a) Funciones como Usuario del TAD Organizador
A continuación se presentan las implementaciones de las funciones solicitadas, asumiendo que eres un usuario del TAD Organizador y solo puedes utilizar su interfaz pública.

---

1. programasEnComun

- **Propósito**: Dadas dos personas y un organizador, denota el conjunto de programas en los que colaboraron.
- **Precondiciones**: Las personas deben existir en el organizador.

Haskell
```
programasEnComun :: Persona -> Persona -> Organizador -> Set Checksum
programasEnComun p1 p2 org =
 let -- Obtener el conjunto de programas para cada persona
    programasP1 = programasDe org p1 -- O(log P)
    programasP2 = programasDe org p2 -- O(log P)
 -- Encontrar la intersección de los dos conjuntos
 in intersection programasP1 programasP2 -- O(Cp * log Cp)
```

- **Justificación de Eficiencia**:
  - Se llama dos veces a programasDe, cada una con un costo de O(logP).
  - intersection sobre dos conjuntos tiene un costo de O(NlogN), donde N es el tamaño del conjunto más pequeño. Llamaremos Cp al número de programas de la persona con menos proyectos.
  - **Costo Total**: O(logP+CplogCp).

---

2. esUnGranHacker

- **Propósito**: Denota verdadero si la persona ha sido autora de todos los programas del organizador.
- **Precondiciones**: La persona debe existir en el organizador.

Haskell
```
esUnGranHacker :: Organizador -> Persona -> Bool
esUnGranHacker org persona =
 let -- Obtener el número total de programas
    totalProgramas = length (todosLosProgramas org) -- O(C)
    -- Obtener el número de programas para la persona dada
    programasPersona = nroProgramasDePersona org persona -- O(log P)
 in totalProgramas == programasPersona
```

- **Justificación de Eficiencia**:
  - todosLosProgramas cuesta O(C), y calcular el length de la lista resultante también cuesta O(C).
  - nroProgramasDePersona cuesta O(logP).
  - **Costo Total**: El costo final está dominado por la operación más costosa, resultando en O(C).

---

b) Implementación del TAD Organizador
Aquí está la implementación completa del TAD Organizador basada en la representación especificada.
Representación e Invariantes
Haskell
```
data Organizador = MkO (Map Checksum (Set Persona)) (Map Persona (Set Checksum))
```

- **Invariantes de Representación**:

1. **Consistencia**: Para cualquier par (checksum, personas) en el primer mapa, para cada persona en ese conjunto de personas, debe existir una entrada correspondiente en el segundo mapa donde la persona es una clave y el checksum está en su conjunto de programas.
2. **Simetría**: Lo inverso también debe ser cierto. Para cualquier par (persona, checksums) en el segundo mapa, para cada checksum en ese conjunto de checksums, la persona debe aparecer en el conjunto de autores para ese checksum en el primer mapa.
3. **Autores no vacíos**: Cada Set Persona asociado a un Checksum en el primer mapa no debe estar vacío.

Implementación de la Interfaz

- **nuevo**:
  - **Propósito**: Un organizador vacío.
  - **Eficiencia**: O(1).

Haskell
```
nuevo :: Organizador
nuevo = MkO emptyM emptyM
```

  - **Justificación**: Esta función solo utiliza emptyM, que es una operación de costo O(1).
- **agregarPrograma**:
  - **Propósito**: Agrega un programa con sus autores al organizador.
  - **Eficiencia**: Sin garantía requerida.

Haskell
```
agregarPrograma :: Organizador -> Checksum -> Set Persona -> Organizador
agregarPrograma (MkO mapCP mapPC) checksum personas =
 let -- Agregar el programa y sus autores al primer mapa
   nuevoMapCP = assocM checksum personas mapCP
   -- Actualizar el segundo mapa para cada autor
   nuevoMapPC = actualizarAutores (set2list personas) checksum mapPC
 in MkO nuevoMapCP nuevoMapPC

-- Subtarea para actualizar el mapa de personas a checksums
actualizarAutores :: [Persona] -> Checksum -> Map Persona (Set Checksum) -> Map Persona (Set
Checksum)
actualizarAutores [] _ mapPC = mapPC
actualizarAutores (p:ps) checksum mapPC =
 let -- Encontrar el conjunto actual de programas para la persona, o un conjunto vacío
   programasActuales = case lookupM p mapPC of
                 Just set -> set
                 Nothing  -> emptyS
   -- Agregar el nuevo programa al conjunto
   nuevosProgramas = addS checksum programasActuales
   -- Actualizar el mapa y continuar recursivamente
 in actualizarAutores ps checksum (assocM p nuevosProgramas mapPC)
```

  - **Justificación**: La eficiencia está dominada por actualizarAutores, que itera a través de todos los P_i autores del nuevo programa. Para cada autor, realiza lookupM (O(logP)), addS (O(logCp)), y assocM (O(logP)).
- **todosLosProgramas**:
  - **Propósito**: Una lista con todos los identificadores de programas.
  - **Eficiencia**: O(C).

