algoritmo divide y vencerás, en lugar de aplicar el caso base cuando **n** es menor que cierto valor, se aplica la recurrencia siempre un número **r** de veces. Calcular cuál será el tiempo de ejecución, suponiendo que un problema de tamaño **n** es dividido en **a** problemas de tamaño **n/b**, siendo el tiempo del caso base $g(n) = n^q$, y el tiempo de la mezcla $f(n) = n^p$.

2.11 Sea T[1..n] un array ordenado formado por enteros diferentes, algunos de los cuales pueden ser negativos. Dar un algoritmo que pueda hallar un índice **i** tal que $1 \le i \le n$ y $T[i]=i$, siempre que este índice exista. El algoritmo debe estar en un tiempo de ejecución O(log n) en el peor caso. ¿En qué tipo de algoritmos lo clasificarías?

2.12 Dado un array de enteros E[1..N], se dice que un entero es un elemento mayoritario de E si aparece más de n/2 veces en E. Dar un algoritmo para calcular si un array E[1..N] contiene un elemento mayoritario. El algoritmo tiene que funcionar en tiempo lineal en el peor caso.

2.13 (TG 9.2) Considerar la aplicación de la técnica divide y vencerás sobre el siguiente problema. Dado un array de **N** números enteros, buscamos la cadena de **n** celdas consecutivas en este array cuya suma sea máxima (obviamente n<N). ¿Es conveniente aplicar divide y vencerás en este caso? ¿Cómo sería una resolución directa del problema?

2.14 Calcular el orden exacto Θ del número promedio de comparaciones que se hacen en la ordenación por mezcla dentro de la función de mezcla.

2.15 (TG 9.4) Construir un algoritmo para calcular el producto de dos matrices cuadradas triangulares superiores, siendo el número de filas y columnas potencia

de 2. Se supondrá que disponemos de un procedimiento **Multip1(A, B, C)**, para multiplicar matrices (triangulares o no) en un tiempo $O(n^3)$. Obtener el tiempo de ejecución del algoritmo creado.
Recordatorio: una matriz se dice que es triangular superior si todos los elementos que están por debajo de la diagonal principal valen 0. Se dice que es triangular inferior si todos los elementos encima de la diagonal principal valen 0. Ejemplo de matriz triangular superior:

1	3	5
0	2	4
0	0	2

2.16 Comentar cómo se implementaría con un método divide y vencerás similar al de Strassen la multiplicación de una matriz cuadrada triangular por una cuadrada no triangular. Habrá que indicar qué funciones sería necesario implementar y qué habría que hacer para ahorrar memoria en las llamadas recursivas. Estudiar también el tiempo de ejecución.

2.17 (TG 9.7) Se trata de resolver el problema de encontrar el cuadrado de unos más grande en una tabla cuadrada de bits (un array n x n).
a) Programar un método directo para resolver el problema y dar una cota superior (no demasiado mala) de su tiempo de ejecución.
b) Programar un método para resolver el problema por divide y vencerás y dar una cota superior (no demasiado mala) de su tiempo de ejecución.

2.18 (TG 9.8) Programar un método de ordenación de un array similar a la ordenación por mezcla, pero dividiendo el problema en 3 subproblemas (de tamaños parecidos) en lugar de en 2.