Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Grupo Omega

Integrante	LU	Correo electrónico
Candia, Matias	721/19	candia.matias2000@gmail.com
Caneva, Diego Gabriel	169/18	diego.g.caneva@gmail.com
Lin Zabala, Juan Ignacio	349/18	juanignacio.lin@gmail.com
Sarmiento, Matias Federico	741/18	matiasfsarmiento@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Módulo Mapa

Interfaz

se explica con: MAPA.

géneros: `mapa`.

Operaciones:

CREARMAPA(**in** *alto* : `nat`, **in** *ancho* : `nat`, **in** *s* : `coordenada`, **in** *ll* : `coordenada`, **in** *f* : `conj(coordenada)`, **in** *p* : `conj(coordenada)`,**in** *ch* : `conj(coordenada)`) $\rightarrow$ *res* : `mapa`
Pre $\equiv \{s \neq ll \wedge todosEnRango(p \cup f \cup ch \cup \{s, ll\}, ancho, alto) \wedge \{s, ll\} \cap (f \cup p) = \emptyset \wedge disjuntosDeAPares(p, f, ch)\}$
Post $\equiv$ {res= nuevoMapa(ancho, alto, s, ll, p, f, ch)}
Complejidad: $\Theta(alto \times ancho)$
Descripción: Crea una nueva instancia de mapa

RESETEARCHOCOS(**in/out** *m* : `mapa`)
Pre $\equiv$ {true}
Post $\equiv$ {Restaura los chocolates a como fue creado originalmente}
Complejidad: $\Theta(c)$
Descripción: Pone en el mapa chocolates en las coordenadas del conjunto chocosOri

SALIDA(**in** *m* : `mapa`) $\rightarrow$ *res* : `coordenada`
Pre $\equiv$ {true}
Post $\equiv$ {res= inicio(m)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la salida del mapa

LLEGADA(**in** *m* : `mapa`) $\rightarrow$ *res* : `coordenada`
Pre $\equiv$ {true}
Post $\equiv$ {res= llegada(m)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la llegada del mapa

FANTASMAS(**in** *m* : `mapa`) $\rightarrow$ *res* : `arreglo(arreglo(bool))`
Pre $\equiv$ {true}
Post $\equiv \{(\forall c : coordenada)(enRango(c, largo(m), alto(m)) \Rightarrow_{\text{L}} (c \in fantasmas(m) \iff res[c] = true))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve una matriz de bool con true donde hay fantasmas y false donde no
Aliasing: Devuelve una referencia no modificable

PAREDES(**in** *m* : `mapa`) $\rightarrow$ *res* : `arreglo(arreglo(bool))`
Pre $\equiv$ {true}
Post $\equiv \{(\forall c : coordenada)(enRango(c, largo(m), alto(m)) \Rightarrow_{\text{L}} (c \in paredes(m) \iff res[c] = true))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve una matriz de bool con true donde hay paredes y false donde no
Aliasing: Devuelve una referencia no modificable

CHOCOLATES(**in** m : `mapa`) $\rightarrow$ res : `arreglo(arreglo(bool))`
Pre $\equiv$ {true}
Post $\equiv$ $\{(\forall c : coordenada)(enRango(c, largo(m), alto(m)) \Rightarrow_{\text{L}} (res[c] = true \ \Rightarrow \ c \in chocolates(m)))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve una matriz de bool con true donde hay chocolates y false donde no
Aliasing: Devuelve una referencia modificable

Representación

`mapa` se representa con `estrMapa`

donde `estrMapa` es `tupla`(*alto*: `nat` , *ancho*: `nat` , *salida*: `coordenada` , *llegada*: `coordenada` , *chocolates*: `arreglo(arreglo(bool))` , *fantasmas*: `arreglo(arreglo(bool))` , *paredes*: `arreglo(arreglo(bool))` , *chocosOri*: `conj(coordenada)`)

Rep : estrMapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\Longleftrightarrow$

1. $m.chocolates$, $m.fantasmas$ y $m.paredes$ tienen dimensión $m.alto \times m.ancho$.
2. $m.salida$ y $m.llegada$ son coordenadas en rango. y distintos
3. $m.fantasmas$ y $m.paredes$ tienen falso al indexarlos en $m.salida$ y en $m.llegada$.
4. Las coordenadas con valor true de $m.chocolates$ son elementos de $m.chocosOri$.
5. No hay ninguna coordenada con valor true en más de una de las matrices $m.chocolates$, $m.fantasmas$ y $m.paredes$.
6. No hay ninguna coordenada de $m.chocosOri$con valor true en las matrices $m.fantasmas$ y $m.paredes$.

