DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

SimCity

22 de junio de 2022 — Algoritmos y Estructuras de Datos 2

Integrante	LU	Correo electrónico
Ivo Córdoba	906/21	cordobaivo33@gmail.com
Ramiro Sanchez Posadas	796/21	rsanchezposadas2001@gmail.com
Martino Simon	374/21	martinosimon@gmail.com
Jonathan Bekenstein	348/11	jbekenstein@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Especificación

1.1. Renombres

TAD NIVEL ES NAT

géneros nivel

Fin TAD

TAD CONSTRUCCION ES STRING

géneros construccion

Fin TAD

TAD NOMBRE ES STRING

géneros nombre

Fin TAD

1.2. TAD Pos

Extendemos el TAD Tupla simplemente para definir renombres más semánticos para las operaciones de acceso a los componentes de la tupla.

TAD POS EXTIENDE TUPLA(NAT, NAT)

géneros pos

exporta x, y

otras operaciones

x : pos $\longrightarrow$ nat

y : pos $\longrightarrow$ nat

axiomas $\forall p$: pos

x(p) $\equiv$ π_1(p)

y(p) $\equiv$ π_2(p)

Fin TAD

1.3. TAD Mapa

TAD MAPA

igualdad observacional

$$(\forall m, m' : \text{mapa}) \left(m =_{\text{obs}} m' \iff \begin{pmatrix} \text{horizontales}(m) =_{\text{obs}} \text{horizontales}(m') \\ \wedge_{\text{L}} \text{ verticales}(m) =_{\text{obs}} \text{verticales}(m') \end{pmatrix} \right)$$

géneros mapa

exporta mapa, observadores, generadores

usa Nat, Conjunto

observadores básicos

horizontales : mapa $\longrightarrow$ conj(nat)

verticales : mapa $\longrightarrow$ conj(nat)

generadores

crear : conj(nat) $\times$ conj(nat) $\longrightarrow$ mapa

axiomas $\forall hs, vs$: conj(nat)

horizontales(crear(hs, vs)) $\equiv$ hs

verticales(crear(hs, vs)) $\equiv$ vs

Fin TAD

1.4. TAD SimCity

TAD SimCity

igualdad observacional

$$(\forall s, s' : \text{simcity}) \left(s =_{\text{obs}} s' \iff \begin{pmatrix} \text{mapa}(s) =_{\text{obs}} \text{mapa}(s') \\ \wedge\ \text{casas}(s) =_{\text{obs}} \text{casas}(s') \\ \wedge\ \text{comercios}(s) =_{\text{obs}} \text{comercios}(s') \\ \wedge\ \text{popularidad}(s) =_{\text{obs}} \text{popularidad}(s') \end{pmatrix} \right)$$

géneros simcity

exporta simcity, generadores, observadores, turnos, unionValida, construccionesValidas

usa Mapa, Construccion, Pos, Nivel, Bool, Nat, Diccionario, Conjunto

observadores básicos

mapa	: simcity	$\longrightarrow$	mapa
casas	: simcity	$\longrightarrow$	dicc(pos, nivel)
comercios	: simcity	$\longrightarrow$	dicc(pos, nivel)
popularidad	: simcity	$\longrightarrow$	nat

generadores

iniciar	: mapa	$\longrightarrow$	simcity	
avanzarTurno	: simcity s $\times$ dicc(pos $\times$ construccion) cs	$\longrightarrow$	simcity	{construccionesValidas(s, cs)}
unir	: simcity a $\times$ simcity b	$\longrightarrow$	simcity	{unionValida(a, b)}

otras operaciones

turnos	: simcity	$\longrightarrow$	nat	
construccionesValidas	: simcity $\times$ dicc(pos $\times$ construccion)	$\longrightarrow$	bool	
unionValida	: simcity $\times$ simcity	$\longrightarrow$	bool	
noSeSolapanConRios	: simcity $\times$ simcity	$\longrightarrow$	bool	
noSeSolapanMaxNivel	: simcity $\times$ simcity	$\longrightarrow$	bool	
posLibre	: simcity $\times$ pos	$\longrightarrow$	bool	
posOcupadas	: simcity	$\longrightarrow$	conj(pos)	
xOcupadas	: simcity	$\longrightarrow$	conj(nat)	
xOcupadasAux	: conj(pos)	$\longrightarrow$	conj(nat)	
yOcupadas	: simcity	$\longrightarrow$	conj(nat)	
yOcupadasAux	: conj(pos)	$\longrightarrow$	conj(nat)	
posConNivel	: simcity $\times$ nat	$\longrightarrow$	conj(pos)	
posConNivelAux	: simcity $\times$ nat $\times$ conj(pos)	$\longrightarrow$	conj(pos)	
nivel	: simcity s $\times$ pos p	$\longrightarrow$	nat	{p $\in$ posOcupadas(s)}
maxNivel	: simcity	$\longrightarrow$	nat	
maxNivelAux	: simcity $\times$ conj(pos)	$\longrightarrow$	nat	
subirNivel	: dicc(pos $\times$ nivel)	$\longrightarrow$	dicc(pos, nivel)	
adoptarNiveles	: simcity $\times$ dicc(pos $\times$ nivel)	$\longrightarrow$	dicc(pos, nivel)	
nuevasConstrucciones	: dicc(pos $\times$ construccion) $\times$ construccion	$\longrightarrow$	dicc(pos, nivel)	
unirConstrucciones	: dicc(pos $\times$ nivel) $\times$ dicc(pos $\times$ nivel)	$\longrightarrow$	dicc(pos, nivel)	
sacarCasasSolapadas	: dicc(pos $\times$ nivel) $\times$ dicc(pos $\times$ nivel)	$\longrightarrow$	dicc(pos, nivel)	
sacarComerciosSolapados	: dicc(pos $\times$ nivel) $\times$ dicc(pos $\times$ nivel)	$\longrightarrow$	dicc(pos, nivel)	

distanciaManhattan : pos × pos ⟶ nat
nivelManhattan : pos × dicc(pos × nivel) ⟶ nat

axiomas $\forall s, s1, s2$: simcity, $\forall p, p1, p2$: pos, $\forall ps$: conj(pos), $\forall c$: construccion, $\forall cs$: dicc(pos × construccion), $\forall ns$: dicc(pos × nivel), $\forall n$: nat

mapa(iniciar(m)) ≡ m
mapa(avanzarTurno(s, cs)) ≡ mapa(s)
mapa(unir(s1, s2)) ≡ crear(
horizontales(mapa(s1)) ∪ horizontales(mapa(s2)),
verticales(mapa(s1)) ∪ verticales(mapa(s2))
)

casas(iniciar(m)) ≡ vacio
casas(avanzarTurno(s, cs)) ≡ subirNivel(unirConstrucciones(
casas(s),
nuevasConstrucciones(cs, 'casa')
))
casas(unir(s1, s2)) ≡ sacarCasasSolapadas(
unirConstrucciones(casas(s1), casas(s2)),
unirConstrucciones(comercios(s1), comercios(s2))
)

comercios(iniciar(m)) ≡ vacio
comercios(avanzarTurno(s, cs)) ≡ subirNivel(
unirConstrucciones(
comercios(s),
adoptarNiveles(s, nuevasConstrucciones(cs, 'comercio'))
)
)
comercios(unir(s1, s2)) ≡ adoptarNiveles(
unir(s1, s2),
sacarComerciosSolapados(
unirConstrucciones(comercios(s1), comercios(s2)),
unirConstrucciones(casas(s1), casas(s2))
)
)

popularidad(iniciar(m)) ≡ 0
popularidad(avanzarTurno(s, cs)) ≡ popularidad(s)
popularidad(unir(s1, s2)) ≡ popularidad(s1) + popularidad(s2) + 1

