Algoritmos y Estructuras de Datos II

Trabajo Práctico 1

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

JuegoDePalabras

Integrante	LU	Correo electrónico
Tiracchia, Thiago	1502/21	thiago.tiracchia.com
Vega, Angela	585/21	luciavsm@gmail.com
Bousoño, Ignacio	503/21	bousonoignacio@gmail.com
Oscar Manuel Lopez Coronel	519/21	lopezcoroneloscarmanuel@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

Comentario para el corrector:Nos dimos cuenta que los algoritmos de Juego y Servidor estan poco modularizados o "prolijos", pudimos haberlo hecho mas elegante con los algoritmos usando los que ya declaramos ya que es la idea de usar interfaces, pero el tp fue cambiando mucho y nos quedamos cortos de tiempo

1. Módulo Juego

1.1. Interfaz

usa: TABLERO, JUGADOR, COLA, VARIANTE

se explica con: JUEGO

géneros: `juego`.

Operaciones básicas de Juego

INICIARJUEGO(**in** *jugadores*: `nat`, **in** *r*: `cola(letra)`, **in** *v*: `variante`) → *res* : `juego`
Pre ≡ {tamaño(r) ≥ tamañoTablero(v)^2 + jugadores · #fichas(v)}
Post ≡ { res=nuevoJuego(jugadores,v,r)}
Complejidad: $\mathcal{O}(N^2 + \Sigma \cdot K + F \cdot K)$
Descripción: Inicia un juego
Aliasing: Produce aliasing ya que pasa la variante del juego y el repositorio por referencia

JUGADAVALIDA(**in** *j*: `juego` , **in** *jugada*: `ocurrencia`) → *res* : `bool`
Pre ≡ {longitud(jugada) ≤ LongitudPalabraMax(PalabrasLegitimas(variante(j))}
Post ≡ {res = jugadaValida?(j,jugada)}
Complejidad: $\mathcal{O}(L^2_{max})$
Descripción: Indica si una jugada es válida.
Aliasing: Sin aliasing

LongitudPalabraMax : Palabras Conjunto(Secuencia(Letra)) ⟶ nat
LongitudPalabraMax(c) = if vacio?(c) then 0 else if long(dameUno(c)) >longitudPalabraMax(sinUno(c)) then long(dameUno(c)) else longitudPalabraMax(sinUno(c))

UBICARCONJJUEGO(**in/out** *j*: `juego`, **in** *c*: `ocurrencia`)
Pre ≡ {$j = j_0 \wedge$ JugadaValida?(j,c) $\wedge$ OcurrenciaValida(c) }
Post ≡ {j = ubicar(j_0, c)}
Complejidad: $\mathcal{O}(m)$
Descripción: Ubica fichas en el tablero
Aliasing: Genera aliasing con la instancia de juego pasado como parámetro

INFOVARIANTE(**in** *j*: `juego`) → *res* : `variante`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ variante(j)}
Complejidad: $\mathcal{O}(1)$
Descripción: Muestra información sobre la variante actual del juego
Aliasing: Produce aliasing ya que devuelve la variante del juego por referencia

TURNODELJUGADOR(**in** *j*: `juego`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$turno(j)}
Complejidad: $\mathcal{O}(1)$
Descripción: Muestra el jugador al que le toca jugar
Aliasing: Produce aliasing ya que pasa la variante del juego por referencia

PUNTAJEDELJUGADOR(**in** *j*: `juego`,**in** *jug*: `nat`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$puntaje(j,jug)}
Complejidad: $\mathcal{O}(1 + m \cdot L_{max})$
Descripción: Devuelve el puntaje de un jugador
Aliasing: Sin aliasing

ObtContCoord(**in** j : `juego`,**in** x : `nat`,**in** x : `nat`) $\rightarrow$ res : `letra`
Pre $\equiv$ {0 $\leq$ x $<$ tamaño(tablero(j)) $\wedge$ 0 $\leq$ y $<$ tamaño(tablero(j)) $\wedge_L$ hayLetra?((tablero(j),x,y) }
Post $\equiv$ {res $=_{obs}$letra(tablero(j),x,y)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dada un posición en el tablero devuelve su contenido
Aliasing: Sin aliasing

#LetraXEnJugador(**in** j : `juego`,**in** x : `letra`,**in** jug : `nat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {x $\in$ Repositorio(j) (o expandir variante para que de el dicc)$\wedge$ jug $<$ #jugadores(j)}
Post $\equiv$ {res = CantiApariciones(fichas(j,jug),X)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de letras X que tiene jugador
Aliasing: Sin aliasing

reponer(**in/out** j : `juego`,**in** cid : `Nat`,**in** o : `ocurrencia`) $\rightarrow$ res : `MultiConj(Letra)`
Pre $\equiv$ {j = j0 $\wedge$ OcurrenciaValida(o) $\wedge$ cid $<$#jugadores(j) }
Post $\equiv$ { Se le sacan las fichas de la ocurrencia a las fichasenmano del jugador y se reemplazan por las de res, y al repositorio se le desencolan las proximas |o| fichas $\wedge$ res = FichasQueRepuso}
Complejidad: $\mathcal{O}(|o|)$ donde esta esta acotada por Lmax
Descripción: Repone $|o|$ cantidad de fichas y devuelve que fichas fueron repuestas
Aliasing: aliasing ya que modifica el repositorio y las fichas en mano de los jugadores

Fin Interfaz

TAD Variante

extiende Variante

otras operaciones

PalabrasLegitimas : v $\longrightarrow$ variante {nat}

PalabrasLegitimas(NuevaVariante(n,f,d,c) $\equiv$ c

Fin TAD

1.2. Representación

Representación de Juego

`juego` **se representa con** `estr`

donde `estr` es `tupla`(*varianteDelJuego*: `variante`, *turnoGeneral*: `nat`, *repositorio*: `cola(letra)`
, *t*: `tablero` , *fichasEnMano*: `vector(vector(nat) ) (fichas en mano)`
, *jug*: `vector(tupla(puntaje : nat, fichasUbicadas : cola(tupla(fichasJugadas : ocurrencia,turno : Nat ,EsHorizontal? : bool))))`
, # *jugadores*: `nat`)

Ficha es tupla <x : nat , y : nat , l : letra>
Ocurrencia es ListaEnlazada(Ficha)
Post Correccion: Nos dimos cuenta que usando iteradores de un conjuntoLineal se cumple la misma funcion y las ocurrencias estan limitadas por Lmax y no hace falta usar Lmax en los algoritmos ya que todas las ocurrencias jugadas van a ser <= que Lmax, y los algoritmos no se ven apenas modificados mas alla de que en vez de recorrer con un while y un iterador i , se usa un iterador it, y la funcion haysiguiente?(it)

