***Inteligencia Artificial: Práctica 3***

***Ejercicio 1:***

En el primer ejercicio se nos pedía implementar una función que duplicase cada elemento de una lista, es decir, dada la función **duplica(L, L1),** que L1 tenga lls elementos de L repetidos.

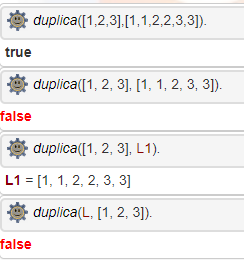
Para ello hemos implementado un caso base y una función principal:

* *Caso base :* las dos listas están vacías pues ya no hay nada más que duplicar.



* *Función principal :* pasamos como argumentos las listas separadas por primer elemento y lista, de modo que L contiene un X y L1 lo contiene dos veces pues lo queremos repetido. Esta función se llama recursivamente con el resto de las dos listas para que realice lo mismo hasta que llegue al caso base.



******Los siguientes ejemplos muestran su funcionamiento, pasándole la segunda lista y devolviendo True si está bien, o False si está mal, y devolviendo también la segunda lista correctamente duplicada. El último ejemplo nos devuelve False puesto que la segunda lista es una lista que no tiene elementos duplicados, solamente aparecen una vez, luego no corresponde a ningún resultado posible.

***Ejercicio 2:***

A continuación implementamos una función que invierte una lista. Para ello hacemos uso de la función *concatena*, la cual nos dan y se basa en concatenar los dos primeros argumentos en el último.



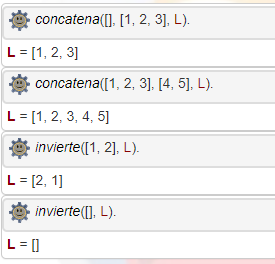
Para implementar la función **invierte(L, L1)**, donde L1 es L del revés, realizamos dos casos:

* *Caso base :* ambas listas están vacías luego no hay elementos que invertir y son iguales.



* *Función principal :* la primera lista se separa en un primer elemento H y el resto T, de modo que llamamos recursivamente a la función con el resto T para ir dándole la vuelta a la lista, y concatenando los resultados en la lista L1 (segundo parámetro).



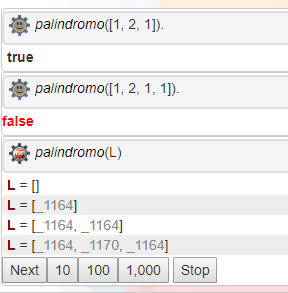


Estos ejemplos muestran que la función concateba funciona pasándole una lista vacía o no, devolviendo el resultado en el tercer argumento, y por otro lado, nos invierte la lista [1, 2]. También comprobamos que si invertimos una lista vacía el resultado es una lista vacía (pues es su inversa).

***Ejercicio 3:***

El ejercicio 3 nos pedía la función **palindromo(L)** en la que tenemos que devolver True si L es palíndromo, es decir, se lee igual de derecha a izquierda que de izquierda a derecha; y devuelve False si no es un palíndromo.

Para ello, hacemos uso de la función **invierte** del ejercicio 2 y simplemente comprobamos si *invierte(L, L)* nos devuelve True o False, es decir, si al darle la vuelta a L nos queda exactamente la misma palabra.



Los dos primeros ejemplos muestran una lista que si cumple la función y otra que no. Además, podemos observar que al introducir *invierte(L)* se queda en un bucle infinito buscando todas las combinaciones posibles que cumplan la función, empezando por un elemento, dos elementos… y así sucesivamente. No aparecen números sino zonas de memoria, pues no importa qué contengan, sino que sean un palíndromo.

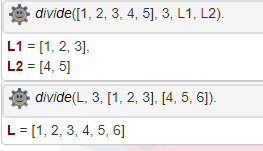
***Ejercicio 4:***

En este ejercicio implementamos la función **divide(L, N, L1, L2)**, la cual se satisface cuando L1 contiene los N primeros elementos de L, mientras que L2 contiene los restantes.