Abs : estrMapa e $\longrightarrow$ mapa {Rep(e)}

Abs(e) $\equiv$ m: mapa $\mid$ $largo(m) = e.ancho \land alto(m) = e.alto \land inicio(m) = e.salida \land llegada(m) = e.llegada \land (\forall c : coordenada)(c \in paredes(m) \iff e.paredes[c] = true) \land (\forall c : coordenada)(c \in fantasmas(m) \iff e.fantasmas[c] = true) \land chocolates(m) = e.chocosOri$

Algoritmos

---

crearMapa(**in** *alto*: `nat`, **in** *ancho*: `nat`, **in** *s*: `coordenada`, **in** *ll*: `coordenada`, **in** *f*: `conj(coordenada)`, **in** *p*: `conj(coordenada)`,**in** *ch*: `conj(coordenada)`) $\rightarrow$ *res* : mp

1: $res.alto \leftarrow alto$ ▷ $\Theta(1)$
2: $res.ancho \leftarrow ancho$ ▷ $\Theta(1)$
3: $res.salida \leftarrow s$ ▷ $\Theta(1)$
4: $res.llegada \leftarrow ll$ ▷ $\Theta(1)$
5: $res.chocosOri \leftarrow ch$ ▷ $\Theta(c)$
6: $res.fantasmas \leftarrow inicializar(alto, inicializar(ancho, false))$ ▷ $\Theta(ancho \times alto)$
7: $res.paredes \leftarrow inicializar(alto, inicializar(ancho, false))$ ▷ $\Theta(ancho \times alto)$
8: $res.chocolates \leftarrow inicializar(alto, inicializar(ancho, false))$ ▷ $\Theta(ancho \times alto)$
9: $itConj(coordenada)\ itF \leftarrow crearIt(f)$ ▷ $\Theta(1)$
10: **while** $haySiguiente(itF)$ **do** ▷ $\Theta(1)$. El ciclo se hace $\#f$ veces
11: $x \leftarrow siguiente(itF).first$, $y \leftarrow siguiente(itF).second$; ▷ $\Theta(1)$
12: **for** $j \leftarrow max(-3, -x)$; $j <= min(3, res.alto - 1 - x)$;$++ j$ **do**
13: **for** $k \leftarrow max(-3, -y)$; $k <= min(3, res.ancho - 1 - y)$; $++ k$ **do**
14: **if** $abs(k) + abs(j) <= 3$ **then**
15: $res.fantasmas[x + j][y + k] \leftarrow true$
16: **end if**
17: **end for**
18: **end for**
19: $avanzar(itF)$ ▷ $\Theta(1)$
20: **end while**
21: $itConj(coordenada)\ itP \leftarrow crearIt(p)$ ▷ $\Theta(1)$
22: **while** $haySiguiente(itP)$ **do** ▷ $\Theta(1)$. El ciclo se hace $\#p$ veces
23: $res.paredes[siguiente(itP)] \leftarrow true$ ▷ $\Theta(1)$
24: $avanzar(itP)$ ▷ $\Theta(1)$
25: **end while**
26: $itConj(coordenada)\ itCh \leftarrow crearIt(ch)$ ▷ $\Theta(1)$
27: **while** $haySiguiente(itCh)$ **do** ▷ $\Theta(1)$. El ciclo se hace c veces
28: $res.chocolates[siguiente(itCh)] \leftarrow true$ ▷ $\Theta(1)$
29: $avanzar(itCh)$ ▷ $\Theta(1)$
30: **end while**