turnos(iniciar(m)) ≡ 0
turnos(avanzarTurno(s, cs)) ≡ turnos(s) + 1
turnos(unir(s1, s2)) ≡ max(turnos(s1), turnos(s2))

construccionesValidas(s, cs) ≡ #(claves(cs)) > 0 ∧ claves(cs) ∩ posOcupadas(s) $=_{obs}$ ∅
unionValida(s1, s2) ≡ noSeSolapanConRios(s1, s2) ∧ noSeSolapanConRios(s2, s1)
$\wedge_L$ noSeSolapanMaxNivel(s1, s2) ∧ noSeSolapanMaxNivel(s2, s1)

noSeSolapanConRios(s1, s2) ≡ xOcupadas(s1) ∩ verticales(mapa(s2)) $=_{obs}$ ∅
∧ yOcupadas(s1) ∩ horizontales(mapa(s2)) $=_{obs}$ ∅
noSeSolapanMaxNivel(s1, s2) ≡ posConNivel(s1, maxNivel(s1)) ∩ posOcupadas(s2) $=_{obs}$ ∅

posLibre(s, p) ≡ p ∉ claves(casas(s)) ∧ p ∉ claves(comercios(s))
∧ x(p) ∉ verticales(mapa(s)) ∧ y(p) ∉ horizontales(mapa(s))
posOcupadas(s) ≡ claves(casas(s)) ∪ claves(comercios(s))

xOcupadas(s) ≡ xOcupadasAux(posOcupadas(s))

```
xOcupadasAux(ps)  ≡ if vacio?(ps) then
                      ∅
                    else
                      Ag(x(dameUno(ps)), xOcupadasAux(sinUno(ps)))
                    fi
yOcupadas(s)      ≡ yOcupadasAux(posOcupadas(s))
yOcupadasAux(ps)  ≡ if vacio?(ps) then
                      ∅
                    else
                      Ag(y(dameUno(ps)), yOcupadasAux(sinUno(ps)))
                    fi

posConNivel(s, n)          ≡ posConNivelAux(s, n, posOcupadas(s))
posConNivelAux(s, n, ps)   ≡ if vacio?(ps) then
                               ∅
                             else
                               if nivel(s, dameUno(ps)) =obs n then
                                 Ag(dameUno(ps), posConNivelAux(s, n, sinUno(ps)))
                               else
                                 posConNivelAux(s, n, sinUno(ps))
                               fi
                             fi

nivel(s, p)  ≡ if p ∈ claves(casas(s)) then
                 obtener(p, casas(s))
               else
                 obtener(p, comercios(s))
               fi

maxNivel(s)          ≡ maxNivelAux(s, posOcupadas(s))
maxNivelAux(s, ps)   ≡ if vacio?(ps) then
                         0
                       else
                         max(nivel(s, dameUno(ps)), maxNivelAux(s, sinUno(ps)))
                       fi

subirNivel(ns)  ≡ if vacio?(claves(ns)) then
                    vacio
                  else
                    definir(
                      dameUno(claves(ns)),
                      obtener(dameUno(claves(ns))) + 1,
                      subirNivel(borrar(dameUno(claves(ns)), ns))
                    )
                  fi

adoptarNiveles(s, ns)  ≡ if vacio?(claves(ns)) then
                           vacio
                         else
                           definir(
                             dameUno(claves(ns)),
                             max(
                               obtener(dameUno(claves(ns)), ns),
                               nivelManhattan(dameUno(claves(ns)), casas(s))
                             ),
                             adoptarNiveles(s, borrar(dameUno(claves(ns)), ns))
                           )
                         fi
```

```
nuevasConstrucciones(cs, c) ≡ if vacio?(claves(cs)) then
                                 vacio
                              else
                                 if obtener(dameUno(claves(cs)), cs) =obs c then
                                 definir(
                                      dameUno(claves(cs)),
                                      0,
                                      nuevasConstrucciones(borrar(dameUno(claves(cs)), cs), c)
                                 )
                                 else
                                 nuevasConstrucciones(borrar(dameUno(claves(cs)), cs), c)
                                 fi
                              fi

unirConstrucciones(ns1, ns2) ≡ if vacio?(claves(ns2)) then
                                 ns1
                               else
                                 unirConstrucciones(
                                      definir(
                                           dameUno(claves(ns2)),
                                           if def?(dameUno(claves(ns2)), ns1) then
                                                max(
                                                     obtener(dameUno(claves(ns2)), ns1),
                                                     obtener(dameUno(claves(ns2)), ns2)
                                                )
                                           else
                                                obtener(dameUno(claves(ns2)), ns2)
                                           fi,
                                           ns1
                                      ),
                                      borrar(dameUno(claves(ns2)), ns2)
                                 )
                               fi

sacarCasasSolapadas(ns1, ns2) ≡ if vacio?(claves(ns2)) then
                                   ns1
                                 else
                                   if def?(dameUno(claves(ns2)), ns1)
                                   ∧L obtener(dameUno(claves(ns2)), ns1)
                                   ≤ obtener(dameUno(claves(ns2)), ns2) then
                                      sacarCasasSolapadas(
                                           borrar(dameUno(claves(ns2)), ns1),
                                           borrar(dameUno(claves(ns2)), ns2)
                                      )
                                   else
                                      sacarCasasSolapadas(
                                           ns1,
                                           borrar(dameUno(claves(ns2)), ns2)
                                      )
                                   fi
                                 fi
```

```
sacarComerciosSolapados(ns1, ns2) ≡ if vacio?(claves(ns2)) then
                                        ns1
                                    else
                                        if def?(dameUno(claves(ns2)), ns1)
                                        ∧L obtener(dameUno(claves(ns2)), ns1)
                                        < obtener(dameUno(claves(ns2)), ns2) then
                                            sacarComerciosSolapados(
                                                borrar(dameUno(claves(ns2)), ns1),
                                                borrar(dameUno(claves(ns2)), ns2)
                                            )
                                        else
                                            sacarComerciosSolapados(
                                                ns1,
                                                borrar(dameUno(claves(ns2)), ns2)
                                            )
                                        fi
                                    fi

distanciaManhattan(p1, p2) ≡ |x(p1) - x(p2)| + |y(p1) - y(p2)|
nivelManhattan(p, ns)      ≡ if vacio?(claves(ns)) then
                                0
                             else
                                if distanciaManhattan(p, dameUno(claves(ns))) = 3 then
                                max(
                                    obtener(dameUno(claves(ns)), ns),
                                    nivelManhattan(p, borrar(dameUno(claves(ns)), ns))
                                )
                                else
                                nivelManhattan(p, borrar(dameUno(claves(ns)), ns))
                                fi
                             fi
```