Rep : estr $\longrightarrow$ bool

Rep(j) $\equiv$ true $\iff$ TamañoTablero(($j.varianteDelJuego$)) = ($e.t$).tamaño $\wedge$
$0 \leq j.turnoGeneral < j.\#jugadores \wedge$
RepositorioValido($e.repositorio$, DiccPuntajexLetra($j.varianteDelJuego$))) $\wedge$
longitud($j.fichasEnMano$) = $j.\#jugadores \wedge$
($\forall$ i : nat)(0 $\leq$ i <longitud($j.fichasEnMano$) $\Rightarrow$ $_L$
longitud($j.fichasEnMano[i]$) = tamaño(claves(DiccPuntajexLetra($j.varianteDelJuego$)))))
$\wedge$ longitud($j.fichasEnMano$) = $j.$ # $jugadores$
$\wedge$ sumatoria(j.fichasEnMano[i]) = #Fichas(j.varianteDelJuego)$\wedge$ ($\forall$ j : nat) ($\forall$ l : letra) 0 $\leq$ j <longitud(j.jug) $\wedge$ l $\in$ claves(DiccPuntajexLetra(j.varianteDelJuego))) $\wedge$
($\forall$ i : nat) 0 $\leq$ i <longitud(j.jug) $\Rightarrow$ ($\forall$ o : ocurrencia)(o $\in$ j.jug[i].fichasUbicadas.fichasJugadas $\Rightarrow$ eshorizontal(o) = j.jug[i].fichasUbicadas.EsHorizontal?) $\wedge$ ($\forall$ f : tupla<x : Nat, y : Nat, l: Letra>) f $\in$ j.jug[i].fichasUbicadas.fichasJugadas $\Rightarrow$ (x <TamañoTablero(j.varianteDelJuego) $\wedge$ y <TamañoTablero(j.varianteDelJuego) $\wedge$ l $\in$ claves(DiccPuntajeXLetra(j.varianteDelJuego)))

para todo indice de j.jug su puntaje va a corresponder con las jugadas en el tablero pero como los elementos de tablero son una tupla<letra, turno en que se jugo la ficha, si esta vacia la casilla>y no tiene un elemento que haga referencia a que persona jugo esa ficha es complicado escribir la relacion puntaje tablero en la representacion. lo mismo con j.jug.fichasUbicadas.turno , ya que va a haber relacion entre el turno que se jugo la ocurrencia, las ocurrencias en el tablero.

RepositorioValido : cola(letra) $\times$ diccLineal(letra $\times$ nat) $\longrightarrow$ bool

RepositorioValido(r, d) $\equiv$ **if** $\neg$ EsVacia?(r) **then**
pertenece?(claves(d),proximo(r)) $\wedge$
RepositorioValido(desencolar(r),d)
else
true
fi

Sumatoria : vector[nat] $\longrightarrow$ Nat

Sumatoria(v) $\equiv$ $\sum_{i=0}^{(long(v)} v[i]$

Abs : estr e $\longrightarrow$ juego {Rep(e)}

Abs(e) =$_{obs}$ j: juego | e.varianteDelJuego = variante(j) $\wedge$ e.tablero = tablero(j) $\wedge$
e.turnoGeneral = turno(J) $\wedge$ e.repositorio = repositorio(j) $\wedge$

e.#jugadores = # jugadores(j) ∧
(∀ i : nat)(0 ≤ i < e.#*jugadores* ⇒
(∃ l : letra)(l = DameUno(fichas(j,i) $\wedge_L$ #fichas(j,i) = e.fichasEnMano[i][ord(l)])) ∧ (∀ i : nat) (0 ≤ i < e.#jugadores ⇒ (puntaje(j,i) = e.jug[i].puntaje))

1.3. Algoritmos

iIniciarJuego(**in** v : `variante`, **in** k : `nat` , , **in** r : `cola(letras)`) $\longrightarrow$ `j:juego`

```
1:  j.varianteDelJuego ← v                                   ▷ esto es porque se pasa por referencia Θ(1)
2:  j.t ← iNuevoTablero(iTamaoTablero(v))                    ▷ Esto es O(N^2), donde n es el tamaño de jugadores
3:  j.r ← r
4:  j.turno ← 0
5:  j.#jugadores ← k
6:  i ← 0
7:  l ← 0
8:  mientras i < k hacer                                     ▷ Esto es O(k), donde k numero de jugadores
9:      mientras l <#Claves(v.puntajeXletra) hacer           ▷ Esto es O(Σ) donde Σ es la cantidad de letras
10:         j.fichasEnMano[i][j] ← 0
11: z ← 0
12: mientras z < k hacer                                     ▷ Esto es O(k), donde k numero de jugadores
13:     j.jug[z] ←< 0, vacia(), vacia() >
14:     p ← 0
15:     mientras p < v.#fichas hacer        ▷ Esto es O(F), donde F es la cantidad de fichas para cada jugador
16:         j.fichasEnMano[i][ord(proximo(j.r))] ← j.fichasEnMano[i][ord(proximo(∗j.repositorio))] + +
17:         j, jug[i].fichasEnMano ← (agregarAdelante(proximo(j.repositorio) desencolar ( j.repositorio)
18:
19:         l ← l + 1
20:     i ← i + 1
21: devolver j
    Complejidad:O(N^2+ | Σ | K + FK)
    Explicacion: Asigna el repositorio y variante mediante referencia y crea el fichasEnMano para saber que
22: cantidad de letra x tiene cada jugador, despues reparte las fichas a cada jugador y le suma al fichasEnMano
23: las letras que le tocan.
```

iInfoVariante(**in** j : `juego`) $\longrightarrow$ `variante`

devolver j.varianteDelJuego
Complejidad: $O(1)$

iTurnoDelJugador(**in** j : `juego`) $\longrightarrow$ `jugador`

devolver j.turnoGeneral mód j.#Jugadores
Complejidad:$O(1)$

iObtContCoord(**in** x : `nat`, **in** y : `nat`) $\longrightarrow$ `ficha`

devolver iLetra(j.t, x, y)
Complejidad:$O(1)$

i#LetraXEnJugador(**in** j : `juego`, **in** x : `letra`, **in** jug : `jugador`) $\longrightarrow$ `nat`

n ← j.fichasEnMano[jug][ord(x)]
devolver n
Complejidad:$O(1)$

JugadaValida(**in** *jugada* : `ocurrencia`, **in** j : `juego`) $\longrightarrow$ `bool`

res $\leftarrow$ *false*
si (longitud(o) $\leq$ Lmax(j.varianteDelJuego) **entonces**
  si Chequeo(j, o) **entonces** ▷ $O(Lmax^2)$
    res $\leftarrow$ *true*
devolver RES
Complejidad:$O(Lmax^2)$
Explicación:la función chequeo verifica si todas las fichas ubicadas en una jugada forman palabras válidas tanto vertical como horizontalmente.

Pre $\equiv$ {Ocurrenia valida y dentro del tablero}
Post $\equiv$ {Devuelve si la ocurrencia forma alguna palabra valida segun el diccionario de PalsLegit con las fichas del tablero, tanto vertical y horizontalmente}

Chequeo(**in** j : `juego`, **in** o : `ocurrencia`) $\longrightarrow$ `bool`

j.tablero $\leftarrow$ *AUXUbicarConFichas*(*jugada*, *j.tablero*) ▷ $O(Lmax)$
res $\leftarrow$ *true*
$i \leftarrow 0$
mientras $\neg$*vacia?*(o) **hacer** ▷ $O(Lmax)$
  res $\leftarrow$ *res*$\wedge$ (FormaPalVertical?(prim(o), j.tablero, j) $\wedge$
  FormaPalHorizontal?(prim(o), j.tablero, j)) ▷ $O(Lmax)$
  $o \leftarrow fin(o)$
j.tablero $\leftarrow$ *DesUbicarConFichas*(*jugada*, *j.tablero*) ▷ $O(Lmax)$
devolver RES
Complejidad: O($Lmax^2$)