Complejidad: $\Theta(alto \times ancho)$, ya que $\#f, \#p$ y $c \leq alto \times ancho$

---

resetearChocos(**in/out** m : mp)

1: $itConj(coordenada)$ $itCh \leftarrow crearIt(m.chocosOri)$ ▷ $\Theta(1)$
2: **while** $haySiguiente(itCh)$ **do** ▷ $\Theta(c)$
3: $res.chocolates[siguiente(itCh)] \leftarrow true$ ▷ $\Theta(1)$
4: $avanzar(itCh)$ ▷ $\Theta(1)$
5: **end while**

Complejidad: $\Theta(c)$

---

salida(**in** m : mp) → res : coordenada

1: $res \leftarrow m.salida$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

llegada(**in** m : mp) → res : coordenada

1: $res \leftarrow m.llegada$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

fantasmas(**in** m : mp) → res : arreglo(arreglo(bool))

1: $res \leftarrow m.fantasmas$ ▷ $\Theta(1)$

Esto ahora rompe la especificación.

Complejidad: $\Theta(1)$
Es $\Theta(1)$ porque retorna un referencia (no modificable)

---

paredes(**in** m : mp) → res : arreglo(arreglo(bool))

1: $res \leftarrow m.paredes$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Es $\Theta(1)$ porque retorna un referencia (no modificable)

---

chocolates(**in** m : mp) → res : arreglo(arreglo(bool))

1: $res \leftarrow m.chocolates$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Es $\Theta(1)$ porque retorna un referencia (modificable)

Servicios Usados

Del módulo Conjunto Lineal(α) se utilizó:

- CrearIt(in c: conj(α)): crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalúe a false. Tiene una complejidad $\Theta(1)$.
- haySiguiente(in it: itConj(α)): devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar. Tiene una complejidad $\Theta(1)$.
- Siguiente(in it: itConj(α)): devuelve el elemento siguiente a la posición del iterador. Tiene una complejidad $\Theta(1)$.
- Avanzar(in/out it: itConj(α)): avanza el iterador a la posición siguiente. Tiene una complejidad $\Theta(1)$.

Del módulo Arreglo(α) se utilizó:

- $\bullet[\bullet]$: devuelve o define el elemento que se encuentra en la i-ésima posición del arreglo en base 0. Tiene una complejidad de $\Theta(1)$.
- Extendemos el módulo Arreglo con la siguiente operación:

INICIALIZAR(**in** dim : `nat`,**in** a : α) $\rightarrow$ res : `arreglo(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{long(res) = dim \wedge (\forall i : nat)(0 \leq i < dim \Rightarrow_{\text{L}} res[i] == a)\}$
Complejidad: $\Theta(dim * copy(a))$
Descripción: Devuelve un arreglo de dim posiciones todas con valor a
Aliasing: No aplica

2. Módulo Partida

Interfaz

se explica con: PARTIDA.

géneros: `partida`.

Operaciones:

CREARPARTIDA(**in** m : `mapa`) $\rightarrow$ res : `partida`
Pre $\equiv$ {true}
Post $\equiv$ {res = nuevaPartida(m)}
Complejidad: $\Theta(1)$
Descripción: Crea una nueva instancia de partida
Aliasing: Mapa es pasado por referencia modificable

RESETEARPARTIDA(**in/out** p: `partida`)
Pre $\equiv$ $\{p = P_0\}$
Post $\equiv$ {p = nuevaPartida(resetearChocos(mapa(P_0)))}
Complejidad: $\Theta(c)$
Descripción: Resetea los choclates en el mapa de la partida
Aliasing: Modifica el mapa de la partida

MOVERSE(**in/out** p: `partida`,**in** d: `dir`)
Pre $\equiv \{p = P_0 \wedge \neg(gano?(p) \vee perdio?(p))\ \}$
Post $\equiv \{p = mover(P_0, d)\}$
Complejidad: $\Theta(1)$
Descripción: Mueve el jugador en la dirección d, actualiza la inmunidad según corresponda, si come un chocolate lo borra del mapa, actualiza el puntaje
Aliasing: Modifica el mapa de partida en caso de comer un chocolate