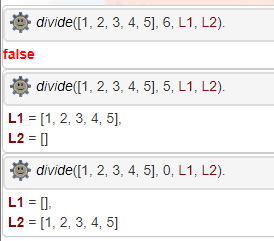
Para implementarlo hemos utilizado la función **concatena** del ejercicio 2. Básicamente esta función consiste en que L1 tenga una longitud N y que la concatenación de L1 y L2 sea L, y así se forma la única combinación posible.



Hemos probado los ejemplos que se pedían en el enunciado, dando un resultado correcto:



Además, hemos implementado otros ejemplos para comprobar su correcto funcionamiento. En el primero comprobamos que nos da False cuando introducimos una N mayor que la longitud de L, pues no podemos introducir elementos de más en L1. En el segundo comprobamos que L2 sale vacía cuando L1 tiene la misma longitud que L; y finalmente, si introducimos una N igual a 0, L1 estará vacía y L2 será igual que L.



***Ejercicio 5:***

El ejercicio 5 nos pedía realizar la función **aplasta(L, L1)** la cual “aplasta” la lista L en la lista L1, es decir, L es una lista de listas de números, esta función la convierte en solamente una lista. Para su implementación lo hemos dividido en tres casos:

* *Caso base :* dos listas vacías pues es trivial ver que una es la aplastada de la otra.



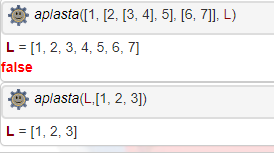
* *Caso 1 : el primer elemento de L es una lista.* En este caso, aplastamos ese primer elemento con una lista L2, aplastamos el resto de L con otra lista L3 y luego concatenamos tanto L2 como L3 en L1, pues esas listas corresponden a los elementos sueltos sin lista.



* *Caso 2 : el primer elemento de L no es una lista.* En este otro caso, solamente aplastamos el resto de la lista, es decir H, pues el primer elemento ya se encuentra en la lista a devolver.



Hemos implementado los ejemplos del enunciado para comprobar su correcto funcionamiento. Podemos observar que al introducir *aplasta(L, [1,2,3])* nos devuelve la misma lista, pues su origen puede estar aplastado o no.



***Ejercicio 6:***

En este ejercicio se nos pide implementar una función **primos(N, L)**, en la que N es un número natural, y L es una lista que contiene sus divisores. Para implementarla hemos creado tres funciones:

1. **NEXT\_FACTOR(N, F, NF)** : esta función recibe el natural N, y dado un factor divisor F, calcula el siguiente NF, siendo la primera iteración con un 2 y la siguiente con 3, 5… y los números impares.



1. **PRIMOS(N, L)** : es la función principal, simplemente comprueba que N es mayor que 0 y llama a la función divisores, que es la encargada de la recursión, comenzando por probar el primo más pequeño, el 2.



1. **DIVISORES(N, L, F)**: es la función recursiva en la que N es el factor a dividir, L contiene los divisores y F es el factor que probamos como divisor o no. Lo hemos dividido en 3 casos:

* *Caso base :* cuando N es 1 hemos llegado al final, pues no podemos seguir dividiendo el 1 en más primos.



* *Caso 1:* F divide a N, es decir, N es 0 en módulo F. En este caso, dividimos F por F, metemos F en la lista de divisores y llamamos recursivamente a divisores con N/F y **sin cambiar el factor F**, pues puede tener multiplicidad mayor que 1.

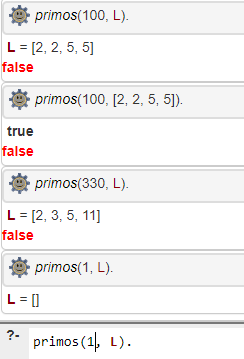


* *Caso 2:* en este caso F no divide a N, por lo que no cambiamos N ni metemos F en la lista de los divisores, solamente calculamos el siguiente factor con next\_factor y llamamos a divisores con este nuevo factor NF.