Puntaje(**in** j : `juego`, **in** id : `nat`) $\longrightarrow$ `nat`

puntos $\leftarrow$ *j.puntos*[*id*].*puntos* ▷ O(1)
PuntajePalabras $\leftarrow$ 0
mientras $\neg$ esVacia?(j.puntos[id].fichaspuestas) **hacer** ▷ O(K) donde K esla #Jugadas sin llamar a la funcion puntaje
  si (proximo(j.puntos[id].fichaspuestas).hor)) **entonces**
    PuntajePalabras $\leftarrow$ *PuntajePalabras* + *palabraQFormaHorizontal*(*prim*(*proximo*(*j.jug*[*id*].*fichaspuestas*).*jugada*), *proximo*(*j.puntos*[*id*].*fichaspuestas*).*turno*, j) ▷ $O(F * Lmax)$ donde F es la #Fichas por jugada
    PuntajePalabras $\leftarrow$ *PuntajePalabras* + *PuntajePalabraSQFormanVertical*(*proximo*(*j.jug*[*id*].*fichaspuestas*), j) ▷ $O(F * Lmax)$ donde F es la #Fichas por jugada
  else
    PuntajePalabras $\leftarrow$ *PuntajePalabras* + *PuntajepalabraQFormaVerticall*(*prim*(*proximo*(*j.jug*[*id*].*fichaspuestas*).*jugada*), *proximo*(*j.puntos*[*id*].*fichaspuestas*).*turno*, j) ▷ $O(F * Lmax)$ donde F es la #Fichas por jugada

    PuntajePalabras $\leftarrow$ *PuntajePalabras* + *PuntajePalabraSQFormanHorizontall*(*proximo*(*j.jug*[*id*].*fichaspuestas*), j) ▷ $O(F * Lmax)$ donde F es la #Fichas por jugada
  desencolar(j.puntos[id].fichaspuestas) ▷ $O(1)$
puntos $\leftarrow$ *puntos* + *PuntajePalabras*
devolver PUNTOS
Complejidad: O(1 + K*F*Lmax) = O(1 + m*Lmax) ya que (K*F)=m donde m es la #Fichas que ubico el jugador desde el ultimo llamado a la funcion puntaje

Pre $\equiv$ {x e y estan en rango del tablero, l esta definido en dicc puntajexLetra , tablero y juego valido}
Post $\equiv$ {Devuelve true si forma palabra vertical }

FormaPalVertical?(**in** f : `tupla<x : nat,y : nat, l: letra>`, **in** t : `tablero` , **in** j : `juego`) $\longrightarrow$ `bool`

$Palabra \leftarrow vacia()$
$Palabra \leftarrow agregarAtras(Palabra, f.l)$ ▷ $\Theta(1)$
$sig \leftarrow 1$
$ant \leftarrow 1$
$x \leftarrow (f.x)$
$y \leftarrow (f.y)$
$Lmax \leftarrow Lmax(j.varianteDeljuego)$
$res \leftarrow false$
$i \leftarrow 0$
mientras $i < Lmax$ **hacer** ▷ O(Lmax)
  si (y + sig < tamaño(t) $\wedge_L$ iHayLetra?(t[x][y + sig])) **entonces**
    agregarAtras(Palabra. iLetra(t,x,y+sig)
    $sig \leftarrow sig + 1$
  si (y -ant < tamaño(t) $\wedge_L$ iHayLetra?(t[x][y - ant]) **entonces**
    agregarAdelante(Palabra. iLetra(t,x,y-ant)
    $ant \leftarrow ant + 1$
  $i \leftarrow i + 1$
si ($\neg$ iHayLetra(t, y + sig +1) $\vee$ y + sig >tamaño(t)) $\wedge$ ($\neg$iHayLetra(t, y - ant - 1) $\vee$ y -ant >tamaño(t))
entonces $res \leftarrow definido(v.PalsLegit, Palabra)$ ▷ O(| $Palabra$ |) donde palabra puede llegar a ser Lmax
devolver RES
Complejidad:O(Lmax)
Explicación: Incia una lista con la ficha dada y agrega adelante y/o atras de lista las letras ubicadas en el tablero previamente, hace la iteracion hasta Lmax y se mueve en las coordenadas verticales dependiendo si la casilla siguiente esta vacia o no y si esta fuera de rango.

Pre $\equiv$ {x e y estan en rango del tablero, l esta definido en dicc puntajexLetra , tablero y juego valido}
Post $\equiv$ {Devuelve true si forma palabra horizontal }

FormaPalHorizontal?(**in** f : `tupla<x : nat,y : nat, l: letra>`, **in** t : `tablero` , **in** j : `juego`) $\longrightarrow$ `bool`

$Palabra \leftarrow vacia()$
$Palabra \leftarrow agregarAtras(Palabra, f.l)$ ▷ O(1)
$sig \leftarrow 1$
$ant \leftarrow 1$
$x \leftarrow (f.x)$
$y \leftarrow (f.y)$
$Lmax \leftarrow Lmax(j.varianteDeljuego)$ ▷ O(1)
$res \leftarrow false$
$i \leftarrow 0$
mientras $i < Lmax$ **hacer** ▷ O(Lmax)
  si x + sig <tamaño(t)$\wedge$ iHayLetra?(t[x + sig][y]) **entonces**
agregarAtras(Palabra. iLetra(t,x+ sig ,y)
$sig \leftarrow sig + 1$
  si x + (-ant) <tamaño(t)$\wedge$ iHayLetra?(t[x - ant][y]) **entonces**
agregarAdelante(Palabra. iLetra(t,x - ant ,y)
$ant \leftarrow ant + 1$
$i \leftarrow i + 1$
si ($\neg$ iHayLetra(t,x + sig +1, y) $\vee$ x + sig > tamaño(t)) $\wedge$ (iHayLetra(t, x - ant - 1) ,y) $\vee$ y -ant > tamaño(t))
entonces $res \leftarrow definido(v.PalsLegit, Palabra)$ ▷ O(| $Palabra$ |) donde palabra puede llegar a ser Lmax
devolver RES
Complejidad:O(Lmax)
Explicación: Incia una lista con la ficha dada y agrega adelante y/o atras de lista las letras ubicadas en el tablero previamente, hace la iteracion hasta Lmax y se mueve en las coordenadas horizontales dependiendo si la casilla siguiente esta vacia o no y si esta fuera de rango.

UbicarConFichas(**in** *jugada* : `ocurrencia`, **in** j : `juego`) $\longrightarrow$ `tablero`

$j.puntos[iturnodeljugador(j)].fichaspuestas \leftarrow encolar < o, j.turnoGeneral, esHorizontal?(o) >$

mientras ¬ EsVacia(jugada) **hacer** ▷ O(| jugada |) Donde jugada es la #fichas a ubicar
iPonerLetra(prim(o).l, prim(o).x,prim(o).y)
$j.tablero \leftarrow ModificarTurno(j.tablero, [o.x], [o.y], j.turnoGeneral)$
j.fichasEnMano[turnoJugador(j)][ord(prim(jugada))]-1
$jugada \leftarrow fin(jugada)$
$j.jug[turnoJugador(j)].fichasEnMano \leftarrow agregarAdelante(proximo(j.repositorio)$
$j.fichasEnMano[turnoJugador(j)][ord(proximo(j.repositorio))] \leftarrow$
j.fichasEnMano[turnoJugador(j)][ord(proximo(j.repositorio))]+1
desencolar(j.repositorio)
$j.turnoGeneral \leftarrow j.turnoGeneral + 1$
Complejidad:O(#*Fichas* a ubicar)
Explicación: Ubica el conjunto de fichas y resta la cantidad de letras que el jugador ponga en su fichasEnMano. Despues le reparte fichas del repositorio y le suma su aparicion en el fichasEnMano

Pre ≡ {igual que la funcion ubicarConfichas)}
Post ≡ {Ubica la ocurrencia pero no modifica el turno del casillero}

AUXUbicarConFichas(**in** *jugada* : `ocurrencia`, **in** j : `juego`)$\longrightarrow$ `tablero`

mientras ¬ EsVacia(jugada) **hacer** ▷ O(#*Fichas* a ubicar)
iPonerLetra(prim(o).l, prim(o).x,prim(o).y)
$jugada \leftarrow fin(jugada)$
devolver j.tablero
Complejidad:O(#*Fichasaubicar*)
Explicación: Ubica fichas sin poner el turno en las que se ubico para despues poder desubicarlo con comodidad.

