Informe: Proyecto Final PFC

Juan Diego Escobar Triviño

`

2359393

Trabajo presentado a:
Carlos Andrés Delgado S.

Universidad del Valle
Fundamentos de Programación Funcional y Concurrente
Ingeniería de Sistemas
Tuluá - Valle del Cauca
2024

1. Informe de procesos

Esto es realizado tomando en cuenta lo explicado en el taller, por lo que se usarán los distintos Types otorgados por el documento los cuales engloban:

type Tablon = (Int , Int , Int)

type Finca = Vector [Tablon]

type Distancia = Vector [Vector [Int]]

type ProgRiego = Vector [Int]

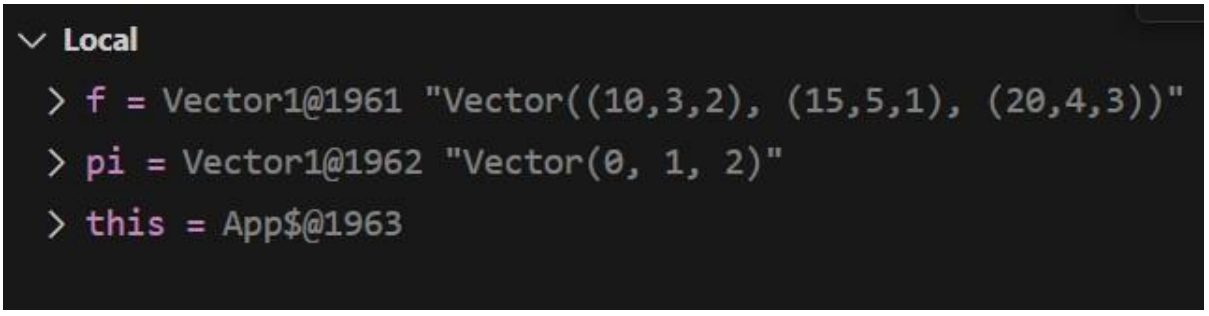
type TiempoInicioRiego = Vector [Int]

Además de en el caso de tablón también se usarán las funciones para buscar dentro del type, las cuales son tsup, treg y prio.

1.1 Calculando el tiempo de inicio de riego

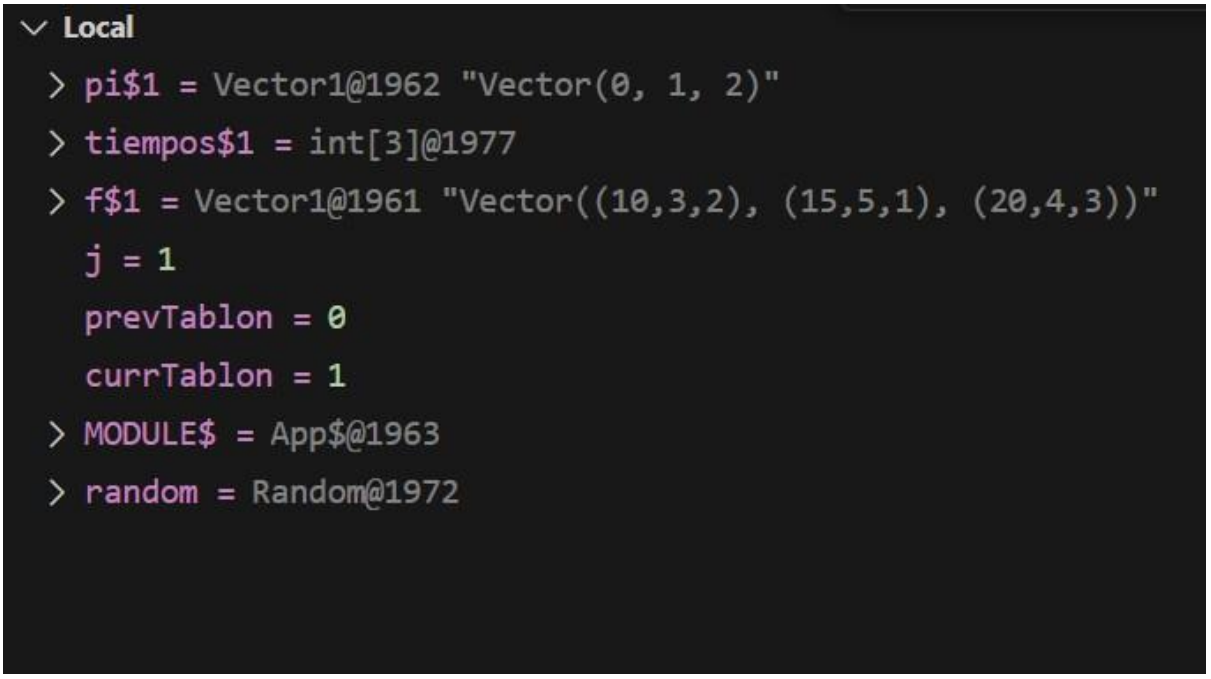
Se implementó la Función “TiR” que recibe una “f” de “n” tamaño la cual representa una Finca y un “pi” el cual es una programación de riego, la cual es generada por una función que será vista más adelante, en este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:

