UNIVERSIDAD DEL VALLE

Programación Funcional y Concurrente - 750013C-01

Proyecto de curso

Nicolás Garcés Larrahondo **2180066-3743**

Santiago González Gálvez **2183392-3743**

Wilson Andrés Mosquera Zapata **2182116-3743**

Informe sobre las funciones

Para este trabajo, se definieron los siguientes tipos de datos:

```
type Tablon = (Int,Int,Int)
type Finca = Vector[Tablon]
type Distancia = Vector[Vector[Int]]
type ProgRiego = Vector[Int]
type TiempoInicioRiego = Vector[Int]
```

Donde un tablón representa una tupla de 3 enteros, donde el primer elemento corresponde con el tiempo de supervivencia del tablón, el segundo corresponde con el tiempo de regado y el tercero a la prioridad del tablón.

Una finca consta de varios tablones.

Entre cada tablón de una finca existe una distancia numérica, que se representa a través de una matriz de distancias "$Distancia$".

Una programación de riego "$ProgRiego$" es un vector de enteros, que representa el orden en el que se regarán los tablones.

De igual forma, el tiempo de inicio de riego es un vector de enteros, que representa el tiempo en el que se iniciará el riego de cada tablón, de acuerdo con una programación específica.

Además de ello, se definieron las siguientes funciones auxiliares:

```scala
def fincaAlAzar(long: Int): Finca = {
  val v = Vector.fill(long){
    (
      random.nextInt(long * 2) + 1,
      random.nextInt(long) + 1,
      random.nextInt(4) + 1
    )}
  v
}
```

```scala
def distanciaAlAzar(long: Int): Distancia = {
  val v = Vector.fill(long, long){
    random.nextInt(long * 3) + 1
  }
  Vector.tabulate(long, long){
    (i,j) => if (i < j) v(i)(j)
    else if (i==j) 0
    else v(j)(i)
  }
}
```

Donde $fincaAlAzar$ y $distanciaAlAzar$ son útiles para generar realizar pruebas, generando tanto fincas como matrices de distancias de forma aleatoria.

A su vez, por cuenta propia, se definió la siguiente función para generar una programación al azar:

```scala
def programacionAlAzar(n: Int): Vector[Int] = {
  val numeros = (0 ≤ until < n).toVector
  val numerosAleatorios = Random.shuffle(numeros)

  Vector.tabulate(n)(i => numerosAleatorios(i))
}
```

Finalmente, para acceder a los atributos de un tablón de una finca, se definió $tsup$, $treg$ y $prio$, que representan el tiempo de supervivencia, el tiempo de riego y la prioridad, respectivamente.

```scala
def tsup(f: Finca, i: Int): Int = {
  f(i)._1
}

Santiago González
def treg(f: Finca, i: Int): Int = {
  f(i)._2
}

Santiago González
def prio(f: Finca, i: Int): Int = {
  f(i)._3
}
```

1. Función tIR

1.1. Estructuras de datos empleadas

La implementación de la función tIR fue la siguiente:

```scala
def tIR(f: Finca, pi: ProgRiego): TiempoInicioRiego = {
  def acumulado(v: Vector[Int]): Vector[Int] = {
    v.scanLeft(0)(_ + _).zip(pi).sortBy(_._2).map(x => x._1)
  }

  val tiempoInicioRiego = Vector.tabulate(pi.length - 1)(i => treg(f, pi(i)))
  acumulado(tiempoInicioRiego)
}
```

La función tIR calcula los tiempos de inicio de riego de cada tablón en la programación pi para una finca específica f. Utiliza una estructura de datos de tipo Vector para almacenar los resultados.

La función interna $acumulado$ realiza una serie de operaciones en un vector de enteros. Utiliza $scanLeft$ para calcular la suma acumulada de los elementos en el vector, generando un nuevo vector con las sumas parciales. Luego combina este vector con la programación pi utilizando zip, creando pares de elementos donde el primero es la suma acumulada y el segundo es el elemento correspondiente de pi. Posteriormente, ordena estos pares según los elementos de pi utilizando $sortBy$. Finalmente, proyecta solo las sumas acumuladas utilizando map y devuelve un nuevo vector con estas sumas en el mismo orden que pi organizado ascendentemente.

La variable $tiempoInicioRiego$ se crea utilizando $Vector.tabulate$, generando un vector de longitud $pi.length - 1$. Cada elemento en la posición i de este vector se obtiene llamando a la función $treg$ con los argumentos f y $pi(i)$. Esto significa que cada elemento en $tiempoInicioRiego$ representa el tiempo de riego del tablón i en la programación pi.

Por último, la función tIR devuelve el resultado de llamar a la función acumulado pasando $tiempoInicioRiego$ como argumento. Esto proporciona un vector que contiene los tiempos de inicio de riego de cada tablón en orden ascendente, según la programación pi. La estructura de datos $Vector$ es utilizada en todo el proceso para almacenar y manipular los resultados de manera eficiente.

1.2. Argumentación sobre la corrección

Sea P_f la función tIR implementada en Scala.
Sea $\mathcal{F}$ una finca con n tablones, es decir:

$$\mathcal{F} = Vector(\mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1}), n \geq 1$$

Donde cada $\mathcal{T}_i$ es una tupla $< ts_i^{\mathcal{F}}, tr_i^{\mathcal{F}}, p_i^{\mathcal{F}} >$, y cada atributo representa el tiempo de supervivencia, el tiempo de regado y la prioridad del tablón i en la finca $\mathcal{F}$, respectivamente.

Y sea $\mathcal{P}$ una programación de los n tablones, es decir:

$$\mathcal{P} = Vector(0,1,\dots,n-1), n \geq 1$$

Donde el orden de los n tablones en la programación $\mathcal{P}$ puede ser distinto, pues una programación de riego es una permutación $\mathcal{P} =< \mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1} >$ de $< 0,1,\dots,n-1 >$ que indica el orden en que se regaran los tablones.

Sea $t_i^{\mathcal{P}}$ la función que describe la manera en que se calcula el tiempo de inicio de riego del tablón i para la programación $\mathcal{P}$.

Donde $t_i^{\mathcal{P}} = t_{i-1}^{\mathcal{P}} + tr_{i-1}^{\mathcal{F}}$ y $t_0^{\mathcal{P}} = 0$

Se desea demostrar que P_f retorna un vector que contiene los tiempos de inicio de riego de cada tablón de la finca $\mathcal{F}$ con la programación $\mathcal{P}$, donde la posición i del vector representa en qué tiempo se riega el tablón i.

Formalmente:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego: P_f(\mathcal{F}, \mathcal{P}) \rightarrow Vector\left(t_0^{\mathcal{P}}, t_1^{\mathcal{P}}, t_2^{\mathcal{P}}, \dots, t_{n-1}^{\mathcal{P}}\right)$$

Para ello, se usará el modelo de substitución.

Primero hallemos a qué corresponde la variable $tiempoInicioRiego$ substituyendo.

$$tiempoInicioRiego = Vector.tabulate(\mathcal{P}.\text{length} - 1)\left(i \Rightarrow treg\left(\mathcal{F}, \mathcal{P}(\text{i})\right)\right)$$

Suponiendo que $Vector.tabulate$ funciona correctamente, se generará un vector de tamaño $\mathcal{P}.\text{length} - 1$, donde el elemento en la posición i corresponde al tiempo de riego del tablón en la posición i de $\mathcal{P}$.

Note que como el tamaño de $tiempoInicioRiego$ es de $n-1$, sólo se realizará el cálculo de los tiempos de riego hasta el tablón en la posición $n-1$ de $\mathcal{P}$, pues nos interesa conocer el tiempo de inicio de riego de cada tablón, y el tiempo de riego del último tablón sólo sería significativo para calcular el tiempo final de la programación $\mathcal{P}$

Dicho lo anterior,

$$tiempoInicioRiego = Vector\left(treg\left(\mathcal{F}, \mathcal{P}(0)\right), treg\left(\mathcal{F}, \mathcal{P}(1)\right), \dots, treg\left(\mathcal{F}, \mathcal{P}(\text{n} - 2)\right)\right)$$

Suponiendo el correcto funcionamiento de $treg$, el resultado final de $tiempoInicioRiego$ es un vector con los tiempos de riego de cada tablón en el mismo orden de $\mathcal{P}$, es decir:

$$tiempoInicioRiego = Vector\left(tr_0^{\mathcal{F}}, tr_1^{\mathcal{F}}, \dots, tr_{n-2}^{\mathcal{F}}\right)$$

Luego, se le pasa como parámetro este vector a la función auxiliar $acumulado$, veamos qué retorna.

La función $acumulado$ recibe un vector de enteros v y retorna un vector del mismo tipo, que en este caso corresponde con el resultado que buscamos, un vector con los tiempos de riego iniciales.

