Introducción a la Programación

Segundo Recuperatorio Segundo Parcial

- El examen se aprueba con 6 puntos
- Utilizar este archivo fuente de base para la programación. Ya cuenta con los def y las signaturas correctas.
- Para testear el código pueden usar este archivo que ya cuenta con todo lo necesario para desarrollar sus propios tests (este archivo no se entrega)
- **Para aprobar el parcial es requisito indispensable que todos los programas pasen los tests del archivo del punto anterior.**
- **Todo el parcial se puede resolver con las herramientas básicas de imperativo vistas en clase. No está permitido el uso de listas por comprensión, funciones de librerías externas o funciones nativas no vistas en clase (por ejemplo: enumerate, zip, count, remove, index, reversed, etc)**

1) Caminen, chiques, caminen! [2 puntos]

Lita de Lazari fue una conocida ama de casa de la década de los 80's y 90's.
Fue, durante muchos años, la presidenta de la Liga de Amas de Casa.
Su fama se debía, principalmente, a que salía por televisión dando consejos a las amas de casa.
Entre sus frases más famosas está la ya clásica "caminen chicas" (parafraseada y actualizada a los tiempos modernos en el título de este ejercicio).
Esta frase representaba la idea de que, dada la situación económica del país en aquella época (no muy diferente a la actual) la mejor forma de ahorrar era recorrer diferentes comercios en busca de los mejores precios.

Implementar la función mejores_precios() que dadas dos listas super1 y super2, de igual longitud, donde cada i-ésimo elemento de ambas listas representa el precio de un mismo producto en dos supermercados, devuelva una lista de igual longitud con el menor precio de cada producto.

problema mejores_precios (in *super1*: seq⟨String x R⟩, in *super2*: seq⟨String x R⟩): seq⟨String x R ⟩ {
  requiere: {$|super1| = |super2|$}
  requiere: {Todos los elementos en las segundas posiciones de las tuplas de *super1* y de *super2* son positivos}
  requiere: {Todos los elementos en las primeras posiciones de las tuplas de *super1* y de *super2* son iguales}
  asegura: {$|res| = |super1|$ }
  asegura: {Cada posición de $|res|$ contiene una tupla con el nombre del producto correspondiente al de esa posición en *super1* y el mínimo valor entre los elementos que se encuentran en esa posición en *super1* y *super2*}
}

Por ejemplo, dado
super1 = [("leche", 151.0), ("yerba", 4719.5), ("jabón", 269.2)]
super2 = [("leche", 261.2), ("yerba", 3939.1), ("jabón", 319.2)]
se debería devolver *res* = [("leche", 151.0), ("yerba", 3939.1), ("jabón", 269.2)]

2) Seguidilla [2 puntos]

Implementar la función *seguidilla()* que dada una secuencia de enteros *calificaciones*, y un entero *nota_minima*, devuelva la cantidad de elementos de la subsecuencia más larga que cumplen que son mayores o iguales a la nota_minima.
En caso de que esta seguidilla no exista, devolver 0.

problema seguidilla (in *calificaciones*: seq⟨Z⟩, in *nota_minima*: Z): Z {
  requiere: {todos los elementos de *calificaciones* son mayores o iguales a 0 y menores o iguales a 100}
  requiere: {*nota_minima* es mayor o igual a 0 y menor o igual a 100}
  asegura: {*res* = |subsec| si solo si existe una subsecuencia de *calificaciones* (*subsec*), y todos los elementos de *subsec* son mayores o iguales a la *nota_minima*}
  asegura: {No existe otra subsecuencia de calificaciones que tenga longitud mayor a res}
  asegura: {*res* = 0 si y solo si no hay ningún elemento de *calificaciones* que sea mayor a *nota_minima*}
}

Ejemplo 1: dada los siguientes inputs:

calificaciones = [10,55,60,87,54,98,87,65,55,45,57]; *nota_minima* = 60
se debería devolver *res* = 3, que es la longitud de la subsecuencia [98,87,65]

Ejemplo 2: dada los siguientes inputs:

calificaciones = [10,55,60,65,54,64,65,55,45,57]; *nota_minima* = 70
se debería devolver *res* = 0, ya que no hay ninguna subsecuencia de calificaciones con elementos mayores o iguales a nota_minima

3) Posiciones pares [3 puntos]

Implementar la función *elem_en_pos_pares()* que dada una lista de listas *matriz* y un elemento *elem* devuelva una lista de *bool* de igual longitud de *matriz*, que indique en cada posición si *elem* se encuentra en alguna posición par de la sublista de *matriz* que ocupa esa posición.

problema elem_en_pos_pares(in *matriz*:seq⟨seq⟨Z⟩⟩, in *elem*:Z) : seq⟨Bool⟩ {
  asegura: {(|*res*| = |*matriz*|) }
  asegura: {Cada i-ésima posición de *res* indica si *elem* pertenece a la lista matriz[i] en una posición par}
}

Por ejemplo, dados:
elem= 1; *M* = [
[1, 2, 3, 4, 5, 6, 7, 8, 9],
[9, 8, 7, 6, 4, 5, 3, 2, 1],
[0, 0, 0, 0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 4, 0, 0, 0],
[0, 1, 0, 0, 6, 0, 0, 1, 0],
]
se debería devolver *res* = [true, true, true,false,false]

4) Molinete [3 puntos]

La forma de pago del transporte público varía ampliamente entre ciudades.
En muchas ciudades del mundo (Ej. París) existen alternativas de contratación del servicio con una tarifa plana.
Esto implica que pagando un monto fijo por mes (o por otros periodos de tiempo) es posible utilizar libremente el transporte, tantas veces como sea requerido.
Para esto, se carga la tarifa en una tarjeta con tecnología NFC (como la SUBE) y al subir a cada transporte se pasa la tarjeta por el sensor para verificar que se tenga contratado el servicio.

En el presente ejercicio vamos a trabajar con un diccionario (*viajes_diarios*) que registrará los usurarios que subieron a algún transporte cada día del mes.
El diccionario tendrá como clave el número de cada uno de los días del mes, y como valores, el registro de usuarios que pasaron su tarjeta por algún transporte público en el día correspondiente a la clave.

Implementar la función *dias_viajados()* que dado un diccionario *viajes_diarios* y una lista *usurarios* devuelva un nuevo diccionario, con los elementos de *usuarios* como claves y como valor la cantidad de días en que el usuario tomó algún transporte.

problema viajes_por_dia(in *viajes_diarios*: dict⟨Z,seq⟨String⟩, in *usuarios*:⟨String⟩): dict⟨String, Z⟩ {
  requiere: {Las claves de *viajes_diarios* están entre 1 y 31}
  requiere: {La secuencia de *usuarios* no tiene elementos repetidos}
  requiere: {Cada valor de *viajes_diarios* es una secuencia de elementos de *usuarios*}
  asegura: {*res* tiene como claves a todos los elementos de *usuarios*}
  asegura: {Cada valor de *res* representa en cuántos valores de *viajes_diarios* aparece el usuario correspondiente}
}

Por ejemplo, dado el siguiente diccionario y lista de usuarios:

viajes_diarios = {1 : ["Juan", "Maria"], 2 : ["Marcela","Juan"]}
usuarios = ["Juan", "Maria", "Marcela"]

resultado_esperado es:
{"Juan" : 2, "Maria" : 1, "Marcela": 1}