Pre ≡ {La lista de fichas no puede ser vacía y las fichas deben ser consecutivas}
Post ≡ {Devuelve si las fichas están ubicadas horizontalmente}

esHorizontal?(**in** o : `Ocurrencia`) $\longrightarrow$ `res : bool`

$it \leftarrow CrearIt(o)$
$fila \leftarrow Siguiente(it).x$
$res \leftarrow true$
mientras HaySiguiente(it) **hacer**
  si Siguiente(it).x == fila **entonces**
    $res \leftarrow res \wedge true$
  else
    $res \leftarrow false$
devolver res
Complejidad: O(Lmax) el tamaño de o esta acotado por Lmax

Pre ≡ {ocurrencia tiene que ser valida y el tablero tambien}
Post ≡ {desubica la ocurrencia dada}

DesubicarConFichas(**in** *jugada* : `ocurrencia`, **in** t : `tablero`) $\longrightarrow$ `tablero`

si ¬ EsVacia(jugada) **entonces** ▷ O(#*Fichas* a desubicar)
  $t \leftarrow modificarVacia(t, primero(jugada).x, primero(jugada).y, true)$
  $t \leftarrow DesubicarConFichas(fin(jugada), t)$
devolver t
Complejidad: O(#*Fichas* a desubicar)

PuntajePalabraSQFormanVertical(**in** f : `tupla<jugada : lista(fichas) ,turno : nat, hor: bool>` , **in** j : `juego`) $\longrightarrow$ `nat`

$Puntaje \leftarrow 0$
mientras ¬ vacia?(f.jugada) **hacer** ▷ O(Lmax) la jugada puede llegar a ser Lmax
$Puntaje \leftarrow Puntaje + PuntajePalabraQformaVertical(prim(F.jugada, F.turno, juego)$ ▷ O(Lmax)
$f.jugada \leftarrow fin(f)$
devolver Puntaje
Complejidad:O($Lmax^2$)
Explicación: Suma los puntajes de las palabras formadas verticalmente por la ocurrencia

Pre ≡ {x, y en rango dentro del tablero, l esta definido en claves(PuntajeXPalabra), }
Post ≡ {Devuelve el puntaje que forma una palabra horizontal partiendo del una coordenada dada por una ficha}

PuntajePalabraSQFormanHorizontal(**in** f : `tupla<jugada : lista(fichas) ,turno : nat, hor: bool>` , **in** j : `juego`) $\longrightarrow$ `nat`

$Puntaje \leftarrow 0$
mientras $\neg vacia?(f.jugada)$ **hacer** ▷ O(Lmax) la jugada puede llegar a ser Lmax
$Puntaje \leftarrow Puntaje + PuntajePalabraQformaHorizontal(prim(F.jugada, F.turno, juego)$ ▷ O(Lmax)
$f.jugada \leftarrow fin(f)$
devolver Puntaje
Complejidad:O($Lmax^2$)
Explicación: Suma los puntajes de las palabras formadas horizontalmente por la ocurrencia

Pre ≡ {x, y en rango dentro del tablero, l esta definido en claves(PuntajeXPalabra)}
Post ≡ {Devuelve el puntaje que forma una palabra horizontal partiendo del una coordenada dada por una ficha}

PUNTAJEPALABRAQFORMAHORIZONTAL(**in** f : `tupla<x : nat,y : nat, l: letra>`, **in** *turno* : `nat` , **in** j : `juego`) $\longrightarrow$ `nat`

$x \leftarrow (f.x)$
$y \leftarrow (f.y)$
$sig \leftarrow x + 1$
$ant \leftarrow x - 1$
$Puntaje \leftarrow j.varianteDelJuego.PuntajeXLetra[ord(f.l)]$
$i \leftarrow 0$
mientras $i < j.varianteDelJuego.Lmax$ **hacer** ▷ O(Lmax)
  si (sig < tamaño(t) $\wedge_L$ iHayLetra?(j.tablero, sig, y)) $\wedge$ TurnoTablero(j,sig,y) <= turno **entonces**
    $Puntaje \leftarrow Puntaje + j.varianteDelJuego.PuntajeXLetra[ord(iletra(j.tablero, sig, y)]$
    $sig \leftarrow sig + +$
  si (ant <tamaño(t)$\wedge_L$ iHayLetra?(j.tablero, ant, y))$\wedge$ TurnoTablero(j,ant,y) <= turno) **entonces**
    $Puntaje \leftarrow puntaje + j.varianteDelJuego.PuntajeXLetra[ord(iletra(j.tablero, ant, y)$ ▷ O(1)
    $ant \leftarrow ant - -$
  $i \leftarrow i + 1$
  devolver PUNTAJE

Complejidad:O(Lmax)
Explicación: Idea similar o casi igual a palabraQformaHorizontal pero en vez de hacer una lista y crear la palabra, suma los puntajes de la palabra formada Letra por Letra sin necesidad de crear la palabra. Usa el turnoGeneral en donde se ubico la ficha como ”historial” para no contar letras que fueron puestas posteriormente al turno jugado. ejemplo : El jugador 1 puso ” asombras” pero posteriormente el jugador 2 formó ´´asombraste´´, querremos que el puntaje del jugador 1 cuente solamente las fichas que estaban al momento de hacer su jugada, es decir no contará la t y la e.

Pre ≡ {x, y en rango dentro del tablero, l esta definido en claves(PuntajeXPalabra), turno menor estricto que la cantidad de jugadores}
Post ≡ {Devuelve el puntaje que forma una palabra vertical partiendo del una coordenada dada por una ficha}

PUNTAJEPALABRAQFORMAVERTICAL(**in** f : `tupla<x : nat,y : nat, l: letra>`, **in** *turno* : `nat` , **in** j : `juego`) $\longrightarrow$ `nat`

$x \leftarrow (f.x)$
$y \leftarrow (f.y)$
$sig \leftarrow y + 1$
$ant \leftarrow y - 1$
$Puntaje \leftarrow j.varianteDelJuego.PuntajeXLetra[ord(f.l)]$
$i \leftarrow 0$
mientras $i < j.varianteDelJuego.Lmax$ **hacer** ▷ O(Lmax)
  si (sig <tamaño(t)$\wedge_L$ iHayLetra?(j.tablero, x, sig))$\wedge_L$ TurnoTablero(j,x,sig)<= turno **entonces**
$Puntaje \leftarrow Puntaje + j.varianteDelJuego.PuntajeXLetra[ord(iletra(j.tablero, x, sig)]$
$sig \leftarrow sig + +$
  si (ant <tamaño(t)$\wedge$ iHayLetra?(j.tablero, x, ant)) $\wedge$ TurnoTablero(j,x,ant) <= turno) **entonces** ▷ $\Theta(1)$

$Puntaje \leftarrow Puntaje + j.varianteDelJuego.PuntajeXLetra[ord(iletra(j.tablero, x, ant)$ ▷ O(1)
$ant \leftarrow ant - -$
  i++
  devolver PUNTAJE

Complejidad:O(Lmax)

REPONER(**inout** j : `juego` , **in** cid : `Nat` , **in** o : `Ocurrencia`) $\longrightarrow$ `nat`

$it \leftarrow CrearIt(o)$
$res \leftarrow vacio()$
mientras HaySiguiente(it) **hacer**
  $fichasEnMano[cid][ord(siguiente(it))] \leftarrow fichasEnMano[cid][ord(siguiente(it))] - 1$
  $fichasEnMano[cid][ord(proximo(j.repositorio))] \leftarrow fichasEnMano[cid][ord(proximo(j.repositorio))] + 1$
  $res \leftarrow agregar(res, proximo(j.respositorio)$
  $j.repositorio \leftarrow desencolar(j.repositorio)$
  $it \leftarrow avanzar(it)$
  devolver res