Al ser de tamaño 3 el “pi” que se le asignará a la función va a ser: [0, 1, 2]

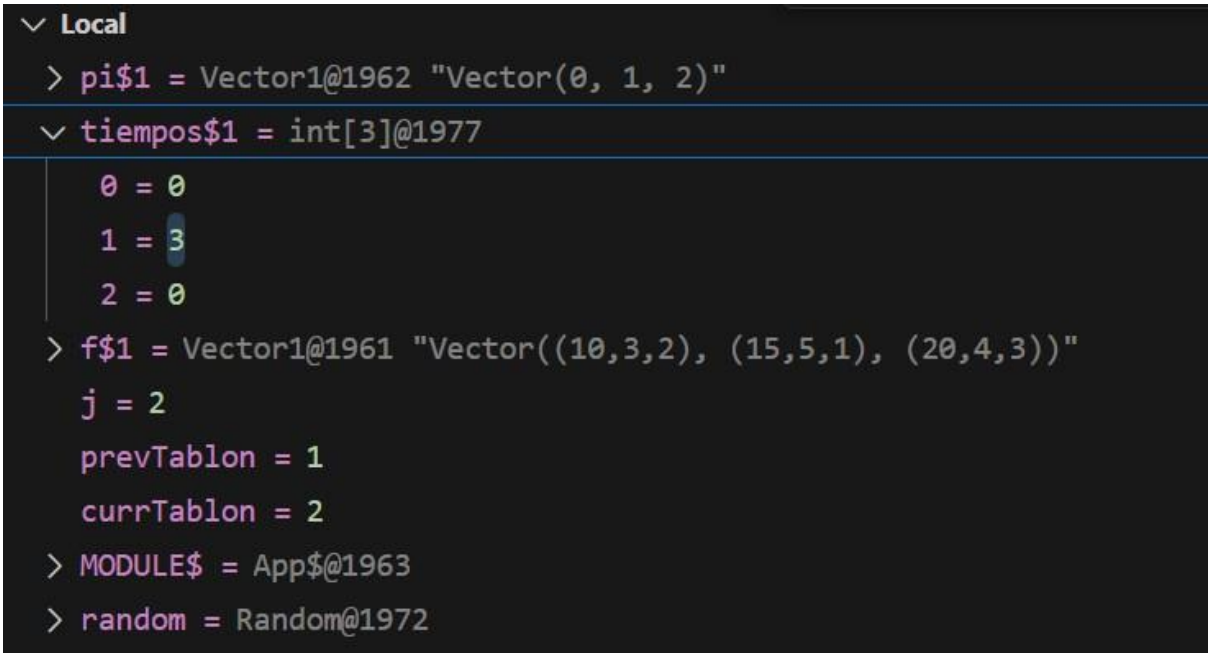


Luego de que se tengan asignados los valores que necesita la función se pasa a la siguiente parte en la que se crea un Array denominado “tiempos”, este Array se le asigna de tamaño el “length” de f y se llena con 0s antes de usarlo como un placeholder. Esto antes de crear otro bucle for el cual esta vez es usado para llenar tiempos en sus respectivos placeholders, se empieza desde el elemento [1] del array ya que el inicio del primer riego siempre será 0.