Veamos entonces qué retorna P_f usando el modelo de substitución:

$$\begin{aligned} P_f(\mathcal{F},\mathcal{P}) &\to acumulado(tiempoInicioRiego) \\ &\to tiempoInicioRiego.scanLeft(0)(\_ + \_) \\ &= Vector\left(tr_0^{\mathcal{F}}, tr_1^{\mathcal{F}}, \dots, tr_{n-2}^{\mathcal{F}}\right).scanLeft(0)(\_ + \_) \end{aligned}$$

Suponiendo que $scanLeft$ funciona correctamente, se retornará un vector con los resultados parciales de sumar los elementos de la colección y el elemento inicial, así:

$$P_f(\mathcal{F},\mathcal{P}) \to Vector\left(\overbrace{0}^{t_0^{\mathcal{P}}=0}, \underbrace{tr_0^{\mathcal{F}}}_{t_1^{\mathcal{P}}=t_0^{\mathcal{P}}+tr_0^{\mathcal{F}}}, \overbrace{tr_0^{\mathcal{F}} + tr_1^{\mathcal{F}}}^{t_2^{\mathcal{P}}=t_1^{\mathcal{P}}+tr_1^{\mathcal{F}}}, \dots, \underbrace{t_{n-2}^{\mathcal{P}} + tr_{n-2}^{\mathcal{F}}}_{t_{n-1}^{\mathcal{P}}=t_{n-2}^{\mathcal{P}}+tr_{n-2}^{\mathcal{F}}}\right)$$

Note que el vector resultante coincide con la función $t_i^{\mathcal{P}}$, sin embargo, se requiere que los tiempos de inicio de riego sea un vector donde la posición i represente en qué tiempo se riega el tablón i.

Para este caso, como se tomó una programación que ya está en orden ascendente $(0,1,\dots,n-1)$, el vector está organizado, sin embargo, recordemos que el orden de los n tablones en la programación $\mathcal{P}$ puede ser distinto. En ese caso, sigamos substituyendo para ver el resultado de P_f.

$$\begin{aligned} P_f(\mathcal{F},\mathcal{P}) &\to Vector\left(0, t_0^{\mathcal{P}}, t_1^{\mathcal{P}}, t_2^{\mathcal{P}}, \dots, t_{n-1}^{\mathcal{P}}\right).zip(\mathcal{P}) = \\ &= Vector\left(\left(t_0^{\mathcal{P}}, 0\right), \left(t_1^{\mathcal{P}}, 1\right), \left(t_2^{\mathcal{P}}, 2\right), \dots, \left(t_{n-1}^{\mathcal{P}}, n-1\right)\right) \end{aligned}$$

Luego, asumiendo que la función $sortBy$ funciona correctamente se organizarán las tuplas del vector de acuerdo con el segundo elemento de la tupla, en orden ascendente. En este caso, ya están organizados.

Finalmente, asumiendo el correcto funcionamiento de map el resultado de P_f será el resultado de proyectar el primer elemento del vector anterior, es decir:

$$P_f(\mathcal{F},\mathcal{P}) \rightarrow Vector\left(\left(t_0^{\mathcal{P}},0\right),\left(t_1^{\mathcal{P}},1\right),\left(t_2^{\mathcal{P}},2\right),\ldots,\left(t_{n-1}^{\mathcal{P}},n-1\right)\right).map(x \Rightarrow x.\_1) \\ == Vector\left(t_0^{\mathcal{P}},t_1^{\mathcal{P}},t_2^{\mathcal{P}},\ldots,t_{n-1}^{\mathcal{P}}\right)$$

Lo cual corresponde con el vector que contiene los tiempos de inicio de riego de cada tablón de la finca $\mathcal{F}$ con la programación $\mathcal{P}$, en el orden adecuado.

Queda demostrado entonces, utilizando el modelo de substitución, que:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego: P_f(\mathcal{F},\mathcal{P}) \rightarrow Vector\left(t_0^{\mathcal{P}},t_1^{\mathcal{P}},t_2^{\mathcal{P}},\ldots,t_{n-1}^{\mathcal{P}}\right)$$

1.3. Casos de prueba

Los casos de prueba que se establecieron fueron

Para el **primer** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 1, 4, 3, 2 >

La secuencia de salida esperada es (0, 2, 10, 9, 5), donde cada posición 'i' representa el tiempo de inicio de riego del tablón 'i'. A continuación, se sigue la siguiente lógica en la programación:

Comenzamos con el tablón (0), que inicia en el tiempo 0. Luego, se programa el tablón (1), cuyo tiempo de inicio es calculado sumando el tiempo de inicio del tablón anterior (0) con su tiempo de regado. Por lo tanto, el tablón (1) tiene un tiempo de inicio de riego de 0 + 2 = 2.

Continuando con la programación, el siguiente elemento es el tablón (4). Siguiendo la misma lógica, su tiempo de inicio de riego es calculado sumando el tiempo de inicio del tablón anterior (1) con su tiempo de regado. En este caso, el tablón (4) tiene un tiempo de inicio de riego de 2 + 3 = 5.

Luego, se programa el tablón (3), cuyo tiempo de inicio de riego es calculado sumando el tiempo de inicio del tablón anterior (4) con su tiempo de regado. Por lo tanto, el tablón (3) tiene un tiempo de inicio de riego de 5 + 4 = 9.

Finalmente, se programa el tablón (2), cuyo tiempo de inicio de riego es calculado sumando el tiempo de inicio del tablón anterior (3) con su tiempo de regado. En este caso, el tablón (2) tiene un tiempo de inicio de riego de 9 + 1 = 10.

Organizando los tiempos obtenidos de acuerdo con el orden de los tablones, de forma ascendente, el resultado esperado sería (0, 2, 10, 9, 5).

Para el **segundo** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 4, 1, 2, 3 >

Siguiendo el mismo análisis que se empleó para la prueba 1, se espera obtener como resultado: (0, 6, 9, 11, 2)

Para el **tercer** caso se tienen de premisas:

finca = << 1, 5, 5 >, < 4, 2, 5 >, < 1, 4, 2 >< 2, 4, 2 >, < 3, 1, 3 >>

programación = < 0, 4, 1, 2, 3 >

Se espera como resultado: (0, 6, 8, 12, 5)

Para el **cuarto** caso se tienen de premisas:

finca = << 3,2,4>>

programación = < 0 >

Se espera obtener: (0), pues al sólo haber un tablón, su tiempo de inicio de riego siempre será 0

Para el **quinto** caso se tienen de premisas:

finca = << 2, 4, 1 >, < 1, 4, 5 >>

programación = < 1,2 >

Se espera: (0, 4)

Luego, los resultados obtenidos fueron:

```
val resultado1: Riego.TiempoInicioRiego = Vector(0, 2, 10, 9, 5)
val resultado2: Riego.TiempoInicioRiego = Vector(0, 6, 9, 11, 2)
val resultado3: Riego.TiempoInicioRiego = Vector(0, 6, 8, 12, 5)
val resultado4: Riego.TiempoInicioRiego = Vector(0)
val resulado5: Riego.TiempoInicioRiego = Vector(0, 4)
```

Se observa entonces, que se obtuvieron los resultados esperados.

2. Función *costoRiegoTablon*

2.1. Estructuras de datos empleadas

La implementación de la función $costoRiegoTablon$:

```
def costoRiegoTablon(i: Int, f: Finca, pi: ProgRiego): Int = {
  val tiempoInicioRiego = tIR(f,pi)
  val tiempoSupervivencia = tsup(f,i)
  val tiempoRiego = treg(f,i)
  val prioridadTablon = prio(f,i)

  if (tiempoSupervivencia - tiempoRiego > tiempoInicioRiego(i)) {
    tiempoSupervivencia - (tiempoInicioRiego(i) + tiempoRiego)
    }
  else {
    prioridadTablon * ((tiempoInicioRiego(i) + tiempoRiego) - tiempoSupervivencia)
  }
}
```

La función $costoRiegoTablon$ calcula el costo de riego para un tablón específico en una finca, dado una programación de riego.

La función toma tres parámetros: i representa el índice del tablón en la finca, f es la finca en sí misma y pi es la programación de riego.

Primero, la función accede al tiempo de inicio de riego de cada tablón en la finca a través de la función $tIR(f, pi)$. Luego, obtiene el tiempo de supervivencia del tablón i mediante la función $tsup(f, i)$. También obtiene el tiempo de riego del tablón i utilizando la función $treg(f, i)$. Y finalmente, obtiene la prioridad del tablón i con la función $prio(f, i)$.

A continuación, se verifica una condición: si el tiempo de supervivencia del tablón i menos el tiempo de riego es mayor que el tiempo de inicio de riego del tablón i, entonces se cumple que el tablón puede ser regado dentro del tiempo disponible sin afectar su supervivencia. En este caso, la función simplemente devuelve la diferencia entre el tiempo de supervivencia y la suma del tiempo de inicio de riego y el tiempo de riego. Esto significa que el costo de riego para este tablón es cero, ya que no hay necesidad de ajustar el tiempo de riego.
Sin embargo, si la condición anterior no se cumple, significa que el tiempo de supervivencia del tablón i es menor o igual al tiempo de inicio de riego más el tiempo de riego. En este caso, se necesita ajustar el tiempo de riego para cumplir con el tiempo de supervivencia y se utiliza la prioridad del tablón para determinar el costo de este ajuste. El costo se calcula multiplicando la prioridad del tablón por la diferencia entre la suma del tiempo de inicio de riego y el tiempo de riego, y el tiempo de supervivencia.

