# Algoritmos

## Algoritmos de Fichin

---

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

1: $res \leftarrow \langle m, False, NULL, NULL, vacio \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iNuevaPartida**(**in/out** $f$ : `fichin`, **in** $a$ : `jugador`) $\rightarrow$ $res$ : $bool$

$res \leftarrow False$
**if** $\neg f.alguienJugando$ **then**
  $f.jugadorActual \leftarrow jugador$ ▷ $\Theta(1)$
  $f.alguienJugando \leftarrow True$ ▷ $\Theta(1)$
  $f.partidaActual \leftarrow partida.iNuevaPartida(f.mapa)$ ▷ $\mathcal{O}(c)$
  $res \leftarrow True$
**end if**

Complejidad: $\mathcal{O}(c)$
Justificación: La complejidad de este algoritmo es igual a la complejidad de $iNuevaPartida$, que es $\mathcal{O}(c)$, con $c = \#chocolates$.

---

**iMover**(**in/out** $f$ : `fichin`, **in** $d$ : `direccion`) $\rightarrow$ $res$ : $bool$

1: **if** $f.alguienJugando$ **then**
2:  partida.iMover(f.partidaActual, d) ▷ $\Theta(1)$
3:  $res \leftarrow True$
4:  **if** partida.iGanó?(f.partidaActual) **then** ▷ $\mathcal{O}(1)$
5:  **if** def?(f.jugadorActual, f.ranking) **then** ▷ $\mathcal{O}(|J|)$
6:  **if** obtener(f.jugadorActual, f.ranking) > partida.iCantMov(f.partidaActual) **then** ▷ $\mathcal{O}(|J|)$
7:  definir(f.jugadorActual, partida.iCantMov(f.partidaActual), f.ranking) ▷ $\mathcal{O}(|J|)$
8:  **end if**
9:  **else** definir(f.jugadorActual, partida.iCantMov(f.partidaActual), f.ranking) ▷ $\mathcal{O}(|J|)$
10:  **end if**
11:  **end if**
12: **else**
13:  $res \leftarrow False$ ▷ $\Theta(1)$
14: **end if**

Complejidad: $\mathcal{O}(|J|)$ si se realiza un movimiento que finaliza la partida
Justificación: El diccionario se implementa con un trie, que tiene $\mathcal{O}(|J|)$ como complejidad de def?, acceder a significado y definir.

---

**iVerRanking**(**in** $f$ : `fichin`) $\rightarrow$ $res$ : $dicc$

1: $res \leftarrow f.ranking$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iObjetivo**(**in** $f$ : `fichin`, **in** $j$ : `jugador`) $\rightarrow$ $res$ : $\langle jugador, nat \rangle$

```
1:  i ← 0                                                                  ▷ Θ(1)
2:  C ← obtenerClavesTrie(f.ranking)                                       ▷ Θ(J)
3:  S ← ∅                                                                  ▷ Θ(1)
4:  while i < long(C) − 1 do
5:      if obtener(C[i], f.ranking) < obtener(jugador, f.ranking) then     ▷ O(|J|)
6:          Ag(C[i], S)                                                    ▷ Θ(1)
7:      end if
8:      i ← i + 1                                                          ▷ Θ(1)
9:  end while
10: if ∅?(S) then
11:     res ← ⟨jugador, obtener(f.ranking, jugador)⟩                       ▷ Θ(1)
12: else
13:     max ← obtener(f.ranking, S[0])                                     ▷ O(|J|)
14:     maxJugador ← S[0]                                                  ▷ Θ(1)
15:     j ← 0
16:     while j < long(S) do
17:         if obtener(f.ranking, S[j]) ≥ max then                         ▷ Θ(|J|)
18:             max ← obtener(f.ranking, S[j])                             ▷ Θ(|J|)
19:             maxJugador ← S[j]                                          ▷ Θ(1)
20:         end if
21:         j ← j + 1
22:     end while
23:     res ← ⟨maxJugador, max⟩
24: end if
```

Complejidad: $\sum_{k \in J} \mathcal{O}(|J|) + \mathcal{O}(|J|) + \sum_{k \in S} \mathcal{O}(|J|) = \mathcal{O}(|J| * J)$
Justificación: Los conjuntos $C, S$ son implementados con una lista enlazada para que insertar elementos sea $\mathcal{O}(1)$.

## Módulo Diccionario Trie($\kappa$, $\sigma$)

El módulo Diccionario Trie provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo $\mathcal{O}(|J|)$, donde $J$ es la cantidad de elementos del diccionario y $|J| = \text{máx}_{n \in J} |n|$.