Fin TAD

1.5. TAD Servidor

TAD SERVIDOR

igualdad observacional

$(\forall srv, srv' : \text{servidor})\ (srv =_{\text{obs}} srv' \iff (\text{ciudades(srv)} =_{\text{obs}} \text{ciudades(srv')}))$

géneros servidor

exporta servidor, generadores, observadores

usa SimCity, Mapa, Pos, Nat, Bool, Construccion, nombre, Conjunto, Diccionario

observadores básicos

ciudades : servidor $\longrightarrow$ dicc(nombre, simcity)

generadores

nuevoServidor : $\longrightarrow$ servidor

nuevaCiudad : servidor srv × nombre n × mapa m $\longrightarrow$ servidor $\{\neg$ nombreEnUso(srv, n)$\}$

avanzarTurno : servidor srv × nombre n × dicc(pos × construccion) cs $\longrightarrow$ servidor
$\left\{\begin{array}{l}\text{nombreEnUso(srv, n)} \wedge \neg(\exists\ \text{m: nombre})(\text{n} \in \text{uniones(srv, m)}) \\ \wedge_{\text{L}}\ \text{construccionesValidas(dameCiudad(srv, n), cs)}\end{array}\right\}$

unirCiudades : servidor srv × nombre $n1$ × nombre $n2$ $\longrightarrow$ servidor
$\left\{\begin{array}{l}\text{nombreEnUso(srv, n1)} \wedge \text{nombreEnUso(srv, n2)} \wedge \text{n1} \neq \text{n2} \\ \wedge\ \text{n1} \notin \text{uniones(srv, n2)} \wedge \neg(\exists\ \text{m: nombre})(\text{n1} \in \text{uniones(srv, m)}) \\ \wedge_{\text{L}}\ \text{unionValida(dameCiudad(srv, n1), dameCiudad(srv, n2))}\end{array}\right\}$

otras operaciones

nombreEnUso : servidor × nombre $\longrightarrow$ bool

dameCiudad : servidor srv × nombre n ⟶ simcity {nombreEnUso(srv, n)}
uniones : servidor srv × nombre n ⟶ conj(nombre)

axiomas $\forall srv1, srv2$: servidor, $\forall n, n1, n2$: nombre, $\forall m$: mapa, $\forall p$: pos

ciudades(nuevoServidor()) ≡ vacio
ciudades(nuevaCiudad(srv, n, m)) ≡ definir(n, iniciar(m), ciudades(srv))
ciudades(avanzarTurno(srv, n, cs)) ≡ definir(
    n,
    avanzarTurno(dameCiudad(srv, n), cs),
    ciudades(srv)
)
ciudades(unirCiudades(srv, n1, n2)) ≡ definir(
    n1,
    unir(dameCiudad(srv, n1), dameCiudad(srv, n2)),
    ciudades(srv)
)

nombreEnUso(srv, n) ≡ n ∈ claves(ciudades(srv))
dameCiudad(srv, n) ≡ obtener(n, ciudades(srv))

uniones(nuevoServidor(), n) ≡ ∅
uniones(nuevaCiudad(srv, n1, m), n) ≡ uniones(srv, n)
uniones(avanzarTurno(srv, n1, cs), n) ≡ uniones(srv, n)
uniones(unirCiudades(srv, n1, n2), n) ≡ **if** n $=_{\text{obs}}$ n1 **then**
    Ag(n2, uniones(srv, n)) ∪ uniones(srv, n2)
else
    uniones(srv, n)
fi

Fin TAD

2. Módulos de referencia

2.1. Módulo Mapa

Interfaz

se explica con: MAPA

géneros: `mapa`

Operaciones básicas

CREARMAPA(**in** hs: `conj(nat)`, **in** vs: `conj(nat)`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} crear(hs, vs)\}$
Complejidad: $O(copy(hs) + copy(vs))$
Descripción: Crea un nuevo mapa con los ríos horizontales y verticales provistos.

HORIZONTALES(**in** m: `mapa`) $\rightarrow$ res : `conj(nat)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} horizontales(m)\}$
Complejidad: $O(1)$
Descripción: Devuelve los ríos horizontales, dando su ubicación en el eje vertical.
Aliasing: Genera aliasing, devuelve una referencia no modificable.

VERTICALES(**in** m: `mapa`) $\rightarrow$ res : `conj(nat)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} verticales(m)\}$
Complejidad: $O(1)$
Descripción: Devuelve los ríos verticales, dando su ubicación en el eje horizontal.
Aliasing: Genera aliasing, devuelve una referencia no modificable.

Representación

Un mapa contiene rios infinitos horizontales y verticales. Los ríos se representan como conjuntos lineales de naturales que indican la posición en los ejes de los ríos.

`mapa` **se representa con** `estr`

donde `estr` es `tupla`(*horizontales*: `conj(nat)`, *verticales*: `conj(nat)`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ true

Abs : estr e $\longrightarrow$ mapa {Rep(e)}
Abs(e) $=_{\text{obs}}$ m: mapa | horizontales(m) $=_{\text{obs}}$ e.horizontales $\wedge$ verticales(m) $=_{\text{obs}}$ e.verticales

Algoritmos

CrearMapa(**in** hs: `conj(nat)`, **in** vs: `conj(nat)`) $\rightarrow$ **out** res: `mapa`
1: res.horizontales $\leftarrow$ hs ▷ $O(copy(hs))$
2: res.verticales $\leftarrow$ vs ▷ O($copy(vs)$)
Complejidad: $O(copy(hs) + copy(vs))$

Horizontales(**in** e : `estr`) → **out** res : `conj(nat)`
1: res ← e.horizontales ▷ $O(1)$
Complejidad: $O(1)$
Pues se devuelven los ríos horizontales por referencia no modificable.

Verticales(**in** e : `estr`) → **out** res : `conj(nat)`
1: res ← e.verticales ▷ $O(1)$
Complejidad: $O(1)$
Pues se devuelven los ríos verticales por referencia no modificable.

2.2. Módulo SimCity

Interfaz

se explica con: SimCity

géneros: `simcity`

Operaciones básicas

NuevoSimCity(**in** m : `mapa`) → res : `simcity`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} iniciar(m)\}$
Complejidad: $O(copy(m))$
Descripción: Crea un nuevo SimCity con el mapa provisto.

AvanzarTurno(**in/out** s : `simcity`, **in** cs : `dicc(pos, construccion)`)
Pre ≡ $\{s_0 = s \land construccionesValidas(s, cs)\}$
Post ≡ $\{s =_{\text{obs}} avanzarTurno(s_0, cs)\}$
Complejidad: $O(\#cs)$
Descripción: Avanza un turno en la instancia s construyendo las nuevas casas y comercios en las posiciones indicadas. Como éstas se guardan en diccionarios lineales, agregar nuevas construcciones solo requiere definir rápido las nuevas posiciones de las casas y comercios. Si se agrega una única casa o comercio, $\#cs = 1$ y por lo tanto la complejidad de toda la operación resulta $O(1)$.

Unir(**in/out** a : `simcity`, **in** b : `simcity`)
Pre ≡ $\{a_0 = a \land unionValida(a, b)\}$
Post ≡ $\{a =_{\text{obs}} unir(a_0, b)\}$
Complejidad: $O(1)$
Descripción: Une la instancia b con la instancia a. Esta función no ejecuta propiamente dicha la unión, sino que simplemente agrega atrás de una lista enlazada de uniones una referencia a la instancia b. Por eso tiene complejidad O(1). Los efectos de la unión recién se procesan al observar las casas, comercios, etc.
Aliasing: Se guarda una referencia a la instancia b que fue unidad a la instancia a. Recordemos que a partir de ahora la instancia b ya no puede ser modificada.

Mapa(**in** s : `simcity`) → res : `mapa`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} mapa(s)\}$
Complejidad: $O(p^2 * rs)$ donde p es la popularidad de s, rs es la máxima cantidad de ríos horizontales y verticales de alguna unión (directa o indirecta), sin considerar los ríos que obtiene por sus propias uniones.
Descripción: Devuelve el mapa de la instancia s contemplando todas las uniones realizadas.
Aliasing: No genera aliasing ya que no devuelve el mapa original de la instancia s, sino que construye uno nuevo sumando todos los ríos de todas sus uniones.

Casas(**in** s : `simcity`) → res : `dicc(pos, nivel)`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} casas(s)\}$
Complejidad: $O(p^2 * (maxCasas^2 + maxComercios^2))$
Descripción: Devuelve las posiciones de todas las casas de la instancia s (contemplando sus uniones) mapeando

a su respectivo nivel.
Aliasing: No genera aliasing ya que no devuelve el diccionario de casas de la instancia s, sino que construye uno nuevo sumando todas las casas de todas sus uniones.