2.2. Casos de prueba

Los casos de prueba que se establecieron fueron:

Para el **primer** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 1, 4, 3, 2 >

tablón: 0

Como 1 – 2 < 0

entonces 3 * (0 + 2) - 1

Por lo tanto, se espera: 3

Para el **segundo** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 4, 1, 2, 3 >

tablón: 1

Se espera: 10

Para el **tercer** caso se tienen de premisas:

finca = << 1, 5, 5 >, < 4, 2, 5 >, < 1, 4, 2 >< 2, 4, 2 >, < 3, 1, 3 >>

programación = < 0, 4, 1, 2, 3 >

tablón: 0

Se espera: 9

Para el **cuarto** caso se tienen de premisas:

finca = << 3,2,4>>

programación = < 0 >

tablón: 0

Se espera: 1

Para el **quinto** caso se tienen de premisas:

finca = << 2, 4, 1 >, < 1, 4, 5 >>

programación = < 0, 1 >

tablón: 1

Se espera: 35

```
val primerCaso: Int = 3
val segundoCaso: Int = 10
val tercerCaso: Int = 9
val cuartoCaso: Int = 1
val quintoCaso: Int = 35
```

Se lograron los resultados esperados!

3. Función *costoRiegoFinca*

3.1. Estructuras de datos empleadas

La siguiente imagen corresponde con la implementación de la función *costoRiegoFinca*:

```
def costoRiegoFinca(f: Finca, pi: ProgRiego): Int = {
  val costos = for {
    id_tablon <- pi
  } yield costoRiegoTablon(id_tablon, f, pi)
  costos.sum
}
```

Note que *costoRiegoFinca* recibe una finca f y una programación de riego pi, y retorna un entero, que representa el costo de regar dicha finca.

Para ello, se hace uso de una expresión for, la cual usa el iterador id_tablon, para obtener cada tablón perteneciente a la programación de riego pi. Luego retorna una secuencia de enteros resultante de calcular el costo de riego de cada tablón en pi, haciendo uso de la función *costoRiegoTablon*, pasándole como argumentos, el id_tablon, la finca f y la programación pi.

Finalmente, estos costos individuales se suman para así obtener el costo de riego total de la finca.

3.2. Casos prueba

3.3.

Los casos de prueba que se establecieron fueron

Para el **primer** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 1, 4, 3, 2 >

Se espera: 75

Para encontrar el costo de riego de la finca, debemos sumar todos los costos de riego de los tablones que la componen respecto a la programación que se tiene, en este caso son:

3 + 2 + 44 + 20 + 6 = 75

Para el **segundo** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 4, 1, 2, 3 >

Se espera: 81

Para el **tercer** caso se tienen de premisas:

finca = << 1, 5, 5 >, < 4, 2, 5 >, < 1, 4, 2 >< 2, 4, 2 >, < 3, 1, 3 >>

programación = < 0, 4, 1, 2, 3 >

Se espera: 99

Para el **cuarto** caso se tienen de premisas:

finca = << 3,2,4>>

programación = < 0 >

Se espera: 1

Para el **quinto** caso se tienen de premisas:

finca = << 2, 4, 1 >, < 1, 4, 5 >>

programación = < 0, 1 >

Se espera: 37

```
val primerCaso: Int = 75
val segundoCaso: Int = 81
val tercerCaso: Int = 99
val cuartoCaso: Int = 1
val quintoCaso: Int = 37
```

¡Se lograron los resultados esperados!

3.4. Argumentación sobre la corrección

Sea P_f la función $costoRiegoFinca$ implementada en Scala.
Sea $\mathcal{F}$ una finca con n tablones, es decir:

$$\mathcal{F} = Vector(\mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1}), n \geq 1$$

Donde cada $\mathcal{T}_i$ es una tupla $< ts_i^{\mathcal{F}}, tr_i^{\mathcal{F}}, p_i^{\mathcal{F}} >$, y cada atributo representa el tiempo de supervivencia, el tiempo de regado y la prioridad del tablón i en la finca $\mathcal{F}$, respectivamente.

Y sea $\mathcal{P}$ una programación de los n tablones, es decir:

$$\mathcal{P} = Vector(0,1, \dots, n-1), n \geq 1$$

Donde el orden de los n tablones en la programación $\mathcal{P}$ puede ser distinto, pues una programación de riego es una permutación $\mathcal{P} =< \mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1} >$ de $< 0,1, \dots, n-1 >$ que indica el orden en que se regaran los tablones.
Se desea demostrar entonces que:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego: P_f(\mathcal{F}, \mathcal{P}) \rightarrow CR_{\mathcal{F}}^{\mathcal{P}} \in \mathbb{N}$$

Donde $CR_{\mathcal{F}}^{\mathcal{P}}$ corresponde al valor numérico del costo de regar la finca $\mathcal{F}$ con la programación $\mathcal{P}$.

Para ello, se usará el modelo de substitución.

Inicialmente, la traducción de la expresión for en términos de map sería:

$$costos = \text{pi}.map\big(id\_tablon \rightarrow costoRiegoTablon(id\_tablon, f, pi)\big)$$

Substituyendo:

$$P_f(\mathcal{F}, \mathcal{P}) \rightarrow \text{costos} = \mathcal{P}.map\big(id\_tablon \rightarrow costoRiegoTablon(id\_tablon, \mathcal{F}, \mathcal{P})\big)$$

Suponiendo el correcto funcionamiento de map, a cada tablón de la programación $\mathcal{P}$ se le aplicará la función $costoRiegoTablon$, así:

$$P_f(\mathcal{F},\mathcal{P}) \rightarrow costos$$
$$= Vector(costoRiegoTablon(0,\mathcal{F},\mathcal{P}), costoRiegoTablon(1,\mathcal{F},\mathcal{P}), \ldots,$$
$$costoRiegoTablon(n-1,\mathcal{F},\mathcal{P}))$$

Luego, por corrección de $costoRiegoTablon$, sabemos que cada llamada a esta función retorna un entero que representa el costo de riego de cada tablón en la programación $\mathcal{P}$, es decir:

$$P_f(\mathcal{F},\mathcal{P}) \rightarrow costos = Vector\left(CR_{\mathcal{F}}^{\mathcal{P}}[0], CR_{\mathcal{F}}^{\mathcal{P}}[1], \ldots, CR_{\mathcal{F}}^{\mathcal{P}}[n-1]\right)$$

Finalmente, asumiendo el correcto funcionamiento de la función sum, se sumarán los costos de riego de cada tablón de la finca $\mathcal{F}$ y la programación $\mathcal{P}$, es decir:

$$P_f(\mathcal{F},\mathcal{P}) \rightarrow costos.sum = \sum_{i=0}^{n-1} CR_{\mathcal{F}}^{\mathcal{P}}[i] = CR_{\mathcal{F}}^{\mathcal{P}}$$

Así pues, queda demostrado utilizando el modelo de substitución que:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego: P_f(\mathcal{F},\mathcal{P}) \rightarrow CR_{\mathcal{F}}^{\mathcal{P}} \in \mathbb{N}$$

3.5. Versión paralela

La versión paralela de $costoRiegoTablón$ se implementó de la siguiente manera:

```
def costoRiegoFincaPar(f: Finca, pi: ProgRiego): Int = {
  val costos = for {
    id_tablon <- pi.par
  } yield costoRiegoTablon(id_tablon, f, pi)
  costos.sum
}
```

Observe que la implementación de la función es similar a su versión secuencial, pero ahora se utiliza la técnica de paralelización de datos mediante el método par para convertir la colección pi en un $ParVector$.

Es importante destacar que en este caso podemos utilizar este $ParVector$ sin problemas, ya que el cálculo del costo de riego para cada tablón es independiente entre sí, lo que garantiza la coherencia de los resultados obtenidos. Además, debido

a esta independencia de la información, sería muy beneficioso calcular el costo de riego de cada tablón de forma paralela.

4. Función *costoMovilidad*

4.1. Estructuras de datos empleadas

La implementación de la función *costoMovilidad* fue la siguiente:

```
def costoMovilidad(f: Finca, pi: ProgRiego, d: Distancia): Int = {
  val parejasTablones = for {
    i <- 0 ≤ to ≤ pi.length - 2
  } yield (pi(i), pi(i + 1))
  val distanciaPorPareja = parejasTablones.map{case (x, y) => d(x)(y)}
  distanciaPorPareja.sum
}
```

La función *costoMovilidad*, calcula el costo de movilidad asociado a una programación de riego en una finca.
La función toma tres parámetros: f que representa la finca, pi que es la programación de riego y d que representa la distancia entre los tablones. El objetivo es calcular la distancia total recorrida al desplazarse de un tablón a otro en el orden especificado por la programación de riego.
La función utiliza diferentes estructuras de datos y expresiones para realizar el cálculo:

1. $parejasTablones$ es una variable que almacena todas las parejas de tablones consecutivos en la programación de riego. Utiliza una expresión for para iterar desde el primer elemento hasta el penúltimo elemento de la programación pi. Cada iteración genera una pareja de tablones consecutivos.
2. $distanciaPorPareja$ es una variable que almacena las distancias entre cada pareja de tablones. Utiliza el método map sobre la colección $parejasTablones$ para aplicar una función a cada pareja. La función extrae los elementos de cada pareja (representados por las variables x e y) y utiliza la matriz de distancias d para obtener la distancia entre ellos.
3. Finalmente, la función suma todas las distancias almacenadas en $distanciaPorPareja$ utilizando el método sum para obtener el costo total de movilidad.