Haskell
```
todosLosProgramas :: Organizador -> [Checksum]
```

todosLosProgramas (MkO mapCP _) = domM mapCP

- **Justificación**: Utiliza directamente domM en mapCP, que tiene un costo de O(C).

- **autoresDe**:
  - **Propósito**: El conjunto de autores para un programa dado.
  - **Eficiencia**: O(logC).

Haskell
autoresDe :: Organizador -> Checksum -> Set Persona
autoresDe (MkO mapCP _) checksum = fromJust (lookupM checksum mapCP)

- **Justificación**: Utiliza lookupM en mapCP, que cuesta O(logC).

- **programasDe**:
  - **Propósito**: El conjunto de programas en los que trabajó una persona.
  - **Eficiencia**: O(logP).

Haskell
programasDe :: Organizador -> Persona -> Set Checksum
programasDe (MkO _ mapPC) persona = fromJust (lookupM persona mapPC)

- **Justificación**: Utiliza lookupM en mapPC, que cuesta O(logP).

- **programaronJuntas**:
  - **Propósito**: Verifica si dos personas han trabajado juntas en algún programa.
  - **Eficiencia**: La consigna pide O(logP+ClogC), pero una implementación más eficiente es O(logP+CplogCp). Proveemos la versión más eficiente.

Haskell
programaronJuntas :: Organizador -> Persona -> Persona -> Bool
programaronJuntas org p1 p2 =
 let programasP1 = programasDe org p1 -- O(log P)
   programasP2 = programasDe org p2 -- O(log P)
   interseccion = intersection programasP1 programasP2 -- O(Cp * log Cp)
 in not (isEmptyS interseccion) -- O(1)

- **Justificación**: El costo es la suma de dos llamadas a programasDe (O(logP)) y una intersection (O(CplogCp)). El total es O(logP+CplogCp), donde Cp es el número de programas de la persona con menos proyectos.

- **nroProgramasDePersona**:
  - **Propósito**: La cantidad de programas en los que trabajó una persona.
  - **Eficiencia**: O(logP).

Haskell
nroProgramasDePersona :: Organizador -> Persona -> Int
nroProgramasDePersona org persona =
 let programas = programasDe org persona -- O(log P)
 in sizeS programas -- O(1)

- **Justificación**: Llama a programasDe (O(logP)) y a sizeS, que es O(1). El total es O(logP).

---

c) Variante del TAD con elMayorPrograma

Para agregar elMayorPrograma con eficiencia O(1), la representación debe modificarse para mantener un registro del programa con más autores.

- **Nueva Representación**:

Haskell
-- El Maybe (Checksum, Int) almacena el Checksum del programa con más
-- autores y la cantidad de dichos autores.
data Organizador = MkO (Map Checksum (Set Persona))
                       (Map Persona (Set Checksum))
                       (Maybe (Checksum, Int))

- **Nuevos Invariantes**:
    1. Si el tercer componente es Just (maxC, maxN), entonces maxC debe ser una clave en el primer mapa, y su conjunto de autores asociado debe tener tamaño maxN.
    2. El valor maxN debe ser mayor o igual al tamaño de cualquier otro conjunto de autores en el primer mapa.
    3. Si el primer mapa está vacío, el tercer componente debe ser Nothing.
- **Re-implementación de elMayorPrograma**:
    - **Propósito**: Denota un programa con la mayor cantidad de autores.
    - **Eficiencia**: O(1).

Haskell
```
elMayorPrograma :: Organizador -> Maybe Checksum
elMayorPrograma (MkO _ _ Nothing) = Nothing
elMayorPrograma (MkO _ _ (Just (checksum, _))) = Just checksum
```

    - **Justificación**: Es un simple *pattern matching* y acceso a un campo, lo cual es O(1).
- **agregarPrograma Modificado**: Esta es la única función que requiere un cambio sustancial.

Haskell
```
agregarPrograma :: Organizador -> Checksum -> Set Persona -> Organizador
agregarPrograma (MkO mapCP mapPC maybeMax) checksum personas =
 let -- Lógica original para actualizar los mapas
   nuevoMapCP = assocM checksum personas mapCP
   nuevoMapPC = actualizarAutores (set2list personas) checksum mapPC
   -- Nueva lógica para actualizar el valor máximo
   nuevoMax = case maybeMax of
    Nothing -> Just (checksum, sizeS personas)
    Just (_, maxN) -> if sizeS personas > maxN
                then Just (checksum, sizeS personas)
                else maybeMax
 in MkO nuevoMapCP nuevoMapPC nuevoMax
```

    - **Justificación**: La lógica añadida implica sizeS (O(1)) y una comparación, lo que no cambia la complejidad general (no especificada) de la función, pero permite que elMayorPrograma sea O(1).