COMERCIOS(**in** s: `simcity`) $\rightarrow$ res : `dicc(pos, nivel)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} comercios(s)\}$
Complejidad: $O(p^2 * (maxCasas^2 + maxComercios^2))$
Descripción: Devuelve las posiciones de todos los comercios de la instancia s (contemplando sus uniones) mapeando a su respectivo nivel.
Aliasing: No genera aliasing ya que no devuelve el diccionario de comercios de la instancia s, sino que construye uno nuevo sumando todos los comercios de todas sus uniones.

POPULARIDAD(**in** s: `simcity`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} popularidad(s)\}$
Complejidad: $O(1)$
Descripción: Devuelve la popularidad (cantidad de uniones directas e indirectas) de la instancia s.

TURNOS(**in** s: `simcity`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} turnos(s)\}$
Complejidad: $O(1)$
Descripción: Devuelve el turno actual (antigüedad) de la instancia s.

Representación

Un SimCity contiene un mapa y construcciones de dos tipos: casas y comercios, además de guardar la cantidad de turnos, popularidad y uniones realizadas.

El turno actual de la instancia se trackea con 2 variables: turnos y maxTurnos. La primera, turnos, es la cantidad de turnos transcurridos desde el inicio del SimCity sin contemplar las posibles uniones realizadas. Por otro lado, maxTurnos hace referencia a lo que llamamos la antigüedad del SimCity. Es decir, la cantidad máxima de turnos transcurridos entre esta instancia y todas sus uniones (directas e indirectas). Cada vez que se avanza de turno, se incrementan ambas variables. Esto lo necesitamos por dos razones: para poder calcular correctamente los niveles de las construcciones cuando se realizan uniones, y para no tener que modificar las instancias que fueron unidas (ya que al unir un SimCity se guarda una referencia al mismo pero necesitamos seguir subiendo de nivel sus construcciones).

La popularidad es un contador que indica la cantidad de uniones realizadas en la instancia (tanto directas como indirectas). Se actualiza cada vez que se realiza una unión.

Las casas y comercios son guardados en la estructura como diccionarios que mapean las posiciones de las construcciones con el turno en el que fueron construidos (se usa el turno real de la instancia). Estos diccionarios se implementan con listas enlazadas. Al agregar nuevas construcciones, se pide como precondición que la posición esté libre, es decir, la clave no está definida, y por lo tanto definir nuevas claves tiene complejidad $O(1)$ si se hace de forma rápida.

Las uniones realizadas se guardan en una lista enlazada de tuplas. La primer componente de la tupla es una referencia al SimCity unido, y la segunda es el turno en el que se unió dicho SimCity (se usa el turno real de la instancia a donde se une). Guardamos esta información ya que luego necesitamos poder calcular correctamente los niveles de las construcciones unidas. La unión tiene complejidad $O(1)$ ya que lo único que se hace es agregar atrás de la lista la nueva tupla que representa la unión.

Resolver los conflictos, calcular los niveles de las construcciones y generar el mapa a partir de todas las uniones sucede cuando se observa la instancia. Optimizamos la estructura para poder modificar la instancia de forma eficiente, a expensas de tener una mayor complejidad cuando se quiere observar dicha instancia.

`simcity` **se representa con** `estr`

donde `estr` es `tupla(`*mapa*: `mapa`,
turnos[1]: `nat`,
maxTurnos[2]: `nat`,
popularidad: `nat`,
casas: `diccLineal(pos, nat)`,
comercios: `diccLineal(pos, nat)`,
uniones: `lista(union))`

donde `union` es `tupla(`*simcity*: `puntero(simcity)`, *turnosUnion*: `nat`)

1. Este el turno real de la instancia.
2. Este es el turno máximo entre la instancia y todas sus uniones (directas e indirectas). Es decir, indica la antigüedad.

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ (
// Validación de las posiciones de las casas y comercios propios.
(∀p: pos)(
(p ∈ claves(e.casas) ∨ p ∈ claves(e.comercios) $\Rightarrow_{\text{L}}$ (
// Nuestras construcciones no se pisan con nuestros ríos.
x(p) ∉ verticales(e.mapa) ∧ y(p) ∉ horizontales(e.mapa)
// Ni con los ríos de las uniones.
∧ (∀i: nat)(0 ≤ i < long(e.uniones) $\Rightarrow_{\text{L}}$(
x(p) ∉ verticales(mapa(*e.uniones[i].simcity))
∧ y(p) ∉ horizontales(mapa(*e.uniones[i].simcity))
))
))
∧ (p ∈ claves(e.casas) $\Rightarrow_{\text{L}}$ (
// Nuestras casas no se pisan con nuestros comercios.
p ∉ claves(e.comercios)
// Fue construida en un turno válido.
∧ obtener(p, e.casas) ≤ e.turnos
))
∧ (p ∈ claves(e.comercios) $\Rightarrow_{\text{L}}$ (
// Nuestros comercios no se pisan con nuestras casas.
p ∉ claves(e.casas)
// Fue construido en un turno válido.
∧ obtener(p, e.comercios) ≤ e.turnos
))
)
// Existe al menos una casa y/o comercio de nivel máximo (es decir, se construyó en el turno 0).
∧ (∃p: pos)(
(p ∈ claves(e.casas) $\Rightarrow_{\text{L}}$ obtener(p, e.casas) = 0)
∨ (p ∈ claves(e.comercios) $\Rightarrow_{\text{L}}$ obtener(p, e.comercios) = 0)
)
// Validación de las uniones.
∧ (∀i: nat)(0 ≤ i < long(e.uniones) $\Rightarrow_{\text{L}}$(
// El turno de la instancia unida es menor o igual que el turno máximo entre todas las uniones.
turnos(*e.uniones[i].simcity) ≤ e.maxTurnos
// Fue unido en un turno válido.
∧ e.uniones[i].turnosUnion ≤ e.turnos
∧ (∀p: pos)(
p ∈ claves(casas(*e.uniones[i].simcity)) ∨ p ∈ claves(comercios(*e.uniones[i].simcity)) $\Rightarrow_{\text{L}}$ (
// Las construcciones de las uniones no se pisan con nuestros ríos.
x(p) ∉ verticales(e.mapa) ∧ y(p) ∉ horizontales(e.mapa)
// Ni tampoco se pisan con los ríos de otras uniones.
∧ (∀j: nat)(0 ≤ j < long(e.uniones) ∧ j ≠ i $\Rightarrow_{\text{L}}$(
x(p) ∉ verticales(mapa(*e.uniones[j].simcity))
∧ y(p) ∉ horizontales(mapa(*e.uniones[j].simcity))

```
                )
            )
        )
    ))
    // El turno de esta instancia es menor o igual que el turno máximo entre todas las uniones.
    ∧ e.turnos ≤ e.maxTurnos
    // La popularidad es la cantidad de uniones directas e indirectas.
    ∧ e.popularidad =obs long(e.uniones) + ∑_{i=0}^{long(e.uniones)−1} popularidad(*e.uniones[i].simcity)
)

Abs : estr e ⟶ simcity                                                    {Rep(e)}
Abs(e) =obs s: simcity |
    mapa(s) =obs e.mapa
    ∧ turnos(s) =obs e.maxTurnos
    ∧ popularidad(s) =obs e.popularidad
    ∧ (∀p: pos)(
        // La posición corresponde a una casa en la instancia abstracta s.
        (p ∈ claves(casas(s)) ⇒L (
            // La casa es propia...
            (p ∈ claves(e.casas) ∧L obtener(p, casas(s)) =obs e.turnos - obtener(p, e.casas))
            // O vino de una unión.
            ∨ (∃i: nat)(0 ≤ i < long(e.uniones) ∧L p ∈ claves(casas(*e.uniones[i].simcity)) ∧L (
                obtener(p, casas(s)) =obs (
                    (e.turnos - e.uniones[i].turnosUnion) + obtener(p, casas(*e.uniones[i].simcity))
                )
            ))
        ))
        // La posición corresponde a un comercio en la instancia abstracta s.
        (p ∈ claves(comercios(s)) ⇒L (
            // El comercio es propio...
            (p ∈ claves(e.comercios) ∧L obtener(p, comercios(s)) =obs e.turnos - obtener(p, e.comercios))
            // O vino de una unión.
            ∨ (∃i: nat)(0 ≤ i < long(e.uniones) ∧L p ∈ claves(comercios(*e.uniones[i].simcity)) ∧L (
                obtener(p, comercios(s)) =obs (
                    (e.turnos - e.uniones[i].turnosUnion) + obtener(p, comercios(*e.uniones[i].simcity))
                )
            ))
        ))
    )
```