4.2. Casos Prueba

Los casos de prueba que se establecieron fueron

Para el **primer** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 1, 4, 3, 2 >

Distancia al azar de longitud 5

Se espera: 24

Para el **segundo** caso se tienen de premisas:

finca = << 1, 2, 3 >, < 4, 3, 2 >, < 1, 2, 4 >< 5, 1, 4 >, < 6, 4, 2 >>

programación = < 0, 4, 1, 2, 3 >

Distancia al azar de longitud 5

Se espera: 1

Para el **tercer** caso se tienen de premisas:

finca = << 1, 5, 5 >, < 4, 2, 5 >, < 1, 4, 2 >< 2, 4, 2 >, < 3, 1, 3 >>

programación = < 0, 4, 1, 2, 3 >

Distancia al azar de longitud 5

Se espera: 14

Para el **cuarto** caso se tienen de premisas:

finca = << 3,2,4>>

programación = < 0 >

Distancia al azar de longitud 1

Se espera: 0

Para el **quinto** caso se tienen de premisas:

finca = << 2, 4, 1 >, < 1, 4, 5 >>

programación = < 0, 1 >

Distancia al azar de longitud 2

Se espera: 1

```
val primerCasoM: Int = 24
val segundoCasoM: Int = 14
val tercerCasoM: Int = 14
val cuartoCasoM: Int = 0
val quintoCasoM: Int = 1
```

4.3. Argumentación sobre la corrección

Sea P_f la función $costoMovilidad$ implementada en Scala.
Sea $\mathcal{F}$ una finca con n tablones, es decir:

$$\mathcal{F} = Vector(\mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1}), n \geq 1$$

Donde cada $\mathcal{T}_i$ es una tupla $< ts_i^{\mathcal{F}}, tr_i^{\mathcal{F}}, p_i^{\mathcal{F}} >$, y cada atributo representa el tiempo de supervivencia, el tiempo de regado y la prioridad del tablón i en la finca $\mathcal{F}$, respectivamente.

Sea $\mathcal{P}$ una programación de los n tablones, es decir:

$$\mathcal{P} = Vector(0,1, \dots, n-1), n \geq 1$$

Donde el orden de los n tablones en la programación $\mathcal{P}$ puede ser distinto, pues una programación de riego es una permutación $\mathcal{P} =< \mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{n-1} >$ de $< 0,1, \dots, n-1 >$ que indica el orden en que se regaran los tablones.

Y sea $\mathcal{D}$ una matriz de distancia entre cada par de tablones de la finca $\mathcal{F}$, donde $\mathcal{D}_{\mathcal{F}}[i,j]$ representa el costo de mover el sistema de riego del tablón $\mathcal{T}_i$ al tablón $\mathcal{T}_j$.
Además $\mathcal{D}_{\mathcal{F}}[i,i] = 0$ y $\mathcal{D}_{\mathcal{F}}[i,j] = \mathcal{D}_{\mathcal{F}}[j,i]$, $i,j \in [0,n)$.

Se desea demostrar entonces que:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego, \forall \mathcal{D} \in Distancia: P_f(\mathcal{F}, \mathcal{P}, \mathcal{D}) \rightarrow CM_{\mathcal{F}}^{\mathcal{P}} \in \mathbb{N}$$

Donde $CM_{\mathcal{F}}^{\mathcal{P}}$ corresponde al valor numérico del costo de movilizar el sistema de riego en la finca $\mathcal{F}$ usando la programación $\mathcal{P}$.

Para ello, se usará el modelo de substitución.
Inicialmente, la traducción de la expresión for en términos de map sería:

$$parejasTablones = (0 \text{ to pi.length} - 2).map\Big(i \rightarrow \big(pi(i), pi(i+1)\big)\Big)$$

Substituyendo:

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow parejasTablones = (0\ to\ \mathcal{P}.\text{length} - 2).map\left(i \rightarrow \left(\mathcal{P}(i),\mathcal{P}(i+1)\right)\right)$$

Suponiendo el correcto funcionamiento de la función map, se genera una tupla para cada elemento en el rango de 0 a $\mathcal{P}.length\ -\ 2$ (donde $\mathcal{P}.length$ representa la cantidad de tablones). Cada tupla consta de dos elementos: el primer elemento es el tablón i-ésimo de la programación $\mathcal{P}$, y el segundo elemento es el tablón consecutivo en la programación, es decir, el tablón i-ésimo + 1.

Así, $parejasTablones$ se puede reescribir de la siguiente manera:

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow parejasTablones = Seq\left(\left(\mathcal{P}(0),\mathcal{P}(1)\right),\dots,\left(\mathcal{P}(i-2),\mathcal{P}(i-1)\right)\right)$$

Luego, para cada tupla de $parejasTablones$ se le aplica una función que retorna la distancia que hay entre los tablones de la tupla, haciendo uso de la matriz de distancias $\mathcal{D}$.

Substituyendo:

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow distanciaPorPareja = Seq\left(\left(\mathcal{P}(0),\mathcal{P}(1)\right),\dots,\left(\mathcal{P}(i-2),\mathcal{P}(i-1)\right)\right).map\{\ case(x,y) \Rightarrow \mathcal{D}(x)(y)\}$$

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow distanciaPorPareja = Seq\big(\mathcal{D}(0)(1),\dots,\mathcal{D}(i-2)(i-1)\big)$$

Luego, sabemos que al averiguar la distancia de un tablón $\mathcal{T}_i$ con un tablón $\mathcal{T}_j$ en la matriz D, obtendremos un número natural, que corresponde con el costo de movilizar el sistema de riego desde el tablón $\mathcal{T}_i$ al tablón $\mathcal{T}_j$, es decir:

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow distanciaPorPareja = Seq(\mathcal{D}_{\mathcal{F}}[0,1],\dots,\mathcal{D}_{\mathcal{F}}[i-2,i-1])$$

Finalmente, suponiendo el correcto funcionamiento de la función sum, se sumarán los costos de movilización entre cada par de tablones en $distanciaPorPareja$, es decir:

$$P_f(\mathcal{F},\mathcal{P},\mathcal{D}) \rightarrow distanciaPorPareja.sum\ = \sum_{i=0}^{\mathcal{P}.\text{length}-2} \mathcal{D}_{\mathcal{F}}[\mathcal{T}_i,\mathcal{T}_{i+1}] = CM_{\mathcal{F}}^{\mathcal{P}}$$

Así pues, queda demostrado utilizando el modelo de substitución que:

$$\forall \mathcal{F} \in Finca, \forall \mathcal{P} \in ProgRiego, \forall \mathcal{D} \in Distancia: P_f(\mathcal{F}, \mathcal{P}, \mathcal{D}) \rightarrow CM_{\mathcal{F}}^{\mathcal{P}} \in \mathbb{N}$$

4.4. Versión paralela

```
def costoMovilidadPar(f: Finca, pi: ProgRiego, d: Distancia): Int = {

  def costoMovilidadAux(intervalo: (Int,Int)): Int = {
    val tablones = for {
      i <- intervalo._1 ≤ until < intervalo._2
    } yield (pi(i), pi(i + 1))
    tablones.map { case (x, y) => d(x)(y) }.sum
  }

  val umbral = 800
  if (pi.length <= umbral) {
    costoMovilidad(f, pi, d)
  } else {
    val n = pi.length
    val cuarto = n / 4
    val resto = n % 4
    val divisiones = (0 ≤ until < 4).toVector.map { i =>
      val inicio = i * cuarto + math.min(i, resto)
      val fin = inicio + cuarto + (if (i < resto) 1 else 0)
      (inicio,fin)
    }
    val resul = parallel(costoMovilidadAux(divisiones(0)),
      costoMovilidadAux(divisiones(1)),
      costoMovilidadAux(divisiones(2)),
      costoMovilidadAux(divisiones(3)._1,divisiones(3)._2 - 1))
    resul._1 + resul._2 + resul._3 + resul._4
  }
}
```

La versión paralela de la función $costoMovilidad$ utiliza la paralelización de tareas, y se divide en 4 partes para calcular el costo de movilidad de cada segmento de la programación en paralelo. Esto se logra mediante la función auxiliar $costoMovilidadAux$, que sigue la misma lógica que la versión secuencial, pero opera en un intervalo específico de la programación.

La razón por la que la versión paralela funciona correctamente se basa en dos factores clave. Primero, la función auxiliar $costoMovilidadAux$ realiza exactamente la misma tarea que la función $costoMovilidad$ secuencial, pero se limita a un intervalo particular de la programación. Dado que la versión secuencial ya ha sido demostrada como funcional, podemos confiar en que la lógica de la función auxiliar es correcta.