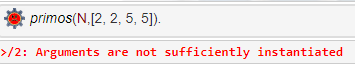


Hemos implementado los siguientes ejemplos:

En el primero y tercero comprobamos que los divisores de 100 y 330 salen bien respectivamente como lista, en el segundo comprobamos que si pasamos los divisores la función nos devuelve true, y finalmente, comprobamos que al pasarle un 1 nos sale una lista vacía pues no tiene divisores primos.



Además, al realizar la función a la inversa nos dice que no hemos introducido suficientes parámetros de entrada. Esto se debe a que esos divisores pueden corresponder a más números, no solamente a uno; luego esta función solamente funciona en un sentido.



***Ejercicio 7:***

En este ejercicio se nos pedía realizar 3 funciones:

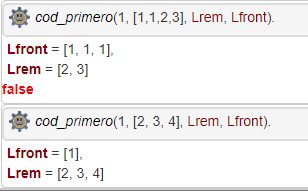
* **COD\_PRIMERO(X, L, Lrem, Lfront)** : esta función consiste en dividir L en dos listas, en Lfront guardamos X y los elementos que sean iguales que X, mientras que en Lrem guardamos el resto de L.

Lo hemos implementado de la siguiente forma:

* *Caso base* : L y Lrem están vacías y Lfront solamente contiene X

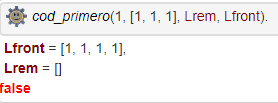


* *Caso 1* : el elemento coincide con el primer elemento de L, de modo que lo metemos en Lfront y llamamos recursivamente a la función con el resto de L, al que hemos llamado F.
* *Caso 2* : el elemento X no coincide con el primer elemento de L, esto quiere decir que ya hemos copiado en Lfront todos los elementos de L que eran iguales que X, de modo que solamente queda el resto de L, es decir, solo queda Lrem.

Hemos probado los siguientes ejemplos:

En este comprobamos que, si X está en L, guarda tanto el elemento como los repetidos en Lfront.

Éste comprueba que Lfront guarda el elemento, aunque no esté presente en la lista.



Por último, este ejemplo muestra que si L solamente contiene un único elemento repetido, Lrem sale vacía.

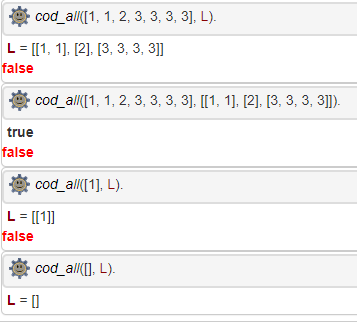
* **COD\_ALL(L, L1) :** este apartado pedía implementar una función que utilizase cod\_primero, de tal modo que L (una lista formada por elementos repetidos o no, de modo que los repetidos aparecen juntos), es dividida en sublistas formadas por los elementos repetidos. L1 es una lista que contiene esas sublistas.
* *Caso base :* L1 y L están vacías, por lo que no quedan más elementos.



* *Función principal :* llamamos a la función cod\_primero con el primer elemento de L, de modo que Lfront sea la primera sublista de elementos repetidos. Recursivamente vamos llamando a cod\_all con Lrem, es decir, con lo que ha “sobrado” de la función para que realice lo mismo con el resto de elementos.



Hemos probado los siguientes ejemplos:

En este observamos que la lista devuelta es correcta y completa.

Este ejemplo nos muestra que al pasar el resultado nos devuelve True, es decir, corresponde a esa lista.

Este ejemplo lo prueba con solamente un elemento.

Finalmente, este ejemplo prueba que si pasamos una lista vacía, el resultado será una lista vacía pues no hay elementos repetidos

* **RUN\_LENGHT(L, L1) :** este es el último apartado, en el que utilizamos las funciones anteriores para resolver el problema completo. Éste consiste en que, pasada una lista L de números repetidos como en los apartados anteriores, nos devuelve en L1, una lista de sublistas en las que cada una se compone del número de repeticiones del elemento, y el elemento que se repite.
* *Caso base:* las dos listas están vacías, es decir, hemos llegado al final del problema.



