

KARRRS!

Cachaaawww! Rocha McQueen es el auto más rápido de toda Radiador Springs y está preparándose junto a sus amigos para competir este año. Para eso nos pidieron que los ayudemos con un programa en Haskell para poder ganar.



Sabemos que las carreras tienen un número de vueltas, la longitud de la pista, los participantes y los nombres del público que la está viendo.

De los participantes sabemos que son autos (duh) y que tienen un nombre, un nivel de nafta, una velocidad, el nombre de su enamorad@ y un truco en particular, que realizan a lo largo de la carrera.

- Los trucos modifican la nafta o la velocidad del auto:
 - **deReversaRocha** que hace que suba la nafta 5 veces por la cantidad de metros de la pista.
 - **impresionar**, si en el público está el/la enamorad@ de un auto, el mismo corre el doble de rápido para impresionarlo ;)
 - **nitro** aumenta su velocidad en 15 km instantáneamente.
 - **comboLoco** que es realizar deReversaRocha con nitro.

Algunos autos son:

- **RochaMcQueen** que tiene 282 litros, su enamorado es Ronco y su truco es deReversaRocha.
- **Biankerr** (nuestro tanque ruso) tiene 378 litros, su enamorado es Tincho y su truco es impresionar.
- **Gushtav** tiene 230 litros, su enamorada es Peti y su truco es nitro.
- **Rodra** tiene 153 litros, su enamorada es Tais y su truco es combo loco.



Volvamos a las carreras: todos los participantes inician con una velocidad en 0 y la misma culmina al completar todas las vueltas.

Al transcurrir cada vuelta, suceden tres cosas:

- 1) Se restará del combustible del auto la cantidad equivalente a la longitud de su nombre x la cantidad de kilómetros de la vuelta.
- 2) Se incrementa la velocidad también según la longitud de su nombre:
 - Si tiene entre 1 y 5 letras aumenta 15 km/h.
 - Si tiene entre 6 y 8 aumenta 20 km/h.
 - Si tiene más de 8 aumenta 30 km/h.

3) Quien tenga la velocidad más baja (último lugar) utilizará su truco para intentar remontar.
Se pide:

- 1) Modelar las carreras y los 4 autos mencionados más arriba.
- 2) Modelar los trucos.
- 3) Hacer la función *darVuelta*, que me devuelve como queda la carrera después de dar una vuelta en la misma.
- 4) Hacer la función *correrCarrera* que implica que los participantes corran todas las vueltas de la misma.
- 5) Obtener el ganador de una carrera.
- 6) Los autos pueden competir nuevamente luego de una carrera si la terminan con más de 27 litros (sí, nos gustan los números arbitrarios) o si son el ganador.
Modelar la función *recompetidores* que me permite saber cuales son los autos que pueden seguir compitiendo después de una carrera.
- 7) Luego tenemos la carrera ultra suprema de las altas ligas que tiene una cantidad infinita de participantes!
De ella queremos saber:
 - a) ¿Podemos correrla?
 - b) ¿Podemos conocer el primer participante luego de 2 vueltas?
 - c) ¿Podemos dar la primera vuelta de la carrera?

Justifique cada respuesta.



No olvidar tipar las funciones, dar ejemplos de uso, ser expresivo y declarativo.

Solo pueden usar recursividad una vez a lo largo de todo el parcial, no es obligatorio hacerlo. Choose wisely.

```
import Text.Show.Functions
```

```
data Auto = Auto {    nombre:: Nombre,
                    nafta:: Nafta,
                    velocidad:: Velocidad,
                    enamoradx:: Enamoradx,
                    truco:: Truco} deriving Show
```

```
type Nombre = String
type Nafta = Int
type Velocidad = Int
type Enamoradx = String
type Truco = Carrera -> Auto -> Auto
type Distancia = Int
```

```
----- Funciones Auxiliares -----
```

```
mapNafta:: (Nafta -> Nafta) -> Auto -> Auto
mapNafta f unAuto = unAuto {nafta = f.nafta $ unAuto}
```

```
mapVelocidad:: (Velocidad -> Velocidad) -> Auto -> Auto
mapVelocidad f unAuto = unAuto {velocidad = f.velocidad $ unAuto}
```

```
mapParticipantes:: (Participantes -> Participantes) -> Carrera -> Carrera
mapParticipantes f unaCarrera = unaCarrera {participantes = f.participantes $ unaCarrera}
```

```
minimumBy:: (Ord b) => (a -> b) -> [a] -> a
minimumBy = foldl1.menorSegun
```

```
compararSegun:: (Ord b) => (b -> b -> Bool) -> (a -> b) -> a -> a -> a
compararSegun comp f a b
    | comp (f a) (f b) = a
    | otherwise = b
```

```
menorSegun:: (Ord b) => (a -> b) -> a -> a -> a
menorSegun = compararSegun (<)
```

```
maximumBy:: (Ord b) => (a -> b) -> [a] -> a
maximumBy = foldl1.mayorSegun
```

```
mayorSegun:: (Ord b) => (a -> b) -> a -> a -> a
mayorSegun = compararSegun (>)
```

```
applyIf:: (a -> Bool) -> (a -> a) -> a -> a
applyIf criterio funcion elemento
    | criterio elemento = funcion elemento
```

| otherwise = elemento

```
mapIf:: (a -> Bool) -> (a -> a) -> [a] -> [a]
mapIf = (.) map.applyIf
```