ACTUAL(**in** p: `partida`) $\rightarrow$ res : `coordenada`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res = jugador(p)\}$
Complejidad: $\Theta(1)$
Descripción: Retorna la coordenada actual del jugador
Aliasing: No aplica

GANO?(**in** p: `partida`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res = gano?(p)\}$
Complejidad: $\Theta(1)$
Descripción: Retorna si el jugador ganó la partida
Aliasing: No aplica

PERDIO?(**in** p: `partida`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res = perdio?(p)\}$
Complejidad: $\Theta(1)$
Descripción: Retorna si el jugador perdió la partida
Aliasing: No aplica

PUNTAJE(**in** p: `partida`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} = \text{cantMov(p)}\}$
Complejidad: $\Theta(1)$
Descripción: Retorna el puntaje de la partida
Aliasing: No aplica

Representación

`partida` se representa con `estrPartida`

donde `estrPartida` es `tupla`(*mapa*: `mapa` , *actual*: `coordenada` , *inmunidad*: `nat` , *puntaje*: `nat`)

Rep : estrPartida $\longrightarrow$ bool
Rep(p) $\equiv$ true $\Longleftrightarrow$

1. $0 \leq inmunidad \leq 10$.
2. *actual* debe ser una coordenada válida.
3. *puntaje* debe ser mayor o igual que $distManhattan(salida, actual)$.

Abs : estrPartida e $\longrightarrow$ partida $\{\text{Rep}(e)\}$

Abs(e) $\equiv$ p: partida $\mid$ $mapa(p) = e.mapa \land jugador(p) = e.actual \land cantMov(p) = e.puntaje \land inmunidad(p) = e.inmunidad \land_{\text{L}} (\forall c : coordenada)(c \in chocolates(p) \iff chocolates(e.mapa)[c] == true)$

Algoritmos

---

crearPartida(**in** m : `mapa`) $\rightarrow$ res : partida

1: $res.mapa \leftarrow m$ ▷ $\Theta(1)$
2: $res.actual \leftarrow salida(m)$ ▷ $\Theta(1)$
3: $res.inmunidad \leftarrow 0$ ▷ $\Theta(1)$
4: **if** $chocolates(m)[res.actual]$ **then** ▷ $\Theta(1)$
5: $\quad res.inmunidad \leftarrow 10$ ▷ $\Theta(1)$
6: $\quad chocolates(m)[res.actual] \leftarrow false$ ▷ $\Theta(1)$
7: **end if**
8: $res.puntaje \leftarrow 0$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
$res.mapa$ se le asigna m por referencia

---

resetearPartida(**in/out** p : `partida`)

1: $resetearChocos(res.mapa) \leftarrow m$ ▷ $\Theta(c)$
2: $res.actual \leftarrow salida(m)$ ▷ $\Theta(1)$
3: $res.inmunidad \leftarrow 0$ ▷ $\Theta(1)$
4: **if** $chocolates(m)[res.actual]$ **then** ▷ $\Theta(1)$
5: $\quad res.inmunidad \leftarrow 10$ ▷ $\Theta(1)$
6: $\quad chocolates(m)[res.actual] \leftarrow false$ ▷ $\Theta(1)$
7: **end if**
8: $res.puntaje \leftarrow 0$ ▷ $\Theta(1)$

Complejidad: $\Theta(c)$

---

moverse(**in/out** p : `partida`,**in** d : `dir`)