Complejidad:O(Lmax) ya que esta acotado por el tamaño de o y este esta acotado por la palabra mas grande.
Explicacion: Repone cantidad de fichas igual al tamaño de o y las saca del repositorio

---

TURNOTABLERO(**in** j : `juego`, **in** x : `nat` ,**in** y : `nat`) $\longrightarrow$ `nat`

---

iTurnoTablero(j.tablero,x,y)
Complejidad:$O(1)$

PUNTAJEOCURRENCIA(**in** j : `Juego`, **in** o : `ocurrencia`) $\longrightarrow$ `nat`

$it \leftarrow crearIt(o)$
si eshorizontal?(o) **entonces**
  $puntaje \leftarrow PuntajePalabraQFormaHorizontal(siguiente(it), j.TurnoGral, j) +$
$PuntajePalabraSQFormanVertical(< o, j.TurnoGral, true >, j)$
else
  $puntaje \leftarrow PuntajePalabraQFormaVertical(siguiente(it), j.TurnoGral, j) +$
$PuntajePalabraSQFormanVertical(< o, j.turnoGral, false >, j)$
  devolver puntaje

Complejidad:O($Lmax^2$) ya que llama a puntajePalabraSQformanVertical y puntajePalabraSQformanHorizontal
Explicacion: Calcula el puntaje de una ocurrencia

---

2. Módulo Notificacion

2.1. Representación

Representacion de Notificacion

`notificacion` se representa con `estr`

donde `estr` es `tupla`(*tipoNotif*: `string`, *iDcliente*: `nat`
, *Empezar*: `nat` , *fichasParaReponer*: `Lista(ficha)` , *fichasUbicadas*: `ocurrencia`

)

Rep : estr $\longrightarrow$ bool

Rep(j) $\equiv$ true $\Longleftrightarrow$ pertenece?(tiposNotificaciones,e.tipoNotif)
donde tiposNotificaciones : tupla(string) = "IdCliente", "Empezar", "TurnoDe", "Ubicar", "Reponer", "SumaPuntos", "Mal"

Abs : estr e $\longrightarrow$ notif {Rep(e)}

Abs(e) $=_{obs}$ n: notif | π_1 datos(n) = e.tipoNotif $\wedge$ π_2 datos(n) = e.idCliente $\wedge$
π_3 datos(n) = e.Empezar $\wedge$ π_4 datos(n) = e.fichasParaReponer $\wedge$
π_5 datos(n) = e.fichasUbicadas

3. Módulo Servidor

3.1. Interfaz

usa: JUEGO, NAT, TUPLA, SECUENCIA, LETRA, OCURRENCIA, VARIANTE, IDCLIENTE

se explica con: SERVIDOR

géneros: `servidor`.

Operaciones básicas de Servidor

INICIARSERVIDOR(**in** k : `nat`,**in** v : `variante`, **in** r : `cola(letras)`) $\rightarrow$ res : `servidor`
Pre $\equiv$ {true}
Post $\equiv$ {res = nuevoServidor(k,v,r)}
Complejidad: $\mathcal{O}(N^2 + \Sigma \cdot K + F \cdot K)$
Descripción: Crea un servidor
Aliasing: Produce aliasing ya que pasa el repositorio y k por referencia

CONECTARCLIENTE(**in/out** s : `servidor`)
Pre $\equiv$ {s = s_0}
Post $\equiv$ {s = conectarCliente(s_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Conecta un cliente
Aliasing: No produce aliasing

CONSULTARNOTIFICACIONES(**in/out** s : `servidor`, **in** cid : `idCliente`) $\rightarrow$ res : `Lista(Notificacion)`
Pre $\equiv$ {cid <#conectados(s) $\wedge$ s=s_0}
Post $\equiv$ {s = consultar(s_0)}
Complejidad: $\mathcal{O}(n)$
Descripción: Consulta Notificaciones
Aliasing: No produce aliasing

RECIBIRMENSAJEC(**in/out** s : `servidor`, **in** cid : `idCliente`, **in** o : `ocurrencia`)
Pre $\equiv$ { s = s_0 $\wedge$ cid <#conectados(s)}
Post $\equiv$ { s = recibirMensaje(s_0,cid,o}
Complejidad: $\mathcal{O}(L_{max}^2 + m)$ donde L_{max} la longitud de la palabra mas larga, y m es la cantidad de fichas a ubicar
Descripción: Recibe un mensaje de un cliente
Aliasing: No produce aliasing

OBT#CLIENTESESPERADOS(**in** s : `servidor`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = #esperados(s)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de clientes esperados
Aliasing: Sin aliasing

#CLIENTESCONECTADOS (**in** s : `servidor`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ { res = #clientesconectados(s)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de clientes conectados
Aliasing: Sin aliasing

JUEGOJUGADOSV(**in** s : `servidor`) $\rightarrow$ res : `juego`
Pre $\equiv$ {true}
Post $\equiv$ {r = obs juego(s)}

Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el juego jugado
Aliasing: No crea aliasing

Fin Interfaz

3.2. Representación

Representación de Servidor

`servidor` se representa con `estr`

donde `estr` es `tupla`(*NotificacionesPorJugador*: `vector(cola(notificacion))`, *Juego*: `juego`
, *repositorio*: `cola(letra)` , *JugadoresConect*: `nat` , *JugadoresEsperados*: `nat`
, *Empezo?*: `bool`
, *vectorParaVerLasPrimerasNotificaciones*: `vector(bool)`)

Rep : estr $\longrightarrow$ bool

Rep(s) $\equiv$ true $\iff$ s.JugadoresConect $\leq$ s.JugadoresEsperados $\wedge$ longitud(s.NotificacionesPorJugador) = s.JugadoresEsperados $\wedge$ (s.JugadoresConect=s.JugadoresEsperados $\Rightarrow$ s.Empezo?) $\wedge$ longitud(s.vectorParaVerLasPrimerasNotificaciones) = s.JugadoresEsperados

Abs : estr e $\longrightarrow$ servidor {Rep(e)}

Abs(e) $=_{\text{obs}}$ s: servidor | e.JugadoresEsperados = #esperados(s) $\wedge$ e.JugadoresConect = #conectados(s) $\wedge$
(variante(e.Juego), e.repositorio) = configuracion(s) $\wedge$
e.Juego = juego(j) $\wedge$
($\forall$ i:nat)(i $\leq$ #conectados(s) $\Rightarrow$ notificaciones(s,i) = e.NotificacionesPorJugador[i])

3.3. Algoritmos

iIniciar_Servidor(**in** k : `nat`, **in** v : `variante`, **in** r : `cola(letra)`) $\longrightarrow$ `servidor`

```
1:  s.vectorParaVerLasPrimerasNotificaciones ← vacio()
2:  j ← 0
3:  mientras j<k  hacer                          ▷ O(P) donde P es la #Jugadores esperados
4:      agregarAtras(vectorParaVerLasPrimerasNotificaciones,false)
5:      j ← j + 1
6:  s.notificaciones ← vacia()
7:  s.jugadoresEsperados ← k
8:  s.jugadoresConect ← 0
9:  s.repositorio ← r                            ▷ O(1) ya que se pasa por referencia
10: s.Empezo? ← false
11: s.Turno ← 0
12: s.juego ← iIniciar_Juego(k, v, r)            ▷ O(N^2+ | Σ | K + FK)
13: i ← 0
14: mientras i<k hacer                           ▷ O(P) donde P es la #Jugadores esperados
15:     AgregarAtras(s.Notificaciones,vacia())
16:     i ← i + 1
    devolver s
    Complejidad: O(N^2+ | Σ | K + FK + P) = O(N^2+ | Σ | K + FK) ya que P es = K y este se acota
```