En segundo lugar, al dividir la programación en 4 partes y calcular el costo de movilidad de cada una en paralelo, la versión paralela aprovecha el poder de la concurrencia para acelerar el procesamiento. Cada parte del cálculo se ejecuta de forma independiente y simultánea, lo que permite un uso más eficiente de los recursos del sistema y una reducción significativa en los tiempos de ejecución.

Finalmente, el resultado de sumar cada uno de los 4 valores calculados en paralelo, será el costo total de movilizar el sistema en la finca f con la programación pi, pues la suma cumple con la propiedad de ser conmutativa.

En resumen, la versión paralela de $costoMovilidad$ funciona correctamente porque se basa en la lógica probada de la versión secuencial, pero divide la tarea en segmentos paralelos para aprovechar la concurrencia y acelerar el procesamiento.

5. Función $generarProgramacionesRiego$

5.1. Estructuras de datos empleadas

La implementación de la función ***generarProgramacionesRiego*** fue la siguiente:

```
def generarProgramacionesRiego(f: Finca): Vector[ProgRiego] = {
  val indices = f.indices.toVector

  def generarPermutaciones(indices: Vector[Int]): Vector[ProgRiego] = {
    if (indices.isEmpty) {
      Vector(Vector.empty[Int]) // Caso base: lista vacía, retornar una programación vacía
    } else {
      indices.flatMap { i =>
        generarPermutaciones(indices.filterNot(_ == i)).map(i +: _)
      }
    }
  }
  generarPermutaciones(indices)
}
```

La función ***generarProgramacionesRiego*** se encarga de generar todas las posibles programaciones de riego para una finca dada. La función recibe como parámetro una finca, que es un vector de tablones. Un tablón es una tupla de tres valores enteros: (tiempo de supervivencia, tiempo de riego, prioridad). La función devuelve un vector de programaciones de riego, donde cada programación es un vector de enteros.

Esta función utiliza flatMap para generar todas las permutaciones posibles de los índices de los tablones en la finca además combina el uso de map y flatten, y permite generar nuevas colecciones a partir aplicando una función a cada elemento y luego aplanando los resultados en una única colección.

La función ***generarProgramacionesRiego*** sigue un enfoque recursivo para generar las permutaciones. En cada llamada recursiva, se selecciona un índice de tablón y se genera de forma recursiva todas las permutaciones restantes excluyendo ese índice. Luego, se combina el índice seleccionado con cada permutación generada para formar nuevas programaciones.

5.2. Argumentación sobre la corrección

Para comenzar, definamos algunas variables:

- Sea $\boldsymbol{f(n)}$ el conjunto de todas las fincas de tamaño $\boldsymbol{n}$.

- Sea $\boldsymbol{\rho(n)}$ el conjunto de todas las programaciones de riego posibles para una finca de tamaño $\boldsymbol{n}$.

Caso base:

Para el caso base, verificamos que la función *generarProgramacionesRiego* devuelve un conjunto vacío cuando se aplica a una finca vacía.

Demostración:

Para $n = 0$ tenemos $f(0) = \{\}$ (conjunto vacío). Por lo tanto, *generarProgramacionesRiego*($F(0)$) = {}, que es el conjunto vacío. Esto cumple con el caso base.

Paso inductivo:

Supongamos que la función *generarProgramacionesRiego* produce el resultado correcto para una finca de tamaño $\boldsymbol{n}$, es decir, devuelve el conjunto $\boldsymbol{P(n)}$ de programaciones de riego válidas.

Queremos demostrar que la función también produce el resultado correcto para una finca de tamaño $\boldsymbol{n+1}$, es decir, devuelve el conjunto $\boldsymbol{P(n+1)}$ de programaciones de riego válidas.

Demostración:

Consideremos una finca $\boldsymbol{F(n+1)}$. Esta finca se puede construir agregando un tablón adicional al conjunto de tablones de una finca $\boldsymbol{F(n)}$. Sea $\boldsymbol{t}$ el tablón adicional que se agrega.

Para obtener $\boldsymbol{P(n+1)}$, podemos considerar dos casos:

1) Programaciones que incluyen el tablón adicional $\boldsymbol{t}$**:**

Estas programaciones se obtienen tomando una programación válida $\boldsymbol{p}$ en P(n) y agregando $\boldsymbol{t}$ al final de la programación. Matemáticamente, podemos expresarlo como:

P1 = $\{ p \cup \{t\} : p \in P(n) \}$

2) Programaciones que no incluyen el tablón adicional $\boldsymbol{t}$**:**

Estas programaciones se obtienen tomando todas las programaciones válidas en $\boldsymbol{P(n)}$ y dejándolas sin cambios. Podemos expresarlo como:

P2 = $P(n)$

El conjunto $\boldsymbol{P(n+1)}$ de programaciones de riego válidas para la finca $\boldsymbol{F(n+1)}$ se puede obtener tomando la unión de los conjuntos P1 y P2:

$\boldsymbol{P(n+1)}$ = P1 ∪ P2

Ahora, debemos demostrar que $\boldsymbol{P(n+1)}$ cumple con la propiedad de tener todas las programaciones de riego válidas para $\boldsymbol{F(n+1)}$.

Para demostrarlo, debemos mostrar dos cosas:

a) Cada programación en $\boldsymbol{P(n+1)}$ es válida para $\boldsymbol{F(n+1)}$

b) Toda programación válida para $\boldsymbol{F(n+1)}$ está en $\boldsymbol{P(n+1)}$.

a) Demostración de la propiedad a):

Tomemos una programación $\boldsymbol{p}$ en P1, es decir, $p = q \cup \{t\}$, donde $\boldsymbol{q}$ es una programación válida en $\boldsymbol{P(n)}$. La programación $\boldsymbol{p}$ tiene el tablón adicional $\boldsymbol{t}$ al final. Dado que $\boldsymbol{q}$ es una programación válida para $\boldsymbol{F(n)}$, sabemos que no contiene repeticiones de tablones y cumple con las condiciones necesarias. Agregar el tablón $\boldsymbol{t}$ al final no afecta la validez de la programación para $\boldsymbol{F}$

En conclusión, utilizando la inducción matemática e hipótesis de inducción, hemos demostrado que la función *generarProgramacionesRiego* es correcta para cualquier tamaño de finca.

5.3. Versión paralela

```
def generarProgramacionesRiegoPar(f: Finca): ParVector[ProgRiego] = {
  val indices = f.indices.toVector

  def generarPermutaciones(indices: Vector[Int]): Vector[ProgRiego] = {
    if (indices.isEmpty) {
      Vector(Vector.empty[Int]) // Caso base: lista vacía, retornar una programación vacía
    } else {
      indices.flatMap{i =>
        generarPermutaciones(indices.filterNot(_ == i)).map(i +: _)
      }
    }
  }

  def generarPermutacionesPar(indices: Vector[Int]): ParVector[ProgRiego] = {
    if (indices.isEmpty) {
      ParVector(Vector.empty[Int]) // Caso base: lista vacía, retornar una programación vacía
    } else {
      indices.par.flatMap { i =>
        generarPermutaciones(indices.filterNot(_ == i)).map(i +: _)
      }
    }
  }
  generarPermutacionesPar(indices)
}
```

La versión paralela de la función $generarProgramacionesRiego$ usa la técnica de paralelización de datos. Esta se implementa con el objetivo de mejorar el rendimiento al generar permutaciones para conjuntos de datos grandes. Para comprender por qué esta versión funciona correctamente, podemos analizar su lógica en comparación con la versión secuencial.

Por corrección de $generarProgramacionesRiego$, sabemos que la función funciona correctamente. La idea principal es generar todas las permutaciones posibles a partir de un conjunto de índices. Esto se logra mediante la recursión y la concatenación de los índices en diferentes órdenes.

En la versión paralela, se introduce una función auxiliar llamada $generarPermutacionesPar$ que utiliza la estructura de datos paralela $ParVector$ en lugar de $Vector$ para almacenar las permutaciones. Esto permite aprovechar el paralelismo proporcionado por la librería de concurrencia.

La lógica de generación de permutaciones es similar en ambas versiones: se recorre cada índice del conjunto y se generan las permutaciones restantes recursivamente. La diferencia radica en el uso de $indices.par.flatMap$ en la versión paralela, lo que crea una versión paralela de la secuencia de índices y aplica la función $flatMap$ en paralelo para generar las permutaciones.

En resumen, la versión paralela de $generarProgramacionesRiegoPar$ divide el trabajo de generación de permutaciones entre múltiples hilos de ejecución, lo que puede mejorar significativamente el rendimiento, especialmente para conjuntos de datos grandes. Dado que la lógica de generación de permutaciones es la misma que en la versión secuencial, podemos inferir que la versión paralela también funcionará correctamente, aprovechando el paralelismo para acelerar el proceso de generación de permutaciones.

6. Función *programacionRiegoOptimo*

6.1. Estructuras de datos empleadas

La implementación de la función ***programacionRiegoOptimo*** fue la siguiente:

```
def programacionRiegoOptimo(f: Finca, d: Distancia): (ProgRiego, Int) = {
  val programaciones = generarProgramacionesRiego(f)
  val costosRiego = programaciones.map { prog => (prog, costoMovilidad(f, prog, d) + costoRiegoFinca(f, prog)) }
  costosRiego.minBy(_._2)
}
```

La función ***programacionRiegoOptimo*** tiene como objetivo calcular la programación óptima de riego para una finca y una matriz de distancias dadas. A continuación, se detalla su funcionamiento:

Se comienza generando todas las posibles programaciones de riego para la finca utilizando la función ***generarProgramacionesRiego***. Esta función genera todas las combinaciones posibles de tablones de la finca.