Algoritmos

NuevoSimCity(**in** m : `mapa`) → **out** res : `simcity`

```
1: res.mapa ← m                                   ▷ O(copy(m))
2: res.casas ← Vacio()                            ▷ O(1)
3: res.comercios ← Vacio()                        ▷ O(1)
4: res.turnos ← 0                                 ▷ O(1)
5: res.maxTurnos ← 0                              ▷ O(1)
6: res.popularidad ← 0                            ▷ O(1)
7: res.uniones ← Vacia()                          ▷ O(1)
```

Complejidad: $O(copy(m))$

AvanzarTurno(**in/out** s: `simcity`, **in** cs: `dicc(pos, construccion)`)

```
1:  it ← CrearIt(cs)                                              ▷ O(1)
2:  while HaySiguiente(it) do                                     ▷ O(#cs)
3:      pos ← SiguienteClave(it)                                  ▷ O(1)
4:      construccion ← SiguienteSignificado(it)                   ▷ O(1)
5:      if construccion = 'casa' then
6:          DefinirRapido(s.casas, pos, s.turnos)                 ▷ O(1)
7:      else if construccion = 'comercio' then
8:          DefinirRapido(s.comercios, pos, s.turnos)             ▷ O(1)
9:      end if
10:     Avanzar(it)                                               ▷ O(1)
11: end while
12: s.turnos ← s.turnos + 1                                       ▷ O(1)
13: s.maxTurnos ← s.maxTurnos + 1                                 ▷ O(1)
```

Complejidad: $O(\#cs)$

Podemos usar DefinirRapido pues por precondición las nuevas construcciones están en posiciones libres.

Unir(**in/out** a: `simcity`, **in** b: `simcity`)

```
1: AgregarAtras(a.uniones, ⟨ &b, a.turnos ⟩)                      ▷ O(1)
2: a.maxTurnos ← max(a.maxTurnos, turnos(b))                      ▷ O(turnos(b))
3: a.popularidad ← a.popularidad + popularidad(b) + 1             ▷ O(popularidad(b))
```

Complejidad: $O(turnos(b)) + O(popularidad(b)) = O(1)$

Mapa(**in** s: `simcity`) → **out** res: `mapa`

Construye el mapa de la instancia teniendo en cuenta las uniones realizadas.

```
1:  hs ← Copiar(Horizontales(s.mapa))                             ▷ O(copy(Horizontales(s.mapa)))
2:  vs ← Copiar(Verticales(s.mapa))                               ▷ O(copy(Verticales(s.mapa)))
3:  itUniones ← CrearIt(s.uniones)                                ▷ O(1)
4:  while HaySiguiente(itUniones) do                              ▷ O(p)
5:      sUnion ← *Siguiente(itUniones).simcity                    ▷ O(1)
6:      sMapa ← Mapa(sUnion)                                      ▷ O(Mapa(sUnion))
7:      UnirConjuntos(hs, Horizontales(sMapa))                    ▷ O(#Horizontales(sMapa))
8:      UnirConjuntos(vs, Verticales(sMapa))                      ▷ O(#Verticales(sMapa))
9:      Avanzar(itUniones)                                        ▷ O(1)
10: end while
11: res ← CrearMapa(hs, vs)                                       ▷ O(copy(hs) + copy(vs))
```

Complejidad: $O(\sum_{i=0}^{p} i * rs) = O(p^2 * rs)$

Si consideramos las uniones como nodos de un árbol tipo rosetree (donde cada nodo puede tener una cantidad arbitraria de hijos, o sea, uniones), la popularidad p del SimCity nos indica la cantidad total de nodos. Para obtener el mapa de forma tal que incluya todos los ríos de todas las uniones, necesitamos iterar p veces (no necesariamente se itera p veces en un único llamado a Mapa, sino que sería la cantidad total de iteraciones entre todos los llamados recursivos a Mapa).

Los casos base de la recursión es cuando no hay uniones. En estos casos, no se hace ninguna iteración (ni llamados recursivos) y la complejidad de Mapa esta dada únicamente por la copia de los ríos, lo cual se hace 2 veces. En ese caso la complejidad resulta: $O(2(copy(hs) + copy(vs))) = O(copy(hs) + copy(vs))$.

Por otro lado, consideramos rs como la máxima cantidad de ríos horizontales y verticales de alguna unión (directa o indirecta), sin considerar los ríos que obtiene por sus propias uniones.

De esta forma 'aplanamos' la recursión como si fuese un único ciclo de p iteraciones (sin llamados recursivos), en donde en el peor caso, hay que insertar rs ríos a los conjuntos hs y/o vs.

Esta lógica para calcular la complejidad la reutilizamos en otros algoritmos.

Casas(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Construye el diccionario de casas de la instancia teniendo en cuenta las uniones realizadas. Resuelve los conflictos y calcula el nivel correcto para las casas resultantes.

```
1: res ← CasasPropias(s)                                   ▷ O(#s.casas)
2: casasUniones ← CasasUniones(s)                          ▷ O(p² * maxCasas²)
3: UnirConstrucciones(res, casasUniones)                   ▷ O(#res * #casasUniones)
4: comercios ← ComerciosPropios(s)                         ▷ O(#s.comercios)
5: comerciosUniones ← ComerciosUniones(s)                  ▷ O(p² * maxComercios²)
6: UnirConstrucciones(comercios, comerciosUniones)         ▷ O(#comercios * #comerciosUniones)
7: SacarCasasSolapadas(res, comercios)                     ▷ O(#res * #comercios)
```

Complejidad: $O(p^2 * (maxCasas^2 + maxComercios^2))$

Comercios(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Construye el diccionario de comercios de la instancia teniendo en cuenta las uniones realizadas. Resuelve los conflictos y calcula el nivel correcto para los comercios resultantes, aplicando la regla de adopción de nivel por la distancia manhattan.

```
1: res ← ComerciosPropios(s)                               ▷ O(#s.comercios)
2: comerciosUniones ← ComerciosUniones(s)                  ▷ O(p² * maxComercios²)
3: UnirConstrucciones(res, comerciosUniones)               ▷ O(#res * #comerciosUniones)
4: casas ← CasasPropias(s)                                 ▷ O(#s.casas)
5: casasUniones ← CasasUniones(s)                          ▷ O(p² * maxCasas²)
6: UnirConstrucciones(casas, casasUniones)                 ▷ O(#casas * #casasUniones)
7: AdoptarNiveles(res, casas)                              ▷ O(#res * (#res + #casas))
8: SacarComerciosSolapados(res, casas)                     ▷ O(#res * #casas)
```

Complejidad: $O(p^2 * (maxCasas^2 + maxComercios^2))$

Popularidad(**in** s : `simcity`) → **out** res : `nat`

```
1: res ← s.popularidad                                     ▷ O(1)
```

Complejidad: $O(1)$

Turnos(**in** s : `simcity`) → **out** res : `nat`

```
1: res ← s.maxTurnos                                       ▷ O(1)
```