1: $coordenada\ destino \leftarrow p.actual$ ▷ $\Theta(1)$
2: **if** $d == DERECHA$ **then** ▷ $\Theta(1)$
3: $destino = destino + \langle 0, 1 \rangle$ ▷ $\Theta(1)$
4: **end if**
5: **if** $d == IQUIERDA$ **then** ▷ $\Theta(1)$
6: $destino = destino + \langle 0, -1 \rangle$ ▷ $\Theta(1)$
7: **end if**
8: **if** $d == ABAJO$ **then** ▷ $\Theta(1)$
9: $destino = destino + \langle 1, 0 \rangle$ ▷ $\Theta(1)$
10: **end if**
11: **if** $d == ARRIBA$ **then** ▷ $\Theta(1)$
12: $destino = destino + \langle -1, 0 \rangle$ ▷ $\Theta(1)$
13: **end if**
14: **if** $0 \leq \pi_0(destino) < alto(p.mapa) \wedge 0 \leq \pi_1(destino) < ancho(p.mapa)$ **then** ▷ $\Theta(1)$
15: **if** $\neg paredes(p.mapa)[destino]$ **then** ▷ $\Theta(1)$
16: $p.actual \leftarrow destino$ ▷ $\Theta(1)$
17: $res.puntaje++$ ▷ $\Theta(1)$
18: **if** $chocolates(m)[p.actual]$ **then** ▷ $\Theta(1)$
19: $p.inmunidad \leftarrow 10$ ▷ $\Theta(1)$
20: $chocolates(m)[p.actual] \leftarrow false$ ▷ $\Theta(1)$
21: **else**
22: **if** $inmunidad > 0$ **then**
23: $p.inmunidad--$ ▷ $\Theta(1)$
24: **end if**
25: **end if**
26: **end if**
27: **end if**

Complejidad: $\Theta(1)$

---

actual(**in** p : `partida`) $\rightarrow$ res : coordenada

1: $res \leftarrow p.actual$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

gano?(**in** p : `partida`) $\rightarrow$ res : bool

1: $res \leftarrow p.actual == llegada(p.mapa)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

perdio?(**in** p : `partida`) $\rightarrow$ res : bool

1: $res \leftarrow fantasmas(p.mapa)[p.actual] \wedge p.inmunudad == 0$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$ Esto es más bien posiciones de susto

---

puntaje(**in** p : `partida`) $\rightarrow$ res : nat

1: $res \leftarrow p.puntaje$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Servicios Usados

Del módulo Arreglo(α) se utilizó:

- $\bullet[\bullet]$: devuelve el elemento que se encuentra en la i-ésima posición del arreglo en base 0. Tiene una complejidad de $\Theta(1)$.

3. Módulo Fichin

Interfaz

se explica con: FICHIN.

géneros: `fichin`.

Operaciones:

CREARFICHIN(**in** m : `mapa`) $\rightarrow$ res : `fichin`
Pre $\equiv$ {true}
Post $\equiv$ {res= nuevoFichin(m)}
Complejidad: $\Theta(1)$
Descripción: Crea una nueva instancia de fichin
Aliasing: Mapa es pasado por referencia modificable

NUEVAPARTIDA(**in/out** f : `fichin`, **in** $string$: `j`)
Pre $\equiv$ $\{f = F_0 \wedge \neg(alguienJugando?(f))\}$
Post $\equiv$ $\{f = nuevaPartida(F_0, j)\}$
Complejidad: $\Theta(c)$
Descripción: Genera una nueva instancia de Partida dentro del fichin f con jugador j
Aliasing: Resetea los chocolates del mapa

MOVER(**in/out** f : `fichin`, **in** dir : `d`)
Pre $\equiv$ $\{f = F_0 \wedge alguienJugando?(f)\}$
Post $\equiv$ $\{f = mover(F_0, d)\}$
Complejidad: Si se ganó la partida es $\Theta(|J|)$, caso contrario es $\Theta(1)$
Descripción: Se mueve la posición del jugador dentro de la partida, siempre que la dirección d de un movimiento válido
Aliasing: En caso de comerse un chocolate se modifica el mapa

VERRANKING(**in** f : `fichin`) $\rightarrow$ res : `lista(`$\langle$`string,nat`$\rangle$`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = ranking(f)\}$
Complejidad: $\Theta(1)$
Descripción: Retorna un diccionario con el nombre y el mejor puntaje de cada jugador que haya ganado alguna partida
Aliasing: Se retorna una referencia no modicable

OBJETIVO(**in** f : `fichin`) $\rightarrow$ res : $\langle$`string,nat`$\rangle$
Pre $\equiv$ $\{alguienJugando?(f) \wedge_{\text{L}} definido(ranking(f), jugadorActual(f))\}$
Post $\equiv$ $\{res = objetivo(f)\}$
Complejidad: $\Theta(J)$
Descripción: Retorna el jugador cuyo puntaje en el ranking es inmediatamente mejor al puntaje en el ranking del jugador actual y su respectivo puntaje