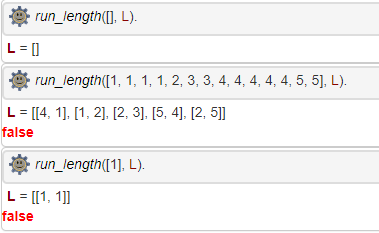
* *Función auxiliar:* hemos implementado una función auxiliar para la resolución del problema, ésta consiste en dividir la lista que devuelve cod\_all en la lista que necesitamos. Básicamente, por cada elemento de la lista que nos devuelve cod\_all, calcula su longitud y la une con el primer elemento de esa lista.



* *Función principal:* simplemente llama a cod\_all y comienza la recursión para dividir la lista resultada mediante la función auxiliar.



Hemos implementado los siguientes ejemplos:

Éste nos muestra que una lista vacía se concierte en otra vacía.

Este ejemplo prueba que funciona correctamente.

Este ejemplo es una prueba con un solo elemento.

***Ejercicio 8***

A continuación, se pedía desarrollar la función *build\_tree(L, T)* que dada la lista L de elementos de tipo *letra-número* se desarrolla el árbol T siendo *tree(inf, izq, der)* siendo izq y der los hijos de la izquierda y de la derecha respectivamente. Para su desarrollo lo hemos dividido en dos casos:

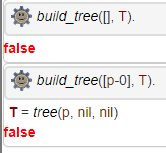
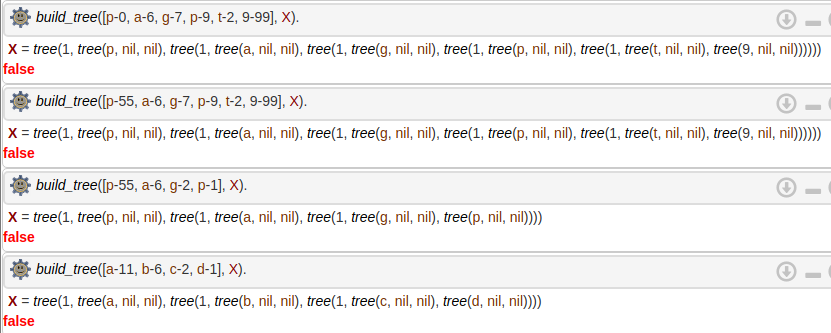
* *Caso Base:* cuando solamente queda un elemento en la lista, el árbol tendrá como información dicho elemento y no tendrá hijos (nodo hoja).

***https://lh6.googleusercontent.com/OYy-i2UsehygLOgfxTLVtWeQtx1pFp3qcc7xl_RaM-5-tE-r_B23upBrACwDT_x9-G4W8dcj0yNspe-LQGkPkNkP6Fbm2DkILh_Tz6pagC-3s3kXym6AescIq2dpaNEISWLC-o8b***

* *Función principal*: coge el primer elemento de la lista y lo inserta como hijo izquierdo del árbol, llamando recursivamente a la función con el resto de la lista para realizar lo mismo hasta que quede un elemento (caso base).

***https://lh6.googleusercontent.com/DgIfHx2bz96tqDTIBeVw96smXigEcUPAWZVcaeSmKrjecuYJxj73EkkIqro0ySlty6OBu7bA0SX0rYaWMVnqE6Jlax2aAaiS0eN8mxdDzZVy2S9qoKzATst-oE2RcLb1nj8d-NcT***

Para su comprobación hemos probado los ejemplos propuestos en el enunciado, dándonos los mismos resultados y construyendo el árbol de izquierda a derecha.

******

También hemos probado el caso en el que la lista está vacía y cuando solamente tiene un elemento, siendo lógica su salida.

***Ejercicio 8.1***

Este apartado nos pedía codificar los elementos para el árbol de Huffman, para ello hemos implementado la función *encode\_elem(x1, x2, Tree)* la cual codifica el valor de x1 en x2 según la información contenida en Tree, de tal modo que nunca se repitan los prefijos, el más a la izquierda sea un 0 y según nos movamos a la derecha vayamos introduciendo 1 o 0. Para ello tenemos tres casos:

* *Caso Base:* el elemento x1 se encuentra a la izquierda en el árbol por lo que insertamos un 0 en x2.