ConectarCliente(**in/out** s : `servidor`)

```
si s.jugadoresEsperados >s.jugadoresConect entonces
    s.jugadoresConect ← s.jugadoresConect + 1                    ▷ O(1)
    encolar(s.NotificacionesGral,(IdCliente,s.jugadoresConect,0,vacio(), vacio()))
si s.juegadoresEsperados=s.jugadoresConect entonces
    s.Empezo? ← true
Complejidad: O(1)
```

ConsultarNotificaciones(**in/out** s : `servidor`, **in** cid : `idCliente`) $\longrightarrow$ Lista(Notificacion)

```
res ← vacia()
si s.Empezo? ∧ s.vectorParaVerLasPrimerasNotificaciones[cid] = false entonces

    s.vectorParaVerLasPrimerasNotificaciones[cid]← true
5:  AgregarAtras(res,(Empezar,0,tamanoTablero(Infovariante(s.juego)),vacio(),vacio()))
    AgregarAtras(res,(Turno,0,0,vacio(),vacio()))
    mientras ¬ esVacia?(s.NotificacionesPorJugador[cid]  hacer
        AgregarAtras(res,proximo(s.NotificacionesPorJugador[cid])
                                                 ▷ O(n) donde n es la cantidad de notificaciones del cliente
10:     s.NotificacionesPorJugador[cid] ← desencolar(s.NotificacionesPorJugador[cid])
    devolver Res
Complejidad:O(N) la cantidad de notificaciones de dicho cliente
```

iRecibirMensajeC(**inout** s : `Servidor` ,**in** id : `idCliente` , **in** o : `ocurrencia`

si s.jugadoresConect == s.jugadoresEsperados ∧ iesJugadaValida(s.juego, o) ∧ s.turno mod s.jugadoresEsperados == id **entonces** ▷ O($Lmax^2$)
$s.juego \leftarrow$ `iUbicarConjJuego(s.juego, o)`
Reponer(s.Juego, id ,o)
encolar(s.NotifacionesPorJugador[s.jugadoresConect],(Reponer,cid,0,0,Reponer(o),vacio) ▷ O(| o |)
$s.Turno \leftarrow s.Turno + 1$
encolar(s.NotifacionesGral,(ubicar,id,o, vacio()))
`encolar(s.NotificacionesGral, SumaPuntos, id, puntaje(s.Juego,o), vacio, vacio)`
else `encolar(s.NotificacionesPorJugador[cid], "Mal", 0, 0, vacio, vacio)`
Complejidad:O($Lmax^2$+ | o | +1)

Obt#ClientesEsperados(**in** s : `servidor`) ⟶ `nat`

devolver s.juego.CantidadEsperados
Complejidad:O(1)

#ClientesConectados(**in** s : `servidor`) ⟶ `nat`

devolver s.JugadoresConect
Complejidad: O(1)

JuegoJugadoSv(**in** s : `servidor`) ⟶ `juego`

devolver s.juego
Complejidad: O(1)

4. Módulo Tablero

4.1. Interfaz

usa: BOOL, NAT, CONJUNTO, OCURRENCIA

se explica con: TABLERO

géneros: `tablero`.

Operaciones básicas de Tablero

TAMAÑO(**in** t : `tablero`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = tamaño(t)}
Complejidad: $\mathcal{O}(1)$
Descripción: Da el largo de un tablero nxn
Aliasing: No presenta aliasing

HAYLETRA?(**in** t : `tablero`, **in** x, y : `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {x < tamaño(t) $\wedge_L$ y < tamaño(t) $\wedge$ enTablero?(t,x,y)}
Post $\equiv$ {res = hayLetra?(t, x,y)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dada una casilla, dice si hay una letra
Aliasing: Sin aliasing

LETRA(**in** t : `tablero`, **in** x, y : `nat`) $\rightarrow$ res : `letra`
Pre $\equiv$ {(x < tamaño(t) $\wedge$ y < tamaño(t)) $\wedge_L$ hayLetra?(t,x,y)}
Post $\equiv$ {res = Letra(t,x,y)}
Complejidad: $\mathcal{O}(1 + copy(res))$
Descripción: Devuelva el valor guardado en una casilla del tablero
Aliasing: Sin aliasing

PONERLETRA(**in/out** t : `tablero`, **in** x, y : `nat`, **in** l : `letra`)
Pre $\equiv$ {t = t_0 $\wedge$ x < tamaño(t) $\wedge$ y < tamaño(t)) $\wedge_L$ $\neg$(HayLetra?(x,y) $\wedge$ enTablero?(t,x,y)}
Post $\equiv$ {t $=_{obs}$Ponerletra(t_0 ,x,y,l)}
Complejidad: $\mathcal{O}(1)$
Descripción: Pone Letra sobre el tablero
Aliasing: Provoca aliasing

NUEVOTABLERO(**in** n : `nat`) $\rightarrow$ res : `tablero`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$nuevoTablero(n)}
Complejidad: $\mathcal{O}(n^2)$
Descripción: Crea un tablero
Aliasing: Sin aliasing

ENTABLERO?(**in** t : `tab`,**in** x, y : `nat`) $\rightarrow$ res : `letra`
Pre $\equiv$ {true}
Post $\equiv$ { res $=_{obs}$x <tamaño(t) $\wedge$ y <tamaño(t))}
Complejidad: $\mathcal{O}(1)$
Descripción: Verifica que x e y esten dentro del tablero
Aliasing: Sin aliasing

PONERLETRAS(**in/out** t : `tablero`,**in** o : `ocurrencia`)
Pre ≡ {t= t_0 ∧ celdasLibres?(t, o) ∧ (∀ i , j : nat)(∀ l , l' : letra) ((⟨i, j, l) /*in* o ∧ ⟨i, j, l' ⟩/*in* o) ⇒ l = l') }
Post ≡ {t = ponerLetras(t_0,o)}
Complejidad: $\mathcal{O}(long(o))$
Descripción: Pone un conjunto de letras en el tablero con su correspondiente coordenada
Aliasing: Crea aliasing

VACIA?(**in/out** t : `tablero`,**in** x : `nat`,**in** y : `nat`) → res : `nat`
Pre ≡ {x <tamaño(t) ∧ y <tamaño(t)}
Post ≡ {res = ¬libre?(t,x,y)}
Complejidad: $\mathcal{O}(1)$
Descripción: Se fija si esa celda esta vacia
Aliasing: Sin aliasing

MODIFICARTURNO(**in/out** t : `tablero`,**in** x : `nat`,**in** y : `nat`, **in** $turno$: `nat`)
Pre ≡ {x < tamaño(t) ∧ y < tamaño(t) ∧ t = t_0}
Post ≡ {t[x][y].turno = turno}
Complejidad: $\mathcal{O}(1)$
Descripción: Guardar el turno (de juego) en el que fue puesta la ficha
Aliasing: Genera aliasing con el tablero

MODIFICARVACIA(**in/out** t : `tablero`,**in** x : `nat`,**in** y : `nat`, **in** $vacia$: `bool`)
Pre ≡ {x < tamaño(t) ∧ y < tamaño(t) ∧ t = t_0}
Post ≡ {libre?(t,x,y) = vacia}
Complejidad: $\mathcal{O}(1)$
Descripción: Cambia la componente de vacia? en esa casilla
Aliasing: Genera aliasing con el tablero

TURNOTABLERO(**in** t : `tablero`,**in** x : `nat`,**in** y : `nat`) → res : `nat`
Pre ≡ {x < tamaño(t) ∧ y < tamaño(t) ∧ HayLetra?(t,x,y)}
Post ≡ {res = ▷ buscar en el tablero el turno de esos valores de x,y y debe coincidir con res}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el turno en el que fue puesta esa ficha
Aliasing: Sin aliasing