Aliasing: No aplica

Representación

`fichin` se representa con `estrFichin`

donde `estrFichin` es `tupla`(*partida*: `partida` , *jugando*: `bool` , *jugador*: `string` , *jugadores*: `diccTrie(string, nat)`)

Rep : estrFichin $\longrightarrow$ bool
Rep(f) $\equiv$ true $\Longleftrightarrow$

1. Si *jugando* es true, entonces *jugador* no es vacío y *partida* no puede estar ganada ni perdida.

Abs : estrFichin e $\longrightarrow$ fichin {Rep(e)}
Abs(e) $\equiv$ f: fichin | $mapa(f) = reiniciarChocos(e.mapa) \wedge alguienJugando(f) = e.jugando \wedge_{\text{L}} (alguienJugando(f) = true \Rightarrow_{\text{L}} jugadorActual(f) = e.jugador) \wedge partidaActual(f) = e.partida \wedge claves(ranking(f)) = claves(e.jugadores) \wedge_{\text{L}} (\forall j : string)(j \in claves(ranking(f))) \Rightarrow_{\text{L}} significado(ranking(f), j) = puntos(Siguiente(significado(e.jugadores, j))))$

Algoritmos

---

crearFichin(**in** m : `mapa`) $\rightarrow$ res : fichin

1: $res.partida \leftarrow crearPartida(m)$ ▷ $\Theta(1)$
2: $res.jugando \leftarrow false$ ▷ $\Theta(1)$
3: $res.jugador \leftarrow$ "" ▷ $\Theta(1)$
4: $res.jugadores \leftarrow vacio()$ ▷ $\Theta(1)$
5: $res.ranking \leftarrow vacio()$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
A *res.mapa* se le asigna una referencia a m

---

nuevaPartida(**in/out** f : `fichin`, **in** j : `string`)

1: $f.partida \leftarrow resetearPartida(f.partida)$ ▷ $\Theta(c)$
2: $f.jugando \leftarrow true$ ▷ $\Theta(1)$
3: $f.jugador \leftarrow j$ ▷ $\Theta(1)$
4: $f.puesto =$ crearIt($f.ranking$) ▷ $\Theta(1)$

Complejidad: $\Theta(c)$

---

mover(**in/out** f : `fichin`, **in** d : `dir`)

```
1:  moverse(f.partida, d)                                                          ▷ Θ(1)
2:  if HaySiguiente(f.puesto) ∧_L puntaje(f.partida) ≥ puntos(Siguiente(f.puesto)) then   ▷ Θ(1)
3:      Avanzar(f.puesto)                                                          ▷ Θ(1)
4:  end if
5:  if gano?(f.partida) then                                                       ▷ Θ(1)
6:      if definido?(f.jugadores, f.jugador) then                                  ▷ Θ(|J|)
7:          if puntos(Siguiente(significado(f.jugadores, f.jugador))) > puntaje(f.partida) then   ▷ Θ(|J|)
8:              EliminarSiguiente(significado(f.jugadores, f.jugador))             ▷ Θ(|J|)
9:              AgregarComoSiguiente(f.puesto, ⟨f.jugador, puntaje(f.partida)⟩)    ▷ Θ(1)
10:             definir(f.jugadores, f.jugador, f.puesto)                          ▷ Θ(|J|)
11:         end if
12:     else
13:         AgregarComoSiguiente(f.puesto, ⟨f.jugador, puntaje(f.partida)⟩)        ▷ Θ(1)
14:         definir(f.jugadores, f.jugador, f.puesto))                             ▷ Θ(|J|)
15:     end if
16: end if
17: f.jugando ← ¬ (gano?(f.partida) ∨ perdio?(f.partida) )                         ▷ Θ(1)
```

Complejidad: $\Theta(|J|)$

---

verRanking(**in** f : `fichin`) → res : lista(⟨string,nat⟩)

```
1:  res ← f.ranking                                                                ▷ Θ(1)
```