***https://lh3.googleusercontent.com/pax75NRsp4X2Qfh63aZ0Px0VODVJnyCIU2fJWkjorAsB5T_2oqTTA-NgpLVRSVtyiEVPis94YJytnAYL9wGrAZsG9jjWdYf2ia_Il2ataU0gwyioCxbTommUu3Pg-efnGqI6ZLFE***

* *Otro Caso Base:* ahora el elemento x1 es el nodo de la derecha, entonces concatenamos un 1 en x2.

***https://lh3.googleusercontent.com/jF9rkRwZ2FoDrX0nhzRUWRFA1xAN8T1q-Ak3hfU7Z62pkpHjXpyrbb4-DVE6a19NZVNJ-gkkSnvlt7pNxgC63lAmjbAvCMXXCd8gmkrW5a-LM1h8zkXDt1LbqotMaYBToQy-SlaA***

* *Función Principal:* el árbol tiene como hijo derecho siempre un subárbol, hasta llegar al último, por tanto, llamamos recursivamente a la función con el elemento x1, con el subárbol y con una lista auxiliar que vaya concatenando los 1 y 0, de modo que la función busca hasta que encuentra x1.

***https://lh3.googleusercontent.com/s0w694sW-QNutyGo3q1ZS7DslTsG6bvf2IfbzsJnvAJtqDJnVgfDb2LuYXCi-qdHmimQ84ndSkYOEZfMAz9UA3TUFNNKrxqGG9wn3gOwfrXHd2O1SKkx49aN17cz0DHPqFiOfKOM***

Hemos ejecutado los ejemplos propuestos en el enunciado dándonos un resultado correcto:

* El primer ejemplo busca la ‘a’ que es el primer elemento del árbol, luego le corresponderá un 0.
* Los siguientes son similares, buscan una ‘b’, una ‘c’ y una ‘d’ que corresponden al segundo, al tercer y al cuarto elemento del árbol, luego su salida es correcta.

******

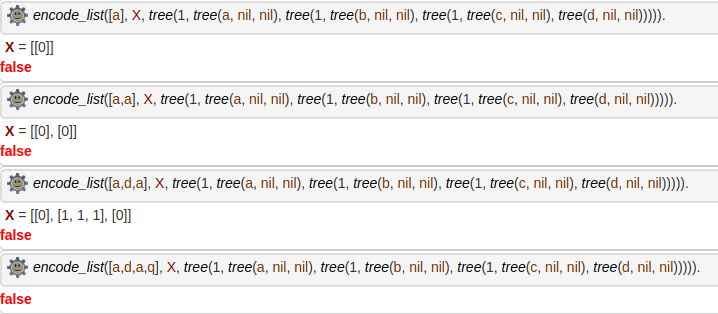
Por último, si intentamos buscar un elemento que no se encuentra en el árbol devuelve False.

***Ejercicio 8.2***

Este apartado es similar al anterior, solo que ahora hay que codificar una lista de elementos en otra lista de elementos. La función es *encode\_list(L1, L2, Tree)* siendo L1 la lista de elementos a codificar, L2 la lista que guarda la codificación y Tree el árbol donde buscan la información. Tenemos dos casos:

* ***https://lh3.googleusercontent.com/4Ia8rgesKSvdRRZepahZeobqfyKT-kiauX5dfoDBSmbtGK73-1vVAT6u4ycuGV3xmHzbz_dVAA52nqNRGCvzkdYI4WNlmZ73yEgqS6fXsIlg3Ph-izLE4dNszZK7Lx-aRoQ4jrDW****Caso Base:* ambas listas están vacías luego no hay más elementos que codificar.
* ***https://lh4.googleusercontent.com/CDGy03K1UVWxMER3-5OLHRJBrR04USg4EpQitvIhGrByYf3leYIdEIxluqo3mJ2Bm9mp5PqWu1DwwWUZmDlTMtQtlJNTIXF-vicVLCojMqdrsKCE7gf087wZ53mqN_3498wk-NTe****Función Principal:* tomamos los primeros elementos de las dos listas y llamamos a la función *encode\_elem* para que codifique el primer elemento de L1 en el primer elemento de L2, y así llamamos recursivamente a *encode\_list* con el resto de elementos hasta que las listas estén vacías.