Complejidad: $O(1)$

CasasPropias(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Reconstruye el diccionario *s.casas* calculando el nivel de cada casa a partir de su turno de creación.

```
1: res ← Vacio()                                           ▷ O(1)
2: itCasas ← CrearIt(s.casas)                              ▷ O(1)
3: while HaySiguiente(itCasas) do                          ▷ O(#s.casas)
4:     pos ← SiguienteClave(itCasas)                       ▷ O(1)
5:     turnosCreacion ← SiguienteSignificado(itCasas)      ▷ O(1)
6:     nivel ← s.turnos - turnosCreacion                   ▷ O(1)
7:     DefinirRapido(res, pos, nivel)                      ▷ O(1)
8:     Avanzar(itCasas)                                    ▷ O(1)
9: end while
```

Complejidad: $O(\#s.casas)$

CasasUniones(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Construye un diccionario de todas las casas de las uniones. Se hace una recursión mútua con la función Casas ya que se piden las casas de cada unión, que vienen con su nivel definido a partir de los turnos del SimCity de la unión. Luego se ajusta el nivel compensando los turnos transcurridos desde la unión (recordemos que un SimCity ya no puede ser modificado luego de ser unido a otro, por lo tanto desde la perspectiva de ese SimCity unido, sus construcciones quedan fijadas en el nivel que tenían cuando sucedió la unión).

```
1:  res ← Vacio()                                                          ▷ O(1)
2:  itUniones ← CrearIt(s.uniones)                                         ▷ O(1)
3:  while HaySiguiente(itUniones) do                                       ▷ O(p)
4:      sUnion ← *Siguiente(itUniones).simcity                             ▷ O(1)
5:      turnosUnion ← Siguiente(itUniones).turnosUnion                     ▷ O(1)
6:      turnosDesdeLaUnion ← s.turnos - turnosUnion                        ▷ O(1)
7:      casasUnion ← Casas(sUnion)
8:      SumarATodos(casasUnion, turnosDesdeLaUnion)                        ▷ O(#casasUnion^2)
9:      UnirConstrucciones(res, casasUnion)                                ▷ O(max{1, #res} * #casasUnion)
10:     Avanzar(itUniones)                                                 ▷ O(1)
11: end while
```

Complejidad: $O(\sum_{i=0}^{p} i * maxCasas^2) = O(p^2 * maxCasas^2)$

De forma similar a la función Mapa, p es la popularidad de la instancia s lo cual representa la cantidad total de uniones (directas o indirectas), y $maxCasas$ representa la máxima cantidad de casas de alguna unión (directa o indirecta), sin considerar los casas que obtiene por sus propias uniones.

ComerciosPropios(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Algoritmo análogo a CasasPropias. Reconstruye el diccionario $s.comercios$ calculando el nivel de cada comercio a partir de su turno de creación.

```
1:  res ← Vacio()                                                          ▷ O(1)
2:  itComercios ← CrearIt(s.comercios)                                     ▷ O(1)
3:  while HaySiguiente(itComercios) do                                     ▷ O(#s.comercios)
4:      pos ← SiguienteClave(itComercios)                                  ▷ O(1)
5:      turnosCreacion ← SiguienteSignificado(itComercios)                 ▷ O(1)
6:      nivel ← s.turnos - turnosCreacion                                  ▷ O(1)
7:      DefinirRapido(res, pos, nivel)                                     ▷ O(1)
8:      Avanzar(itComercios)                                               ▷ O(1)
9:  end while
```

Complejidad: $O(\#s.comercios)$

ComerciosUniones(**in** s : `simcity`) → **out** res : `dicc(pos, nivel)`

Algoritmo análogo a CasasUniones. Construye un diccionario de todos los comercios de las uniones.

```
1:  res ← Vacio()                                                          ▷ O(1)
2:  itUniones ← CrearIt(s.uniones)                                         ▷ O(1)
3:  while HaySiguiente(itUniones) do                                       ▷ O(p)
4:      sUnion ← *Siguiente(itUniones).simcity                             ▷ O(1)
5:      turnosUnion ← Siguiente(itUniones).turnosUnion                     ▷ O(1)
6:      turnosDesdeLaUnion ← s.turnos - turnosUnion                        ▷ O(1)
7:      comerciosUnion ← Comercios(sUnion)
8:      SumarATodos(comerciosUnion, turnosDesdeLaUnion)                    ▷ O(#comerciosUnion^2)
9:      UnirConstrucciones(res, comerciosUnion)                            ▷ O(max{1, #res} * #comerciosUnion)
10:     Avanzar(itUniones)                                                 ▷ O(1)
11: end while
```

Complejidad: $O(\sum_{i=0}^{p} i * maxComercios^2) = O(p^2 * maxComercios^2)$

De forma similar a la función Mapa y la función CasasPropias, p es la popularidad de la instancia s lo cual representa la cantidad total de uniones (directas o indirectas), y $maxComercios$ representa la máxima cantidad de comercios de alguna unión (directa o indirecta), sin considerar los comercios que obtiene por sus propias uniones.

SumarATodos(**in/out** d: `dicc(pos, nivel)`, **in** n: `nat`)
Incrementa n a todos los significados de d.

```
1: it ← CrearIt(d)                                          ▷ O(1)
2: while HaySiguiente(it) do                                ▷ O(#d)
3:     pos ← SiguienteClave(it)                             ▷ O(1)
4:     nivel ← SiguienteSignificado(it)                     ▷ O(1)
5:     Definir(d, pos, nivel + n)                           ▷ O(#d)
6:     Avanzar(it)                                          ▷ O(1)
7: end while
```

Complejidad: $O(\#d^2)$

UnirConstrucciones(**in/out** *dst*: `dicc(pos, nivel)`, **in** *src*: `dicc(pos, nivel)`)
Une todas las construcciones definidas en *src* con las que están en *dst*, quedándose con el mayor nivel en el caso de un conflicto.

```
1: it ← CrearIt(src)                                        ▷ O(1)
2: while HaySiguiente(it) do                                ▷ O(#src)
3:     pos ← SiguienteClave(it)                             ▷ O(1)
4:     nivelSrc ← SiguienteSignificado(it)                  ▷ O(1)
5:     if Definido?(dst, pos) then                          ▷ O(#dst)
6:         nivelDst ← Significado(dst, pos)                 ▷ O(#dst)
7:         Definir(dst, pos, max(nivelDst, nivelSrc))       ▷ O(#dst)
8:     else
9:         DefinirRapido(dst, pos, nivelSrc)                ▷ O(1)
10:    end if
11:    Avanzar(it)                                          ▷ O(1)
12: end while
```

Complejidad: $O(\#src * max\{1, \#dst\})$

SacarCasasSolapadas(**in/out** *casas*: `dicc(pos, nivel)`, **in** *comercios*: `dicc(pos, nivel)`)
Resuelve los conflictos entre casas y comercios eliminando de *casas* las que tienen un nivel igual o menor al de un comercio.

```
1: itCasas ← CrearIt(casas)                                 ▷ O(1)
2: while HaySiguiente(itCasas) do                           ▷ O(#casas)
3:     pos ← SiguienteClave(itCasas)                        ▷ O(1)
4:     nivelCasa ← SiguienteSignificado(itCasas)            ▷ O(1)
5:     if Definido?(comercios, pos) then                    ▷ O(#comercios)
6:         nivelComercio ← Significado(comercios, pos)      ▷ O(#comercios)
7:         if nivelCasa ≤ nivelComercio then                ▷ O(1)
8:             EliminarSiguiente(itCasas)                   ▷ O(1)
9:         end if
10:    end if
11:    Avanzar(itCasas)                                     ▷ O(1)
12: end while
```