Complejidad: $\Theta(1)$
A res se le asigna un referencia no modificable

objetivo(**in** f : `fichin`) $\rightarrow$ res : $\langle$string,nat$\rangle$

1: $puntos \leftarrow$ significado($f.jugadores, f.jugador$) ▷ $\Theta(|J|)$
2: $obj2 \leftarrow 0$ ▷ $\Theta(1)$
3: $itDiccTrie(string, nat)$ $itR \leftarrow crearIt(f.jugadores)$ ▷ $\Theta(1)$
4: **while** $haySiguiente(itR)$ **do** ▷ $\Theta(1)$. El ciclo se hace J veces
5: $sig \leftarrow siguienteSignificado(itR)$ ▷ $\Theta(1)$
6: **if** $obj2 \leq sig \wedge sig < puntos$ **then** ▷ $\Theta(1)$
7: $obj2 \leftarrow sig$ ▷ $\Theta(1)$
8: $obj1 \leftarrow siguienteClave(itR)$ ▷ $\Theta(1)$
9: **end if**
10: $avanzar(itR)$ ▷ $\Theta(1)$
11: **end while**
12: **if** $obj2 == 0$ **then** ▷ $\Theta(1)$
13: $obj2 \leftarrow puntos$ ▷ $\Theta(1)$
14: $obj1 \leftarrow f.jugador$ ▷ $\Theta(1)$
15: **end if**
16: $res \leftarrow \langle obj1, obj2 \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(J)$

Servicios Usados

Del módulo Lista Enlazada(α) se utilizó:

- CrearIt(in l: lista(α)): crea un iterador bidireccional de la lista, de forma tal que al pedir Siguiente se obtenga el primer elemento de l. Tiene una complejidad $\Theta(1)$.
- haySiguiente(in it: itLista(α)): devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar. Tiene una complejidad $\Theta(1)$.
- Siguiente(in it: itLista(α)): devuelve el elemento siguiente a la posición del iterador. Tiene una complejidad $\Theta(1)$.
- hayAnterior(in it: itLista(α)): devuelve `true` si y sólo si en el iterador todavía quedan elementos para retroceder. Tiene una complejidad $\Theta(1)$.
- Anterior(in it: itLista(α)): devuelve el elemento anterior del iterador. Tiene una complejidad $\Theta(1)$.
- Avanzar(in/out it: itLista(α)): avanza el iterador a la posición siguiente. Tiene una complejidad $\Theta(1)$.
- EliminarSiguiente(in/out it: itLista(α)): elimina de la lista iterada el valor que se encuentra en la posición siguiente del iterador. Tiene una complejidad $\Theta(1)$.
- AgregarComoSiguiente(in/out it: itLista(α), in a: α): agrega el elemento a a la lista iterada, entre las posiciones anterior y siguiente del iterador, dejando al iterador posicionado de forma tal que al llamar a Siguiente se obtenga a. Tiene una complejidad ($\Theta(copy(a))$.

El módulo Diccionario Trie (κ, σ) es un diccionario implementado sobre un trie con la siguiente interfaz:

se explica con: Diccionario.

géneros: `diccTrie`.

DEFINIR(**in/out** d: `dicc(`κ, σ`)`, **in** k: κ, **in** s: σ)
Pre $\equiv \{d = D_0\}$
Post $\equiv \{d = \text{definir}(D_0, k, s)\}$
Complejidad: Θ(longitud del elemento de tipo κ más largo de las claves de d) + $\Theta(copy(s))$.
Descripción: define la clave k con el significado s en el diccionario.
Aliasing: los elementos k y s se definen por copia.

DEFINIDO?(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res = \text{def?}(d,\ k)\}$
Complejidad: Θ(longitud del elemento de tipo κ más largo de las claves de d)
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

SIGNIFICADO(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : σ
Pre $\equiv \{\text{def?}(d,\ k)\}$
Post $\equiv \{res = \text{Significado}(d,\ k)\}$
Complejidad: Θ(longitud del elemento de tipo κ más largo de las claves de d)
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.