Fin Interfaz

4.2. Representación

Representación de Tablero

`tablero` **se representa con** `vector(vector( Vacio?: bool , casilla: letra , Turno: nat))`

Rep : estr ⟶ bool
Rep(t) ≡ true ⟺ (∀ j:nat)(0 ≤ j <longitud(t) ⇒ longitud(t) = longitud(t[j])) ∧

Abs : estr e ⟶ tablero {Rep(e)}
Abs(e) =$_{obs}$ t: tablero | longitud(e) = tamaño(t) ∧
(∀ i:nat)(i ≤ longitud(e) ⇒
e[i][k].Vacio? = ¬ hayLetra?(t,i,k) ∧
¬e[i][k].Vacio? ⇒ $_L$ e[i][k].casilla = letra(t,i,k))

4.3. Algoritmos

iTamaño(**in** t : `tablero`) $\longrightarrow$ `nat`

```
1: n ← longitud(tablero)                                   ▷ O(1)
2: devolver N
   Complejidad:O(1)
```

Vacia?(**in** t : `tablero`, x : `Nat`, y : `Nat`) $\longrightarrow$ `bool`

```
   ¬iHayletra?(t,x,y)                                      ▷ O(1)
   Complejidad:O(1)
```

iHayLetra?(**in** t : `tablero`, x : `Nat`, y : `Nat`) $\longrightarrow$ `bool`

```
   devolver ¬ t[x][y].Vacio?                               ▷ O(1)
   Complejidad:O(1)
```

iLetra(**in** t : `tablero`, x : `Nat`, y : `Nat`) $\longrightarrow$ `string`

```
   devolver t.tablero[x][y]                                ▷ O(1)
   Complejidad:O(1)
```

iPonerLetra(**in/out** t : `tablero`, x , y : `Nat`), L : `letra`)

```
   t[x][y].letra ← L                                       ▷ O(1)
   j.tablero[x][y].Vacio? ← false                          ▷ O(1)
   Complejidad:O(1)
```

iNuevoTablero(**in** n : `nat`) $\longrightarrow$ `tablero`

```
   i ← 0
   j ← 0
   t ← []                                                  ▷ O(1)
   casillaVacia ← tupla(true,"a")
5: mientras i < n hacer                                    ▷ O(N)
       agregarAtras(t[i],[])
       mientras j < n hacer                                ▷ O(N)
           agregarAtras(t[i][j],casillaVacia)
           j++
10:    i++
   devolver T
   Complejidad:O(N^2)
```

ienTablero?(**in** t : `tablero`, x : `Nat`, y : `Nat`) $\longrightarrow$ `bool`

```
   res ← true
   longi ← iTamao(t)                                       ▷ O(1)
   si X ≥ LONGI ∨ Y ≥ LONGI entonces                       ▷ O(1)
       res ← false
   devolver RES
   Complejidad:O(1)
```

iModificarTurno(**in/out** t : `tablero`,**in** x : `nat`,**in** y : `nat`, **in** $turno$: `nat`

$t[x][y].turno \leftarrow turno$ ▷ O(1)
Complejidad:O(1)

in/out t : `tablero`,**in** x : `nat`,**in** y : `nat`, **in** $vacia$: `bool`

iModificarVacia(**in/out** t : `tablero`,**in** x : `nat`,**in** y : `nat`, **in** $vacia$: `bool`

$t[x][y].vacia \leftarrow vacia$ ▷ O(1)
Complejidad:O(1)

TurnoTablero(**in** t : `tablero`, **in** x : `nat` ,**in** y : `nat`) $\longrightarrow$ `nat`

devolver t[x][y].turno
Complejidad:$O(1)$

5. Módulo Variante

5.1. Interfaz

usa: BOOL, NAT, SECUENCIA, CONJUNTO, DICCIONARIO

se explica con: VARIANTE

géneros: `variante`.

Operaciones básicas de Variante

TAMAÑOTABLERO(**in** v : `variante`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = tamañoTablero(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Da el tamaño del tablero de la variante
Aliasing: No presenta aliasing

#FICHAS(**in** v : `Variante`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = #fichas(v)}
Complejidad: $\mathcal{O}(1)$
Descripción: Da la cantidad de fichas
Aliasing: Sin aliasing

PUNTAJELETRA (**in** v : `variante`, **in** l : `letra`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = puntajeLetra(v.l)}
Complejidad: $\mathcal{O}(1)$
Descripción: el valor de la letra
Aliasing: Sin aliasing

PALABRALEGITIMA?(**in** v : `variante`, **in** s : `secu(letra)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ palabraLegitima?(v,s)}
Complejidad: $\mathcal{O}(longitud(s))$
Descripción: Ve si una palabra es legitima
Aliasing: No provoca aliasing

NUEVAVARIANTE(**in** t: `nat`,**in** f: `nat` ,**in** PxL: `dicc(letra,nat)`,**in** PL: `diccTries(vector(letra),vector(letra)`) $\rightarrow$ res : `variante`
Pre $\equiv$ { 0 <t $\wedge$ 0 <f)}
Post $\equiv$ { res $=_{obs}$nuevaVariante(t,f,PxL,PL) }
Complejidad: $\mathcal{O}(Copy(Pxl) + Copy(PL))$
Descripción: Crea una nueva Variante
Aliasing: Produce aliasing ya que pasa t y f por referencia

Fin Interfaz

5.2. Representación

Representación de Variante

`variante` se representa con `estr`

donde `estr` es `tupla`(*tamaño*: `nat` , *#Fichas*: `nat` , *PuntajeXLetra*: `vector(nat)` , *PalsLegit*: `diccTries(vector(letra), vector(letra))` , *Lmax*: `nat`)

Rep : estr $\longrightarrow$ bool

Rep(v) $\equiv$ true $\Longleftrightarrow$
(∀ palabra:`vector(letra)`)(definido?(palabra,v.PalsLegit) ∧
(∀ i : nat)(0 ≤ i <longitud(v.PuntajeXLetra) ⇒
(∃k:nat)(0 ≤ k <longitud(palabra) ∧ $ord^{-}1$(i)= $ord^{-}1$ (palabra[k])))) ∧
(∀ palabra : lista(letra)) definido?(v.PalsLegit,palabra) ⇒ (longitud(palabra) ≤ Lmax)
∧ (∃ p: Lista(letra)) definido?(v.PalsLegit, p) ⇒ (Longitud(palabra) = Lmax)

Abs : estr e $\longrightarrow$ variante {Rep(e)}

Abs(e) =$_{obs}$ v: variante | (∀ i:nat)(∃ l: letra)(0 ≤ i<longitud(v.PuntajeXLetra) ∧ $ord^{-1}(i)$=l ⇒
puntajeLetra(v,l) = e.PuntajeXLetra[i] ∧
(∀ palabra:`Lista(letra)`)
(definido?(palabra,e.PalsLegit) = palabraLegitima?(v,palabra)) ∧
tamañoTablero(v) = e.tamaño ∧ #Fichas(v) = e.#Fichas

5.3. Algoritmos

I#FICHAS (**in** v : `variante`) $\longrightarrow$ `nat`

---

devolver V.#FICHAS
Complejidad: O(1)

IPUNTAJELETRA(**in** v : `variante`, **in** l : `letra`) $\longrightarrow$ `nat`

---

devolver V.PUNTAJEXLETRA[ORD(L)]
Complejidad:O(1)

IPALABRALEGITIMA?(**in** v : `variante`, **in** s : `lista(letra)`) $\longrightarrow$ `bool`

---

devolver DEFINIDO?(V.PALSLEGIT ,S)
Complejidad: O(| s |)