Complejidad: $O(\#casas * \#comercios)$

SacarComerciosSolapados(**in**/**out** *comercios*: `dicc(pos, nivel)`, **in** *casas*: `dicc(pos, nivel)`)
Algoritmo análogo a SacarCasasSolapadas. Resuelve los conflictos entre comercios y casas eliminando de *comercios* los que tienen un nivel estrictamente menor al de una casa.

```
1:  itComercios ← CrearIt(comercios)                                   ▷ O(1)
2:  while HaySiguiente(itComercios) do                                 ▷ O(#comercios)
3:      pos ← SiguienteClave(itComercios)                              ▷ O(1)
4:      nivelComercio ← SiguienteSignificado(itComercios)              ▷ O(1)
5:      if Definido?(casas, pos) then                                  ▷ O(#casas)
6:          nivelCasa ← Significado(casa, pos)                         ▷ O(#casas)
7:          if nivelComercio < nivelCasa then                          ▷ O(1)
8:              EliminarSiguiente(itComercios)                         ▷ O(1)
9:          end if
10:     end if
11:     Avanzar(itComercios)                                           ▷ O(1)
12: end while
```

Complejidad: $O(\#comercios * \#casas)$

AdoptarNiveles(**in**/**out** *comercios*: `dicc(pos, nivel)`, **in** *casas*: `dicc(pos, nivel)`)
Construye un nuevo diccionario de comercios con los niveles adoptados (solo si el nivel que podría adoptar es mayor que el nivel que ya tiene).

```
1: itComercios ← CrearIt(comercios)                                    ▷ O(1)
2: while HaySiguiente(itComercios) do                                  ▷ O(#comercios)
3:     pos ← SiguienteClave(itComercios)                               ▷ O(1)
4:     nivelComercio ← SiguienteSignificado(itComercios)               ▷ O(1)
5:     nivelManhattan ← ObtenerNivelManhattan(pos, casas)              ▷ O(#casas)
6:     Definir(comercios, pos, max(nivelComercio, nivelManhattan))     ▷ O(#comercios)
7:     Avanzar(itComercios)                                            ▷ O(1)
8: end while
```

Complejidad: $O(\#comercios * (\#casas + \#comercios))$

ObtenerNivelManhattan(**in** *posComercio*: `pos`, **in** *casas*: `dicc(pos, nivel)`) → **out** *res*: `nat`
Devuelve el nivel máximo que puede adoptar un comercio de acuerdo a los niveles de las casas que están a cierta distancia Manhattan. Si no hay casas a la distancia Manhattan estipulada, se devuelve 0 por convención.

```
1:  res ← 0                                                            ▷ O(1)
2:  itCasas ← CrearIt(casas)                                           ▷ O(1)
3:  while HaySiguiente(itCasas) do                                     ▷ O(#casas)
4:      posCasa ← SiguienteClave(itCasas)                              ▷ O(1)
5:      nivelCasa ← SiguienteSignificado(itCasas)                      ▷ O(1)
6:      distanciaManhattan ← DistanciaManhattan(posComercio, posCasa)  ▷ O(1)
7:      if distanciaManhattan = 3 then                                 ▷ O(1)
8:          res ← max(res, nivelCasa)                                  ▷ O(1)
9:      end if
10:     Avanzar(itCasas)                                               ▷ O(1)
11: end while
```

Complejidad: $O(\#casas)$

DistanciaManhattan(**in** *p1*: `pos`, **in** *p2*: `pos`) → **out** *res*: `nat`

```
1: res ← abs(x(p1) - x(p2)) + abs(y(p1) - y(p2))                       ▷ O(1)
```

Complejidad: $O(1)$

UnirConjuntos(**in/out** $c1$: `conj(nat)`, **in** $c2$: `conj(nat)`)

```
1: it ← CrearIt(c2)                          ▷ O(1)
2: while HaySiguiente(it) do                 ▷ O(#c2)
3:     Agregar(c1, Siguiente(it))            ▷ O(#c1)
4:     Avanzar(it)                           ▷ O(1)
5: end while
```

Complejidad: $O(\#c1 * \#c2)$
Por cada elemento de c2 que se agrega a c1, se debe verificar que estos no pertenezcan ya a c1.

2.3. Módulo Servidor

Interfaz

se explica con: SERVIDOR

géneros: `servidor`

Operaciones básicas

NUEVOSERVIDOR() $\rightarrow$ *res* : `servidor`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nuevoServidor()\}$
Complejidad: $O(1)$
Descripción: Crea un nuevo servidor sin ninguna ciudad.

NUEVACIUDAD(**in/out** srv : `servidor`, **in** n : `Nombre`, **in** m : `mapa`)
Pre $\equiv$ $\{srv_0 = srv \land n \notin claves(ciudades(srv))\}$
Post $\equiv$ $\{srv =_{\text{obs}} nuevaCiudad(srv_0, n, m)\}$
Complejidad: $O(|n|) + O(SimCity.NuevoSimCity)$
Descripción: Crea una nueva ciudad identificada por nombre en el servidor.

AVANZARTURNO(**in/out** srv : `servidor`, **in** n : `Nombre`, **in** cs : `dicc(pos, construccion)`)
Pre $\equiv$ $\{srv_0 = srv$
$\land\ n \in claves(ciudades(srv))$
$\land\ \neg(\exists m : nombre)(n \in uniones(srv, m))$
$\land_{\text{L}}\ construccionesValidas(obtener(n, ciudades(srv)), cs)\}$
Post $\equiv$ $\{srv =_{\text{obs}} avanzarTurno(srv_0, n, cs)\}$
Complejidad: $O(|n|) + O(SimCity.AvanzarTurno)$
Descripción: Avanza el turno de la ciudad identificada por nombre construyendo las casas y comercios en sus posiciones indicadas en cs. Recordemos que $O(AvanzarTurno)$ del módulo SimCity tiene complejidad $O(1)$ cuando se construye una única casa o comercio ya que $\#claves(cs) = 1$. Por lo tanto, en ese caso, la complejidad de esta operación en el servidor resulta $O(|n|)$.

UNIRCIUDADES(**in/out** srv : `servidor`, **in** $n1$: `Nombre`, **in** $n2$: `Nombre`)
Pre $\equiv$ $\{srv_0 = srv$
$\land\ n1 \neq n2$
$\land\ \{n1, n2\} \subseteq claves(ciudades(srv))$
$\land\ \neg(\exists m : nombre)(n1 \in uniones(srv, m))$
$\land_{\text{L}}\ unionValida(obtener(n1, ciudades(srv)), obtener(n2, ciudades(srv)))\}$
Post $\equiv$ $\{srv =_{\text{obs}} unirCiudades(srv_0, n1, n2)\}$
Complejidad: $O(max\{|n1|, |n2|\}) + O(SimCity.Unir)$
Descripción: Une la ciudad identificada por n2 a la ciudad identificada por n1.
Aliasing: La instancia identificada por n1 pasa a tener una referencia no modificable a la instancia identificada por n2.

CIUDADES(**in** srv : `servidor`) $\rightarrow$ *res* : `conj(Nombre)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} claves(ciudades(srv))\}$
Complejidad: $O(\#ciudades(srv))$
Descripción: Devuelve todos los nombres de las ciudades que están en el servidor.

MAPA(**in** *srv*: `servidor`, **in** n: `Nombre`) → *res* : `mapa`
Pre ≡ $\{n \in claves(ciudades(srv))\}$
Post ≡ $\{res =_{\text{obs}} mapa(obtener(n, ciudades(srv)))\}$
Complejidad: $O(|n|) + O(SimCity.Mapa)$
Descripción: Devuelve el mapa de la ciudad identificada por nombre.

CASAS(**in** *srv*: `servidor`, **in** n: `Nombre`) → *res* : `dicc(pos, nivel)`
Pre ≡ $\{n \in claves(ciudades(srv))\}$
Post ≡ $\{res =_{\text{obs}} casas(obtener(n, ciudades(srv)))\}$
Complejidad: $O(|n|) + O(SimCity.Casas)$
Descripción: Devuelve las casas de la ciudad identificada por nombre.