Luego se itera sobre cada una de las programaciones generadas y se calcula su costo total. El costo total de una programación se compone de dos partes: el Costo de movilidad y el Costo de riego de la finca.

Después de calcular el costo total para cada programación de riego, se utiliza la función *minBy* para encontrar la programación con el costo total mínimo. En este caso la utilizamos para encontrar la programación de riego que tenga el costo total más bajo.

En resumen, la función ***programacionRiegoOptimo*** utiliza la generación de todas las posibles programaciones de riego, calcula el costo total para cada una y selecciona la programación con el menor costo total como la programación óptima.

Esta función utiliza principalmente las estructuras de datos, así como expresiones y un bucle foreach para generar las programaciones de riego, calcular los costos y encontrar la programación óptima.

6.2. Argumentación sobre la corrección

Supongamos que la función ***programacionRiegoOptimo*** no encuentra la programación de riego óptima para una finca con ***n*** tablones. Denotemos la programación óptima encontrada por la función como *P_opt*.

Supongamos que hay otra programación *P_distinta* con un costo total menor que *P_opt*. Denotemos el costo total de *P_distinta* como *C_distinta* y el costo total de *P_opt* como *C_opt.*

Entonces, tenemos la siguiente afirmación:

$$Existencia\ de\ P\_distinta\text{: } (\exists\ P\_distinta)\ (C\_distinta\ <\ C\_opt)$$

Sin embargo, esto contradice la definición de la función ***programacionRiegoOptimo***, que selecciona la programación con el menor costo total entre todas las programaciones posibles. Por lo tanto, tenemos la siguiente afirmación:

$$Contradicción\text{: } \neg(\exists\ P\_distinta)\ (C\_distinta\ <\ C\_opt)$$

La contradicción entre la existencia de *P_distinta* y la definición de la función ***programacionRiegoOptimo*** implica que nuestra suposición inicial es falsa. Por lo tanto, podemos concluir que la función ***programacionRiegoOptimo*** encuentra la programación de riego óptima para cualquier número natural ***n*** de tablones en la finca.

En resumen, hemos demostrado mediante contradiccion que la función ***programacionRiegoOptimo*** encuentra la programación de riego óptima para una finca con **n** tablones.

6.3. Versión paralela

La paralelización de $programaciónRiegoOptimo$ se implementó de la siguiente manera:

```
def programacionRiegoOptimoPar(f: Finca, d: Distancia): (ProgRiego, Int) = {
  val programaciones = generarProgramacionesRiegoPar(f)
  val costosRiego = programaciones.map { prog =>
    (prog, costoMovilidadPar(f, prog, d) + costoRiegoFincaPar(f, prog))
  }
  costosRiego.minBy(_._2)
}
```

El objetivo de la función $programacionRiegoOptimoPar$ es mejorar el rendimiento al encontrar la programación óptima que minimiza los costos de movilidad y riego en la finca f.
Es importante destacar que, a pesar de compartir la misma lógica con la versión secuencial, la versión paralela de la función utiliza las variantes paralelas de las funciones $costoMovilidad$, $costoRiegoFinca$ y $programacionRiegoOptimo$. Estas versiones paralelas implementan técnicas tanto de paralelización de tareas como de paralelización de datos, lo que se espera que resulte en una mejora significativa en los tiempos de ejecución en comparación con las versiones secuenciales.

Cabe mencionar que cada una de las versiones paralelas utilizadas en esta función ha sido diseñada para aprovechar el paralelismo de manera eficiente, distribuyendo las tareas entre múltiples hilos de ejecución y procesando los datos de forma simultánea.

En conclusión, debido a que la lógica de la función $programacionRiegoOptimo$ es idéntica en ambas versiones, podemos afirmar con confianza que la versión paralela también funcionará correctamente. Sin embargo, gracias al uso del paralelismo en las funciones auxiliares, se espera obtener una mejora notable en el rendimiento, lo que se traducirá en tiempos de ejecución más rápidos.

6.4. Casos de prueba

Se definieron las siguientes pruebas:

```
// Pruebas de programacionRiegoOptimo
val primerCasoRO = programacionRiegoOptimo(finca10,distancia5)
val segundoCasoM = programacionRiegoOptimo(finca11,distancia5)
val tercerCasoM = programacionRiegoOptimo(finca12,distancia5)
val cuartoCasoM = programacionRiegoOptimo(finca13,distancia1)
val quintoCasoM = programacionRiegoOptimo(finca14,distancia2)
```

Se obtuvieron los siguientes resultados:

```
val primerCasoRO: (Riego.ProgRiego, Int) = (Vector(2, 0, 3, 4, 1),52)
val segundoCasoRO: (Riego.ProgRiego, Int) = (Vector(4, 1, 0, 2, 3),106)
val tercerCasoRO: (Riego.ProgRiego, Int) = (Vector(3, 4, 1, 0, 2),43)
val cuartoCasoRO: (Riego.ProgRiego, Int) = (Vector(0),1)
val quintoCasoRO: (Riego.ProgRiego, Int) = (Vector(1, 0),25)
```

Note que, en casos como el cuarto caso, la programación óptima coincide justamente con la única programación que se puede hacer, que es la del único tablón que tiene la finca.

Evaluación comparativa de las soluciones secuenciales versus las paralelas

A continuación, se presentan cuatro tablas, donde se resumen los resultados de ejecutar el algoritmo secuencial y paralelo de cuatro funciones: $costoRiegoFinca$, $costoMovilidad$, $generarProgramacionesRiego$ y $programacionRiegoOptimo$.

Para la tabla I y II, se encuentran los resultados de 22 tamaños distintos de fincas, desde una finca de 10 tablones hasta una de 5000. Para cada tamaño se ejecutó la misma prueba 3 veces, a fin de aumentar la confiabilidad de los resultados obtenidos.

Para la tabla III y IV, se analizaron 5 tamaños distintos de fincas: 2,4,6,8 y 10. El motivo de probar las funciones $generarProgramacionesRiego$ y $programacionRiegoOptimo$ con fincas relativamente pequeñas, es porque estas funciones tienen una complejidad factorial, lo que significa que tanto el tiempo de ejecución como el costo computacional de las funciones, crece rápidamente a medida que se aumenta el tamaño de la finca.

Esto se vio reflejado en que, para una finca con 11 tablones, el programa pasado unos minutos abortaba con el error de $OutOfMemory$, pues la cantidad de operaciones a manejar es muy alta. Estamos hablando de que, para la finca de 10 tablones, se tienen que hacer 10! llamadas recursivas, es decir 3.628.800 llamadas.

Todas las pruebas fueron diseñadas de manera muy similar, la Figura X muestra la estructura usada para la función $costoRiegoFinca$:

Figura 1. Estructura de las pruebas de rendimiento para $costoRiegoFinca$

```
// ---- Prueba 1 ---- || ---- 10 a 500 ---- \\
val CRF1 = for{
  i <- 1 ≤ to ≤ 2
  j <- 1 ≤ to ≤ 5
  finca = fincaAlAzar(math.pow(10,i).toInt * j)
  programacion = programacionAlAzar(math.pow(10,i).toInt * j)
} yield (standardConfig measure costoRiegoFinca(finca,programacion),
  standardConfig measure costoRiegoFincaPar(finca,programacion), math.pow(10,i).toInt * j)
val resultadoCRF1 = CRF1.mkString("\n")
```

En esta prueba de rendimiento se emplea un bucle for con dos iteradores, i y j, que generan diferentes tamaños de pruebas. Dentro del bucle, se genera al azar una finca y una programación del tamaño indicado por i y j. Estos valores se utilizan como parámetros para las funciones $costoRiegoFinca$ y $costoRiegoFincaPar$.

Para medir los tiempos de ejecución, se utiliza la función $measure$ del objeto $standardConfig$. Esta función permite obtener el tiempo que le toma al sistema ejecutar una determinada expresión.

El resultado de la prueba consiste en una tupla de tres elementos. El primer elemento corresponde al tiempo que le toma al sistema ejecutar la versión secuencial de $costoRiegoFinca$ con la finca y la programación generadas. El segundo elemento corresponde al tiempo de ejecución de la versión paralela de $costoRiegoFincaPar$. El tercer elemento es el tamaño de la iteración actual, calculado como $math.pow(10, i).toInt * j$.

Finalmente, se utiliza $yield$ para recopilar los resultados de cada iteración del bucle en una secuencia. El resultado se convierte en una cadena de texto utilizando $mkString("\n")$, donde cada línea de la cadena corresponde a una iteración de la prueba.