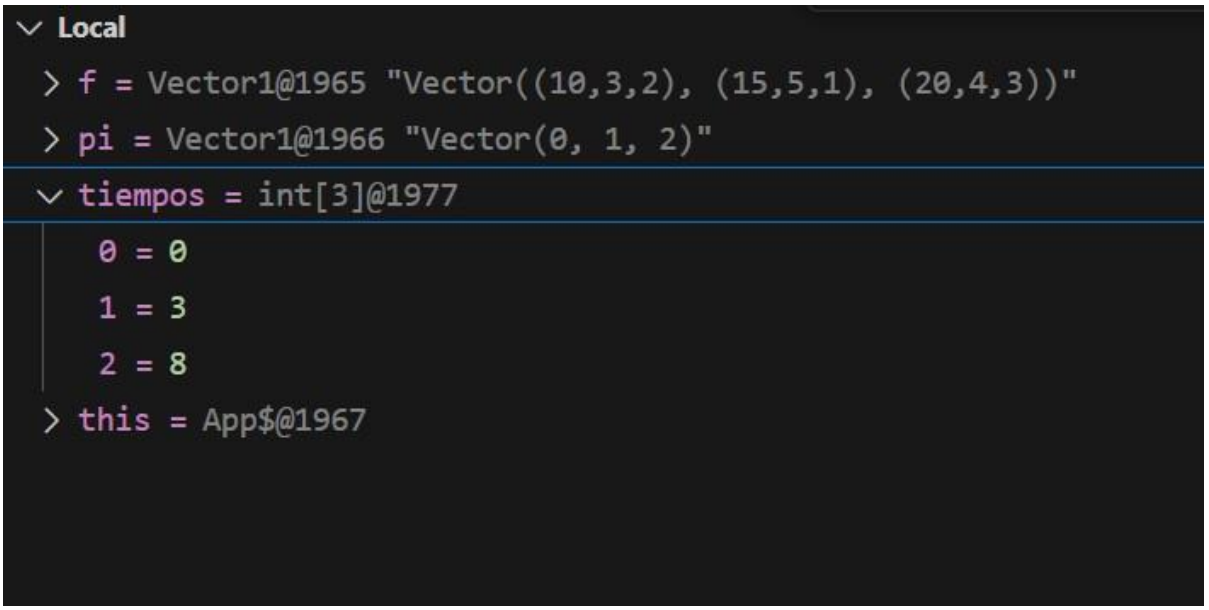
También se crea un par de variables durante el bucle la cual posee de valor el que tiene pi en la posición [j - 1] o [j], dependiendo de si es “prevTablon” o “currTablon”. j es el valor del bucle que va subiendo y las variables representan el tablón previo y el tablón actual respectivamente.



Posteriormente en la posición en la que esté tiempos se realiza una ópera en la cual sumamos el valor del tiempos[prevTablon] con el valor del tiempo de riego de la f en el tablón previo, como el primer caso es 0 solo se asignó a Tiempos[1] un valor de 0 + 3.

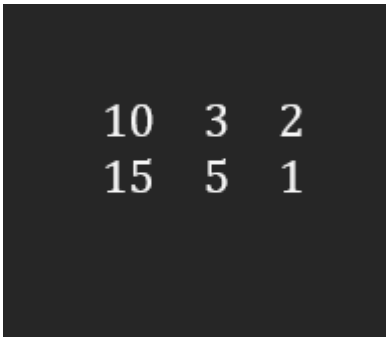


Ya en la posición tiempos[2] si se realiza la suma con un valor de más de 0 por lo que se puede ver como el 3 del tiempo[1] se ha sumado con el tiempo de riego del previo tablón el cual era 5, resultando en 8. Una vez que el bucle cierra lo que sigue es convertir el array en un vector mediante .toVector ya que TiempoDeInicioRiego es un vector.