COMERCIOS(**in** *srv*: `servidor`, **in** n: `Nombre`) → *res* : `dicc(pos, nivel)`
Pre ≡ $\{n \in claves(ciudades(srv))\}$
Post ≡ $\{res =_{\text{obs}} comercios(obtener(n, ciudades(srv)))\}$
Complejidad: $O(|n|) + O(SimCity.Comercios)$
Descripción: Devuelve los comercios de la ciudad identificada por nombre.

POPULARIDAD(**in** *srv*: `servidor`, **in** n: `Nombre`) → *res* : `Nat`
Pre ≡ $\{n \in claves(ciudades(srv))\}$
Post ≡ $\{res =_{\text{obs}} popularidad(obtener(n, ciudades(srv)))\}$
Complejidad: $O(|n|) + O(SimCity.Popularidad)$
Descripción: Devuelve la popularidad de la ciudad identificada por nombre.

TURNOS(**in** *srv*: `servidor`, **in** n: `Nombre`) → *res* : `Nat`
Pre ≡ $\{n \in claves(ciudades(srv))\}$
Post ≡ $\{res =_{\text{obs}} turnos(obtener(n, ciudades(srv)))\}$
Complejidad: $O(|n|) + O(SimCity.Turnos)$
Descripción: Devuelve el turno (antigüedad) de la ciudad identificada por nombre.

Representación

El servidor funciona como un 'proxy' a los SimCitys que contiene. Todas las operaciones de un SimCity son ejecutadas únicamente desde el servidor.

Las ciudades que maneja el servidor son guardadas en un diccionario que mapea el nombre de la ciudad con su respectiva instancia SimCity. Este diccionario se implementa con la estructura de datos Trie para poder garantizar que todas las operaciones del diccionario (definir, obtener, pertenece, borrar, etc) tengan complejidad, en peor caso, $O(|nombre|)$, donde nombre es el nombre del SimCity más largo que contiene el servidor. A su vez, dado un nombre n con el cual estamos realizando alguna de las operaciones del diccionario antes mencionadas, la complejidad exacta sería $\Theta(|n|)$.

`servidor` **se representa con** `estr`

donde `estr` es `tupla(`*ciudades*: `diccTrie(nombre, simcity))`

Rep : estr $\longrightarrow$ bool
Rep(e) ≡ true $\iff$ true

Abs : estr e $\longrightarrow$ servidor {Rep(e)}
Abs(e) $=_{\text{obs}}$ s: servidor | e.ciudades $=_{\text{obs}}$ ciudades(srv)

Algoritmos

NuevoServidor() → **out** *res*: `servidor`
1: res.ciudades ← Vacio() ▷ $O(1)$
Complejidad: $O(1)$

NuevaCiudad(**in**/**out** *srv* : `servidor`, **in** *n* : `nombre`, **in** *m* : `mapa`)

1: Definir(srv.ciudades, n, NuevoSimCity(m)) ▷ $O(|n|) + O(NuevoSimCity)$

Complejidad: $O(|n|) + O(NuevoSimCity)$

AvanzarTurno(**in**/**out** *srv* : `servidor`, **in** *n* : `nombre`, **in** *cs* : `dicc(pos, construccion)`)

1: s ← Significado(srv.ciudades, n) ▷ $O(|n|)$
2: AvanzarTurno(s, cs) ▷ $O(AvanzarTurno)$

Complejidad: $O(|n|) + O(AvanzarTurno)$

UnirCiudades(**in**/**out** *srv* : `servidor`, **in** *n*1 : `nombre`, **in** *n*2 : `nombre`)

1: s1 ← Significado(srv.ciudades, n1) ▷ $O(|n1|)$
2: s2 ← Significado(srv.ciudades, n2) ▷ $O(|n2|)$
3: Unir(s1, s2) ▷ $O(Unir)$

Complejidad: $O(|n1| + |n2|) + O(Unir) = O(max\{|n1|, |n2|\}) + O(Unir)$

Ciudades(**in** *srv* : `servidor`) → **out** *res* : `conj(nombre)`

1: res ← Vacio() ▷ $O(1)$
2: itCiudades ← CrearIt(srv.ciudades) ▷ $O(1)$
3: **while** HaySiguiente(itCiudades) **do** ▷ O(#srv.ciudades)
4: nombre ← SiguienteClave(itCiudades) ▷ $O(1)$
5: AgregarRapido(res, nombre) ▷ $O(1)$
6: **end while**

Complejidad: $O(\#srv.ciudades)$

Mapa(**in** *srv* : `servidor`, **in** *n* : `nombre`) → **out** *res* : `mapa`

1: s ← Significado(srv.ciudades, n) ▷ $O(|n|)$
2: res ← Mapa(s) ▷ $O(Mapa)$

Complejidad: $O(|n|) + O(mapa)$

Casas(**in** *srv* : `servidor`, **in** *n* : `nombre`) → **out** *res* : `dicc(pos, nivel)`

1: s ← Significado(srv.ciudades, n) ▷ $O(|n|)$
2: res ← Casas(s) ▷ $O(Casas)$

Complejidad: $O(|n|) + O(Casas)$

Comercios(**in** *srv* : `servidor`, **in** *n* : `nombre`) → **out** *res* : `dicc(pos, nivel)`

1: s ← Significado(srv.ciudades, n) ▷ $O(|n|)$
2: res ← Comercios(s) ▷ $O(Comercios)$

Complejidad: $O(|n|) + O(Comercios)$

Popularidad(**in** *srv* : `servidor`, **in** *n* : `nombre`) → **out** *res* : `nat`

1: s ← Significado(srv.ciudades, n) ▷ $O(|n|)$
2: res ← Popularidad(s) ▷ $O(Popularidad)$

Complejidad: $O(|n|) + O(Popularidad)$

Turnos(**in** *srv* : `servidor`, **in** *n* : `nombre`) → **out** *res* : `nat`

```
1: s ← Significado(srv.ciudades, n)        ▷ O(|n|)
2: res ← Turnos(s)                         ▷ O(Turnos)
```

Complejidad: $O(|n|) + O(Turnos)$

3. Notas

- Unas horas antes de la entrega del TP se aclaró lo siguiente: 'tanto agregar una casa o n, en una llamada a funcion, ambas tienen que ser O(1) en el SC'. Nuestro entendimiento del enunciado y lo que resolvimos en esta entrega es que agregar una casa tiene complejidad O(1), y agregar n casas tiene complejidad O(n) (hablando de las funciones propias del SimCity). A nuestro criterio, el enunciado admitía perfectamente este entendimiento. No tenemos tiempo para modificar nuestra entrega, pero describimos a continuación los cambios que haríamos si tuviésemos que garantizar esa complejidad. La estructura del SimCity ya no tendría 2 diccionarios independientes para guardar las posiciones de las casas y comercios. En cambio, tendría una única variable $construciones$: $lista(dicc(pos, construccion))$ en donde se agrega en O(1) todas las construcciones del turno cuando se ejecuta AvanzarTurno. De esta forma mantenemos la misma interfaz del módulo, y en las funciones Casas y Comercios, antes de realizar los algoritmos planteados, obtendríamos de la lista de construciones todas las casas o comercios construidos en todos los turnos que pasaron.
- En el cálculo de complejidades abusamos de notación y usamos el operador cardinal (#) tanto para conjuntos como diccionarios. En el caso de un diccionario d cualquiera, se debe interpretar la notación de esta forma: $\#d \equiv \#claves(d)$.
- La lógica que tomamos para resolver los conflictos es quedarnos con la construcción de mayor nivel, y ante un empate entre casa y comercio, nos quedamos con el comercio.