----- Funciones Auxiliares -----

-- Punto 1

```
rochaMcQueen:: Auto
rochaMcQueen = Auto "rochaMcQueen" 82 0 "Ronco" deReversaRocha
```

```
biankerr:: Auto
biankerr = Auto "biankerr" 78 0 "Tincho" impresionar
```

```
gushtav:: Auto
gushtav = Auto "gushtav" 100 0 "Ronco" nitro
```

```
rodra:: Auto
rodra = Auto "rodra" 53 0 "Ronco" comboLoco
```

-- Punto 2

```
deReversaRocha:: Truco
deReversaRocha = mapNafta.(+).(*5).longitudPista
```

```
enamoradoxEstaEnElPublico:: Carrera -> Auto -> Bool
enamoradoxEstaEnElPublico unaCarrera unAuto = elem (enamoradox unAuto).publico $
unaCarrera
```

```
impresionar:: Truco
impresionar unaCarrera = applyIf (enamoradoxEstaEnElPublico unaCarrera) (mapVelocidad
(*2))
```

```
nitro:: Truco
nitro _ = mapVelocidad (+15)
```

```
comboLoco:: Truco
comboLoco unaCarrera = deReversaRocha unaCarrera.nitro unaCarrera
```

```
type Vueltas = Int
type LongitudPista = Int
type Participantes = [Auto]
type Publico = [String]
```

```
data Carrera = Carrera {vueltas:: Vueltas,
```

```
longitudPista:: LongitudPista,  
participantes:: Participantes,  
publico:: Publico} deriving Show
```

-- Punto 3

```
restarNafta:: Distancia -> Auto -> Auto  
restarNafta unaDistancia unAuto = (mapNafta.subtract.(*unaDistancia).largoNombre) unAuto  
unAuto
```

```
restarNaftaParticipantes:: Carrera -> Carrera  
restarNaftaParticipantes unaCarrera = (mapParticipantes.map.restarNafta.longitudPista)  
unaCarrera unaCarrera
```

```
largoNombre:: Auto -> Int  
largoNombre = length.nombre
```

```
nombreMasCortoQue:: Int -> Auto -> Bool  
nombreMasCortoQue = flip (.) largoNombre.flip (<=)
```

```
cuantoAumenta:: Auto -> Velocidad  
cuantoAumenta unAuto  
  | nombreMasCortoQue 5 unAuto = 15  
  | nombreMasCortoQue 8 unAuto = 20  
  | otherwise = 30
```

```
aumentarVelocidad:: Auto -> Auto  
aumentarVelocidad unAuto = (mapVelocidad.(+).cuantoAumenta) unAuto unAuto
```

```
aumentarVelocidadParticipantes:: Carrera -> Carrera  
aumentarVelocidadParticipantes = mapParticipantes.map $ aumentarVelocidad
```

```
autoMasLento:: Carrera -> Nombre  
autoMasLento = nombre.minimumBy velocidad.participantes
```

```
usarTruco:: Carrera -> Auto -> Auto  
usarTruco unaCarrera unAuto = (truco unAuto) unaCarrera unAuto
```

```
usarTrucoDelMasLento:: Carrera -> Carrera  
usarTrucoDelMasLento unaCarrera = mapParticipantes (mapIf ((== autoMasLento  
unaCarrera).nombre) (usarTruco unaCarrera)) unaCarrera
```

```
darVuelta:: Carrera -> Carrera  
darVuelta =  
usarTrucoDelMasLento.aumentarVelocidadParticipantes.restarNaftaParticipantes
```

-- Punto 4

```
correrCarrera:: Carrera -> Carrera
correrCarrera unaCarrera = foldl1 (.) (replicate (vueltas unaCarrera) darVuelta) $ unaCarrera
```

-- Punto 5

```
ganador:: Carrera -> Auto
ganador = maximumBy velocidad.participantes.correrCarrera
```

-- Punto 6

```
recompetidores:: Carrera -> Participantes
recompetidores unaCarrera = filter (puedeCorrerDeNuevo
unaCarrera).participantes.correrCarrera $ unaCarrera
```

```
puedeCorrerDeNuevo:: Carrera -> Auto -> Bool
puedeCorrerDeNuevo unaCarrera unAuto = quedoConNafta unAuto || esElGanador
unaCarrera unAuto
```

```
quedoConNafta:: Auto -> Bool
quedoConNafta = (>27).nafta
```

```
esElGanador:: Carrera -> Auto -> Bool
esElGanador unaCarrera = (== (nombre.ganador) unaCarrera).nombre
```

-- Punto 7

-- a) No. v

-- b) No. v

-- c) No. v

-- No puede terminar de dar una vuelta porque para lo último que hace debe buscar al auto en la última posición, lo cual no terminaría nunca.