Estos son los ejemplos probados que recomienda el enunciado de la práctica:

******

***Ejercicio 8.3***

Finalmente, tenemos que implementar una función que dada una lista de palabras, salga codificada. Para ello hemos implementado ciertas funciones auxiliares:

* 1. *Ordenar(L1, L2):* ordena la lista L1 en la lista L2 de mayor a menor. Tiene un caso base con ambas listas vacías y un caso principal que llama a la función *mayor* con el primer elemento de L1 para que lo guarde de mayor a menor y, a su vez, llama recursivamente con el resto de L1 a sí misma para ordenar el resto de la lista.





También podría llamar a una función *menor* que los ordenase que menor a mayor, pero ahora explicaremos por qué la hemos elegido así.

* 1. *Mayor(P, L1, L2):* ésta es una función auxiliar para ordenar los elementos de mayor a menor, de modo que vamos introduciendo a cada elemento en su posición adecuada. Si la lista L1 está vacía simplemente metemos P en L2, sino los concatenamos, y sino los vamos comprobando e introduciendo de mayor a menor.





* 1. *Decodifica(L1, L2):* esta función decodifica la función L formada por pares [Número, Letra], en la función L2 compuesta por elementos Letra-Número. Para ello lo dividimos en un caso base donde ambas listas están vacías y un caso principal que decodifica el primer elemento de la lista y llama recursivamente a la función hasta que estén ambas vacías.





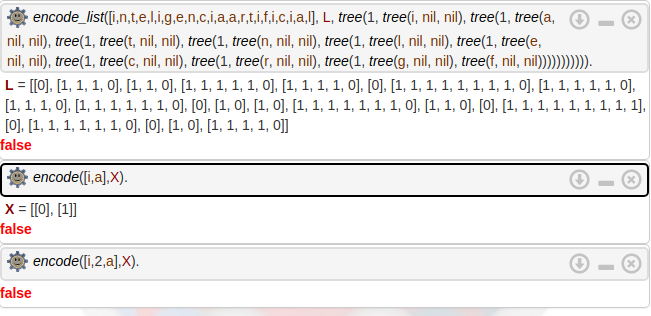
* 1. *Pertenece(L, D):* se satisface cuando todos los elementos de L pertenecen a D, para ello llamamos a la función *member* con cada elemento de L hasta que esté vacía.





Finalmente, desarrollamos la función *encode(L1, L2)* siguiendo los siguientes pasos:

* En primer lugar, asociamos a la variable Dic el diccionario propuesto en el enunciado y comprobamos con la función *pertenece* que todos los elementos de L1 están en Dic.
* A continuación, ordenamos L1 alfabéticamente para que todas las letras que sean iguales estén juntas, de manera que si L1 era [a, b, a, b, a] nos quede [b, b, a, a, a] de mayor a menor, ahora veremos por qué.
* Ahora que tenemos la lista L1 ordenada podemos llamar a *run\_length,* implementada en el ejercicio 7, para que nos cuente cuántas veces aparece cada letra. En el ejemplo anterior nos quedaría [[2, b], [3, a]].
* Una vez tenemos cuántas veces aparece una letra, ordenamos de nuevo la lista de mayor a menor, pues para construir el árbol, necesitamos que las letras más frecuentes vayan primero. Nos quedaría [[3, a], [2, b]].
* Luego llamamos a la función *decodifica* para que nos transforme una lista de pares [Número, Letra] a una lista de elementos Letra-Número. Por ejemplo, nos quedaría [a-3, b-2].
* Por último, construimos un árbol con la lista decodificada y llamamos a *encode\_list* con el árbol generado para que nos codifique la lista que le hemos pasado en la lista L2.

******Hemos probado los ejemplos del enunciado y los codifica correctamente. Además, cuando metemos un elemento que no esté en el diccionario devuelve False, como podemos observar en el último ejemplo.