Figura 2. Estructura de las pruebas de rendimiento para $costoMovilidad$

```
// ---- Prueba 1 ---- || ---- 10 a 500 ---- \\
val CM1 = for{
  i <- 1 ≤ to ≤ 2
  j <- 1 ≤ to ≤ 5
  finca = fincaAlAzar(math.pow(10,i).toInt * j)
  programacion = programacionAlAzar(math.pow(10,i).toInt * j)
  distancia = distanciaAlAzar(math.pow(10,i).toInt * j)
} yield (standardConfig measure costoMovilidad(finca,programacion, distancia),
  standardConfig measure costoMovilidadPar(finca,programacion, distancia), math.pow(10,i).toInt * j)
val resultadoCM1 = CM1.mkString("\n")
```

Para la Figura 2, note que la estructura de la prueba de $costoMovilidad$ es la misma que la de $costoRiegoFinca$, sólo que ahora se requiere generar además una distancia al azar.

Figura 3. Estructura de las pruebas de rendimiento para $generarProgramacionesRiego$

```
// ---- Prueba 1 ---- || ---- Finca de 2,4,6,8 y 10 tablones ---- \\

val GPR1 = for{
  i <- 1 ≤ to ≤ 5
  finca = fincaAlAzar( long = 2 * i)
} yield (standardConfig measure generarProgramacionesRiego(finca),
  standardConfig measure generarProgramacionesRiegoPar(finca), 2 * i)
val resultadoGPR1 = GPR1.mkString("\n")
```

En la Figura 3 se puede observar que la prueba para *generarProgramacionesRiego* es similar a las anteriores, sólo que ahora sólo se requiere generar una finca al azar.

Finalmente, las pruebas diseñadas para *programacionRiegoOptimo*:

Figura 4. Estructura de las pruebas de rendimiento para *programacionRiegoOptimo*

```
val PRO1 = for{
  i <- 1 ≤ to ≤ 5
  finca = fincaAlAzar( long = 2 * i)
  distancia = distanciaAlAzar( long = 2 * i)
} yield (standardConfig measure programacionRiegoOptimo(finca, distancia),
  standardConfig measure programacionRiegoOptimoPar(finca, distancia), 2 * i)
val resultadoPRO1 = PRO1.mkString("\n")
```

Veamos ahora los resultados obtenidos para cada tabla:

Tabla I: tiempos registrados para la función *costoRiegoFinca*

Tamaño de finca	Tiempos registrados (ms)			Promedios		
	Secuencial (S)	Paralela (P)	Aceleración (A)	$\overline{S}$	$\overline{P}$	$\overline{A}$
10	0.0346	0.1487	0.2328	0.0242	0.1442	0.1678
	0.0194	0.1548	0.1251			
	0.0188	0.1291	0.1453			
20	0.0672	0.1978	0.3397	0.0542	0.1777	0.3028
	0.0479	0.1712	0.2797			
	0.0474	0.1641	0.2890			
30	0.1553	0.2019	0.7695	0.1257	0.1745	0.7164
	0.1120	0.1615	0.6934			
	0.1100	0.1602	0.6864			
40	0.2100	0.1977	1.0623	0.2076	0.2022	1.0274
	0.2075	0.2025	1.0249			
	0.2053	0.2063	0.9951			
50	0.3809	0.2258	1.6869	0.3369	0.2306	1.4640
	0.3194	0.2279	1.4019			
	0.3103	0.2381	1.3030			
100	1.8174	0.7399	2.4562	1.5711	0.5262	3.1188
	1.5310	0.4265	3.5900			
	1.3649	0.4123	3.3103			
200	5.7304	1.1002	5.2084	5.6578	1.1437	4.9521
	5.7502	1.1618	4.9493			
	5.4929	1.1690	4.6986			
	12.7715	2.3642	5.4020	13.1854	2.3609	5.5870

300	13.1629	2.2819	5.7685			
	13.6219	2.4366	5.5906			
400	24.8152	4.2322	5.8634	25.5436	4.3462	5.8772
	26.1053	4.3954	5.9393			
	25.7104	4.4109	5.8288			
500	42.1123	7.0208	5.9982	42.1189	7.2497	5.8134
	43.0024	7.5675	5.6825			
	41.2419	7.1609	5.7593			
600	62.3491	11.1855	5.5741	66.8695	12.0998	5.5254
	77.0793	13.9108	5.5409			
	61.1802	11.2030	5.4611			
700	87.7839	14.6445	5.9943	91.5560	20.4493	4.9276
	92.4701	30.2226	3.0596			
	94.4141	16.4808	5.7287			
800	123.5297	23.0674	5.3552	122.4091	23.7026	5.2806
	119.1229	28.1013	4.2390			
	124.5747	19.9392	6.2477			
900	167.0907	27.0196	6.1840	170.9406	26.9198	6.3510
	178.5441	27.8696	6.4064			
	167.1871	25.8703	6.4625			
1000	214.2275	33.8914	6.3210	206.5955	33.2389	6.2173
	195.6624	33.3821	5.8613			
	209.8966	32.4432	6.4697			
2000	986.3374	162.3659	6.0748	1013.1035	165.1679	6.1333
	1025.7103	166.4573	6.1620			
	1027.2628	166.6806	6.1631			
3000	2348.9182	386.6068	6.0757	2330.2176	393.2716	5.9241
	2436.8549	408.2652	5.9688			
	2204.8796	384.9427	5.7278			
4000	4377.7840	4377.7840	6.1256	4312.1570	720.4210	5.9856
	4383.6906	4383.6906	5.9755			
	4174.9963	4174.9963	5.8557			
5000	6940.7437	1183.6230	5.8640	6686.9517	1187.7316	5.6348
	6601.3327	1234.8262	5.3460			
	6518.7787	1144.7455	5.6945			
6000	8984.0541	1708.2672	5.2592	9280.1208	1687.4573	5.5029
	9454.1810	1717.7405	5.5038			
	9402.1274	1636.3643	5.7457			
7000	13823.0845	2384.6305	5.7967	13227.8358	2378.2878	5.5615
	13174.2214	2377.8056	5.5405			
	12686.2014	2372.4271	5.3474			
8000	19056.6870	3171.4068	6.0089	17800.1073	3146.9158	5.6556
	17837.0305	3122.8985	5.7117			
	16506.6043	3146.4422	5.2461			

Según los resultados de la Tabla I, se observa que la versión paralela de la función $costoRiegoFinca$ presenta una mejora significativa a partir de fincas con 50 tablones. A partir de ese punto, los tiempos de ejecución de las versiones paralelas son notablemente más rápidos que las versiones secuenciales, con un promedio de velocidad alrededor de 5 veces mayor. En varios casos, incluso se alcanzan valores cercanos o superiores a 6 veces más rápidos.

En resumen, la paralelización de la función $costoRiegoFinca$ es altamente beneficiosa a partir de fincas con 50 tablones en adelante. Los tiempos de ejecución se reducen de manera significativa, lo que demuestra la eficiencia de la versión paralela en comparación con la secuencial.

Tabla II: tiempos registrados para la función $costoMovilidad$

Tamaño de finca	Tiempos registrados (ms)			Promedios		
	Secuencial (S)	Paralela (P)	Aceleración (A)	$\overline{S}$	$\overline{P}$	$\overline{A}$
10	0.0036	0.0019	1.9355	0.0034	0.0017	1.9887
	0.0035	0.0018	1.9314			
	0.0031	0.0015	2.0992			
20	0.0023	0.0026	0.8723	0.0027	0.0033	0.9058
	0.0033	0.0051	0.6432			
	0.0025	0.0021	1.2019			
30	0.0030	0.0039	0.7764	0.0030	0.0033	0.9043
	0.0031	0.0032	0.9506			
	0.0028	0.0029	0.9861			
40	0.0065	0.0058	1.1129	0.0050	0.0044	1.1233
	0.0041	0.0040	1.0300			
	0.0043	0.0035	1.2271			
50	0.0092	0.0077	1.1910	0.0083	0.0092	0.9466
	0.0116	0.0091	1.2731			
	0.0040	0.0107	0.3756			
100	0.0075	0.0076	0.9895	0.0056	0.0057	0.9966
	0.0050	0.0049	1.0041			
	0.0044	0.0045	0.9964			
200	0.0150	0.0149	1.0094	0.0110	0.0108	1.0234
	0.0091	0.0093	0.9832			
	0.0090	0.0083	1.0774			
300	0.0226	0.0236	0.9565	0.0163	0.0162	1.0197
	0.0139	0.0123	1.1343			
	0.0124	0.0128	0.9684			
400	0.0311	0.0318	0.9771	0.0206	0.0209	0.9850
	0.0136	0.0138	0.9850			
	0.0170	0.0171	0.9930			
500	0.0428	0.0211	2.0305	0.0280	0.0211	1.3295
	0.0188	0.0190	0.9897			
	0.0225	0.0232	0.9683			