Para describir la complejidad de las operaciones, vamos a llamar $copy(k)$ al costo de copiar el elemento $k \in \kappa \cup \sigma$ y $equal(k_1, k_2)$ al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales (i.e., *copy* y *equal* son funciones de $\kappa \cup \sigma$ y $\kappa \times \kappa$ en $\mathbb{N}$, respectivamente).[1]

# Interfaz

**parámetros formales**

**géneros** $\kappa, \sigma$

**función** $\bullet = \bullet$(**in** $k_1$ : $\kappa$, **in** $k_2$ : $\kappa$) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
**Complejidad:** $\Theta(equal(k_1, k_2))$
**Descripción:** función de igualdad de $\kappa$'s

**función** COPIAR(**in** $k$ : $\kappa$) $\rightarrow$ $res$ : $\kappa$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} k$}
**Complejidad:** $\Theta(copy(k))$
**Descripción:** función de copia de $\kappa$'s

**función** COPIAR(**in** $s$ : $\sigma$) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}} s$}
**Complejidad:** $\Theta(copy(s))$
**Descripción:** función de copia de $\sigma$'s

**se explica con**: DICCIONARIO($\kappa, \sigma$), ITERADOR BIDIRECCIONAL(TUPLA($\kappa, \sigma$)).

**géneros**: `dicc`($\kappa, \sigma$), `itDicc`($\kappa, \sigma$).

## Operaciones básicas de diccionario

[1] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* y *equal* en función del tamaño de $k$. A la hora de usarlo, habrá que realizar la traducción.

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

DEFINIR(**in/out** $d$: `dicc`$(\kappa, \sigma)$, **in** $k$: $\kappa$, **in** $s$: $\sigma$)
**Pre** $\equiv \{d =_{\text{obs}} d_0\}$
**Post** $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s)\}$
**Complejidad:** $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
**Descripción:** define la clave $k$ con el significado $s$ en el diccionario.

DEFINIDO?(**in** $d$: `dicc`$(\kappa, \sigma)$, **in** $k$: $\kappa$) $\rightarrow res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{def?}(d,\, k)\}$
**Complejidad:** $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
**Descripción:** devuelve `true` si y sólo $k$ está definido en el diccionario.

SIGNIFICADO(**in** $d$: `dicc`$(\kappa, \sigma)$, **in** $k$: $\kappa$) $\rightarrow res$ : $\sigma$
**Pre** $\equiv \{\text{def?}(d,\, k)\}$
**Post** $\equiv \{\text{alias}(res =_{\text{obs}} \text{Significado}(d,\, k))\}$
**Complejidad:** $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
**Descripción:** devuelve el significado de una clave en el diccionario.

BORRAR(**in/out** $d$: `dicc`$(\kappa, \sigma)$, **in** $k$: $\kappa$)
**Pre** $\equiv \{d = d_0 \wedge \text{def?}(d,\, k)\}$
**Post** $\equiv \{d =_{\text{obs}} \text{borrar}(d_0, k)\}$
**Complejidad:** $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
**Descripción:** elimina la clave $k$ y su significado de $d$.

OBTENERCLAVESDELTRIE(**in** $d$: `dicc`$(\kappa, \sigma)$) $\rightarrow res$ : `conj`$(\kappa)$
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{claves}(d)\}$
**Complejidad:** $\Theta(J)$
**Descripción:** devuelve el conjunto de claves del diccionario.

#CLAVES(**in** $d$: `dicc`$(\kappa, \sigma)$) $\rightarrow res$ : `nat`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \#\text{claves}(d)\}$
**Complejidad:** $\Theta(1)$
**Descripción:** devuelve la cantidad de claves del diccionario.

COPIAR(**in** $d$: `dicc`$(\kappa, \sigma)$) $\rightarrow res$ : `dicc`$(\kappa, \sigma)$
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} d\}$
**Complejidad:** $\Theta\left(\sum_{k \in K} (copy(k) + copy(\text{significado}(k, d)))\right)$, donde $K = \text{claves}(d)$
**Descripción:** genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** $d_1$ : `dicc`$(\kappa, \sigma)$, **in** $d_2$ : `dicc`$(\kappa, \sigma)$) $\rightarrow res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} equal(\langle k_1, s_1\rangle, \langle k_2, s_2\rangle)\right)$, donde $K_i = \text{claves}(d_i)$ y $s_i = \text{significado}(d_i, k_i)$, $i \in \{1, 2\}$.
**Descripción:** compara $d_1$ y $d_2$ por igualdad, cuando $\sigma$ posee operación de igualdad.

**Requiere:** $\bullet = \bullet(\textbf{in } s_1 : \sigma, \textbf{in } s_2 : \sigma) \to res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} (s_1 = s_2)\}$
**Complejidad:** $\Theta(equal(s_1, s_2))$
**Descripción:** función de igualdad de $\sigma$'s