INUEVAVARIANTE(**in** t : `nat`, **in** f : `nat`, **in** PxL : `dicc`, **in** PL : `diccTries()`) $\longrightarrow$ `variante`

$res.tamano \leftarrow t$ ▷ O(1)
$res.\#Fichas \leftarrow f$ ▷ O(1)
$res.PuntajexLetra \leftarrow PxL$ ▷ O(Copy(PxL))
$res.PalsLegit \leftarrow PL$ ▷ O(Copy(PL))
devolver RES
Complejidad:O(Copy(PxL)+Copy(PL)

ITAMAÑOTABLERO(**in** v : `variante`) $\longrightarrow$ `nat`

---

devolver V.TAMAÑO

DICCPUNTAJEXLETRA(**in** v : `variante`) $\longrightarrow$ `DiccTries`

---

devolver V.PUNTAJEXLETRA

LMAX(**in** v : `variante`) $\longrightarrow$ `Nat`

---

devolver V.LMAX

---

6. Módulo Diccionario Tries(κ, σ)

El módulo Diccionario Tries provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo O(|k|) donde k es la clave buscar . Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definición se puede hacer en tiempo $O(1)$.
A su vez asumimos operaciones para acceder a diferentes partes de la clave del diccionario o sea iteradores : crearIt, siguiente, actual y haySiguiente? análogas a los vistos en el apunte de modulos basicos

6.1. Representación

Representación de DiccTries

`DiccTries` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero(nodo)` , *#claves*: `nat`)

`nodo` **se representa con** `( siguientes : vector(puntero(nodo) , definicion : puntero(`σ`) )`

Rep : estr $\longrightarrow$ bool

Rep(d) $\equiv$ true $\iff$ si la raiz $\neq$ NULL entonces no hay nodos inutiles, es decir, si un nodo es una hoja (todos los punteros de siguientes es NULL) entonces guarda un puntero a una definicion, si no es una hoja, existe algun hijo en siguientes que sea una hoja y guarde una definicion

Abs : estr e $\longrightarrow$ DiccTries {Rep(e)}

Abs(e) =$_{\text{obs}}$ d: DiccTries | ($\forall$*clave* :κ) (def?(clave,d) =$_{obs}$definido?(e,clave) $\wedge_L$
def?(c,d) $\Rightarrow$ $_L$ obtener(c,d) = significado(c,e)
Las claves que estan definidas en e y d coinciden, y para todas las claves definidas los significados son los mismos.

6.2. Interfaz

parámetros formales
géneros κ, σ

se explica con: DICCIONARIO(κ, σ)

géneros: `dicc`(κ, σ)

Operaciones básicas de diccionario

VACÍO() $\rightarrow$ *res* : `dicc`(κ, σ)
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio}
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

DEFINIDO?(**in** *d*: `dicc`(κ, σ), **in** *k*: κ) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(*d*, *k*)}
Complejidad: $\Theta(|k|)$ donde |k| es la longitud de la clave
Descripción: devuelve `true` si y sólo *k* está definido en el diccionario.
Aliasing: No presenta aliasing

SIGNIFICADO(**in** *d*: `dicc`(κ, σ), **in** *k*: κ) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(*d*, *k*)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ Significado(*d*, *k*))}
Complejidad: $\Theta(|k|)$ donde |k| es la longitud de la clave
Descripción: devuelve el significado de la clave *k* en *d*.
Aliasing: *res* es modificable si y sólo si *d* es modificable.
Requiere: No presenta aliasing

#CLAVES(**in** *d*: `dicc`(κ, σ)) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ #claves(*d*)}
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de claves del diccionario.
Aliasing: No presenta aliasing

DEFINIR(**in/out** *d*: `dicc`(κ, σ), `in` *clave* : κ, `in` *significado* : σ)
Pre $\equiv$ {d=d_0}
Post$\equiv$ {definir(clave,significado,d_0)=$_{obs}$d}
Complejidad: $\Theta(|k|)$ donde |k| es la longitud de la clave
Descripción: Define un par clave , valor en el diccionario
Aliasing: Alias(d)

BORRAR(**in/out** *d*: `dicc`(κ, σ), `in` *clave* : κ)
Pre $\equiv$ {def?(clave)d=d_0}
Post$\equiv$ {borrar(clave,significado,d_0)=$_{obs}$d}
Complejidad: $\Theta(|k|)$ donde |k| es la longitud de la clave
Descripción: Borra un significado
Aliasing: Presenta aliasing con d

6.3. Algoritmos

VACÍO () $\longrightarrow$ dicc(κ, σ)

```
d.raiz ← NULL
```

DEFINIDO? (**in** d : `dicc(`κ, σ`)`, **in** *clave* : κ) $\longrightarrow$ bool

```
actual ← d.raiz
res ← true
it ← CrearIt(clave)                                          ▷ Θ(1)
mientras HaySiguiente?(it) hacer
    si *actual.siguientes[actual(it)] ≠ NULL entonces
        actual ← *actual.siguientes[actual(it)]
        it ← siguiente(it)
    else
        devolver false
devolver res
```

SIGNIFICADO (**in** d : `dicc(`κ, σ`)`, **in** k : κ) $\longrightarrow \sigma$

```
actual ← d.raiz
it ← CrearIt(k)
mientras HaySiguiente(it) hacer
    actual ← *actual.siguientes[actual(it)]
    it ← siguiente(it)
devolver *(*actual.definicion)
```

DEFINIR (**in**/**out** d: `dicc(`κ, σ`),` `in` $clave$: κ`,in` $significado$: σ)

```
si ¬Definido(d,clave) entonces
    d.#claves ← d.#claves + 1
    actual ← d.raiz
    it ← CrearIt(k)
    padre ← NULL
    mientras actual ≠ NULL ∨ haySiguiente?(it) hacer
        padre ← actual
        actual ← *actual.siguientes[actual(it)]
        it ← Siguiente(it)
    si actual ≠ NULL entonces
        nuevo ← New(nodo)
        *(*nuevo.definicion) ← significado
        actual ← nuevo
    else
        si padre = NULL entonces
            nuevo ← New(nodo)
            *(*nuevo.definicion) ← significado
            raiz ← nuevo
        else
            actual ← padre
            mientras haySiguiente?(it) hacer
                nuevo ← New(nodo)
                *actual.siguientes[actual(it)] ← nuevo
                it ← Siguiente(it)
            *(*actual.definicion) ← significado
```

BORRAR (**in/out** d: `dicc(`κ, σ`),` `in` $clave$ `:` κ)

```
    actual ← d.raiz
    d.#claves ← d.#claves − 1
    it ← CrearIt(clave)
    camino ← vacia()                                              ▷ Θ(1)
5:  mientras HaySiguiente?(it) hacer
        camino ← AgregarAdelante(camino, actual)
        actual ← *actual.siguientes[actual(it)]
        it ← siguiente(it)
    #Hijos ← 0
10: i ← 0
    mientras i<long(*actual.siguientes) hacer
        si *actual.siguientes[i] ≠ NULL entonces
            #Hijos ← #Hijos + 1
        i ← i + 1
15: si #Hijos>0 entonces
        *actual.definicion ← NULL
    else
        caminoAux ← camino
        ultimoNodoUtil ← NULL
20:     mientras ¬Vacia?(camino) hacer
            si primero(camino) ≠ NULL entonces
                ultimoNodoUtil ← camino[j]
            camino ← fin(camino)
        mientras primero(caminoAux) ≠ ultimoNodoUtil hacer
25:         caminoAux ← fin(caminoAux)
        caminoAux ← fin(caminoAux)
        mientras ¬Vacia?(caminoAux) hacer
            delete(primero(caminoAux)
            caminoAux ← fin(caminoAux)
```