600	0.0419	0.0372	1.1278	0.0323	0.0317	1.0100
	0.0249	0.0257	0.9681			
	0.0301	0.0322	0.9341			
700	0.0508	0.0494	1.0272	0.0419	0.3563	0.6994
	0.0350	0.0339	1.0305			
	0.0399	0.9855	0.0405			
800	0.0640	0.0589	1.0857	0.0521	0.0555	0.9420
	0.0428	0.0578	0.7409			
	0.0497	0.0497	0.9994			
900	0.0680	0.0832	0.8174	0.0640	0.0839	0.8938
	0.0646	0.1243	0.5201			
	0.0594	0.0442	1.3440			
1000	0.0938	0.0599	1.5664	0.0750	0.0480	1.5636
	0.0625	0.0424	1.4731			
	0.0687	0.0416	1.6514			
2000	0.2552	0.0831	3.0712	0.8088	0.1150	6.1676
	1.9552	0.1437	13.6072			
	0.2160	0.1184	1.8244			
3000	0.3563	0.1090	3.2676	0.3009	0.1041	2.8921
	0.2900	0.1128	2.5702			
	0.2564	0.0903	2.8386			
4000	0.3916	0.1367	2.8646	0.3976	0.1349	2.9624
	0.4357	0.1528	2.8508			
	0.3654	0.1152	3.1717			
5000	0.4797	0.1563	3.0686	0.9171	0.1723	5.6838
	1.6862	0.1506	11.1957			
	0.5853	0.2100	2.7870			

Antes de analizar los resultados obtenidos, cabe mencionar que se definieron pruebas para $costoMovilidad$ con fincas de 6000 a 10000 tablones, sin embargo, el programa abortaba con $OutOfMemoryError$

En la Tabla II se pueden observar tiempos similares entre la versión secuencial y paralela de la función $costoRiegoFinca$ hasta llegar a fincas con menos de 1000 tablones. En este rango, las aceleraciones promedio son inferiores o cercanas a 1, lo que indica que la paralelización no resulta muy beneficiosa para estas fincas.

Sin embargo, a partir de fincas con 1000 tablones en adelante, se empiezan a notar tiempos notablemente menores en la versión paralela. Esto sugiere que la técnica utilizada de dividir la programación en 4 partes y calcular en paralelo el costo de movilidad está generando ganancias en los tiempos de ejecución.

Es importante destacar que existen casos donde la aceleración aumenta de manera desproporcionada, como en el caso de fincas con tamaño 2000. Sin embargo, al analizar los tiempos de ejecución de la versión secuencial en la segunda prueba, se puede observar que es casi 2 ms, mientras que en la primera y tercera prueba el valor suele ser cercano a 0.2 ms. Esto indica que hay cierta variabilidad en este tipo de pruebas de rendimiento, no solo debido al hardware

utilizado, sino también a factores internos del software, lo cual genera cierto grado de indeterminismo en los resultados.

Tabla III: tiempos registrados para la función *generarProgramacionesRiego*

Tamaño de finca	Tiempos registrados (ms)			Promedios		
	Secuencial (S)	Paralela (P)	Aceleración (A)	$\overline{S}$	$\overline{P}$	$\overline{A}$
2	0.0072	0.1243	0.0576	0.0038	0.0749	0.0493
	0.0044	0.0994	0.0445			
	0.0027	0.0429	0.0622			
	0.0055	0.0975	0.0565			
	0.0016	0.0469	0.0331			
	0.0050	0.0916	0.0551			
	0.0018	0.0342	0.0532			
	0.0054	0.0910	0.0589			
	0.0010	0.0461	0.0225			
4	0.0309	0.1160	0.2662	0.0235	0.0872	0.2606
	0.0285	0.1055	0.2706			
	0.0266	0.0639	0.4167			
	0.0378	0.1130	0.3347			
	0.0120	0.0664	0.1807			
	0.0256	0.1034	0.2479			
	0.0071	0.0644	0.1104			
	0.0331	0.0982	0.3367			
	0.0098	0.0540	0.1817			
6	0.6154	0.6109	1.0074	0.5189	0.3915	1.4268
	0.3842	0.4781	0.8036			
	0.4524	0.5259	0.8602			
	0.4347	0.5019	0.8660			
	0.3888	0.1785	2.1783			
	1.4844	0.4637	3.2012			
	0.2624	0.1661	1.5803			
	0.3827	0.4108	0.9315			
	0.2649	0.1876	1.4123			
8	14.6182	7.0794	2.0649	15.1396	6.5579	2.3177
	14.3880	6.8516	2.0999			
	14.3049	6.4132	2.2305			
	14.7802	6.9834	2.1165			
	15.0154	6.0973	2.4626			
	14.4464	6.5685	2.1993			
	15.6751	6.4154	2.4434			
	16.4843	6.4797	2.5440			
	16.5444	6.1324	2.6979			
10	1793.1939	1029.1810	1.7424	1858.6780	1082.4614	1.7177
	1772.2528	1076.9204	1.6457			

	1822.8792	1067.0682	1.7083			
	1947.3054	1099.4878	1.7711			
	1845.2910	1088.2545	1.6956			
	1812.6084	1113.4348	1.6279			
	1805.4596	1107.9117	1.6296			
	1989.1901	1068.9265	1.8609			
	1939.9216	1090.9681	1.7782			

Los resultados de la tabla III muestran que la paralelización resulta beneficiosa para tamaños de finca más pequeños. A partir de fincas con 6 tablones en adelante, los tiempos de ejecución en paralelo son notablemente mejores, llegando a ser hasta el doble de rápidos que los tiempos secuenciales.

Estos resultados son coherentes con la naturaleza de la función, ya que se mencionó anteriormente que su complejidad es factorial. Esto significa que la cantidad de operaciones a realizar aumenta significativamente a medida que se incrementa el tamaño de la finca. Por lo tanto, la paralelización se vuelve más efectiva en la reducción de tiempos a medida que el problema se vuelve más complejo.

En resumen, la paralelización es beneficiosa para tamaños de finca más pequeños, con mejoras significativas en los tiempos de ejecución a partir de 6 tablones en adelante. Esto es consistente con la complejidad factorial de la función y demuestra los beneficios de utilizar la paralelización para reducir los tiempos en problemas más complejos.

Tabla IV: tiempos registrados para la función $programacionRiegoOptimo$

Tamaño de finca	Tiempos registrados (ms)			Promedios		
	Secuencial (S)	Paralela (P)	Aceleración (A)	$\overline{S}$	$\overline{P}$	$\overline{A}$
2	0.0195	0.2656	0.0735	0.0275	0.1653	0.1184
	0.0038	0.0700	0.0542			
	0.0036	0.0533	0.0674			
	0.0230	0.2557	0.0898			
	0.0089	0.0795	0.1122			
	0.0052	0.0705	0.0743			
	0.0424	0.2749	0.1544			
	0.0033	0.0536	0.0608			
	0.1382	0.3643	0.3793			
4	0.1277	0.4270	0.2991	0.1453	0.3218	0.4531
	0.0552	0.1260	0.4382			
	0.0557	0.1040	0.5354			
	0.1191	0.4526	0.2631			
	0.0732	0.1426	0.5135			
	0.0554	0.1178	0.4703			
	0.5152	0.7244	0.7112			

	0.0541	0.1122	0.4820			
	0.2520	0.6901	0.3651			
6	10.8203	2.5397	4.2605	3.9701	2.0458	1.9921
	3.1390	1.6364	1.9183			
	2.9804	1.4756	2.0198			
	2.8101	2.2109	1.2710			
	4.3802	1.6696	2.6235			
	3.0041	1.5926	1.8863			
	3.3081	2.9598	1.1177			
	3.0830	1.5027	2.0516			
	2.2052	2.8250	0.7806			
8	250.5599	115.2722	2.1736	250.5936	101.9118	2.4668
	267.8139	105.3703	2.5416			
	256.6730	95.5568	2.6861			
	243.8320	96.1583	2.5357			
	271.5912	102.0666	2.6609			
	272.9548	101.3701	2.6927			
	259.5695	104.0668	2.4943			
	252.0714	94.8877	2.6565			
	180.2769	102.4577	1.7595			
10	34862.4711	11512.5386	3.0282	33658.8926	11311.5969	2.9715
	35655.3857	11607.3463	3.0718			
	34250.3561	11089.5087	3.0885			
	32995.9209	11432.6624	2.8861			
	35561.2489	11577.5705	3.0716			
	35847.9154	11709.1855	3.0615			
	34835.0236	11074.0548	3.1456			
	33567.1873	11163.8915	3.0068			
	25354.5247	10637.6139	2.3835			

Los resultados presentados en la tabla IV confirman una vez más que la paralelización resulta beneficiosa a partir de fincas con 6 tablones, tal como se observó en la tabla III. Sin embargo, en este caso parece ser aún más beneficiosa, ya que las aceleraciones promedio a partir de 6 tablones son superiores a 2 e incluso alcanzan valores de 3 en la mayoría de las pruebas con fincas de tamaño 10.

Es importante destacar que, como era de esperar, esta función paralela es superior a la versión secuencial simplemente por el hecho de hacer uso de las versiones paralelas de las funciones mencionadas anteriormente.

En general, se concluye que todas las versiones paralelas son más rápidas que sus contrapartes secuenciales en los siguientes casos:

- Cuando el tamaño de la finca es lo suficientemente grande como para que el costo computacional supere el costo de la gestión de la concurrencia.
- Cuando la complejidad de la función aumenta rápidamente, lo que implica un mayor número de operaciones a realizar.

Estos hallazgos confirman la eficacia de la paralelización en situaciones donde el tamaño y la complejidad del problema justifican el uso de técnicas concurrentes para mejorar los tiempos de ejecución.