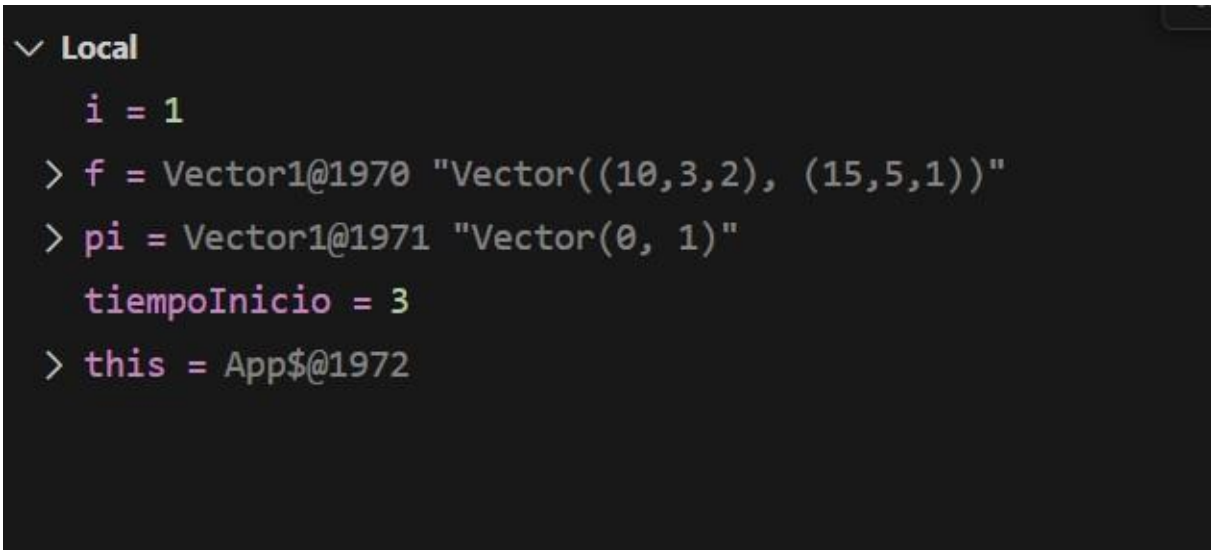


1.1.2 costoRiegoTablon

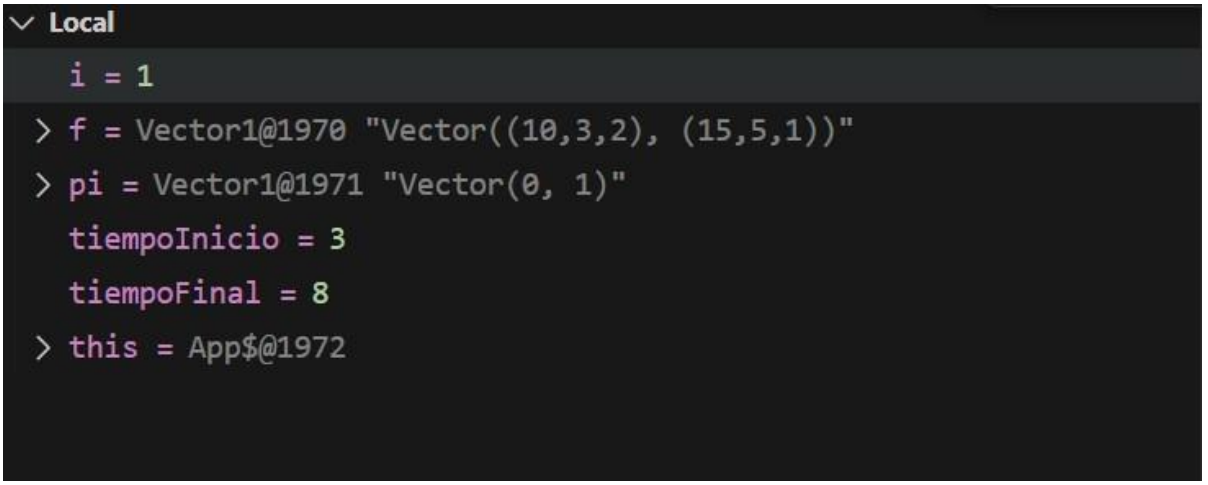
Se implementó la Función “costoRiegoTablon” que recibe un entero “i” el cual representa el número de un tablón al cual vamos a calcular el costo, una finca f de “n” tamaño, un pi que es la programación de riego, en este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:



El pi sería entonces de [0, 1, 2] y el i sería de 1 para poder observar el costo de riego de este tablón luego del anterior, lo primero que se hace es crear la variable TiempoInicio para la cual usaremos el TiR que se vió anteriormente para poder calcular el tiempo total que se demora en regar hasta el tablon, por lo que luego de calcular el TiR se le suma el valor del tiempo de riego del tablón a calcular.

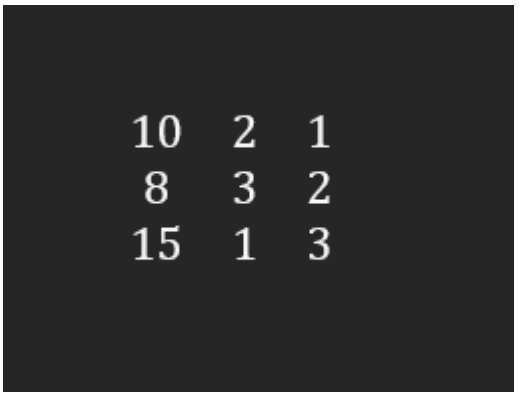


Se calcula la suma resultando en un 8 como tiempoFinal para ser usado posteriormente, ya que primero se debe revisar un condicional el cual es (Ts del tablon - Tr del tablon) >= TiempoInicio el cual en este caso se cumple por lo que el resultado que retorna la función será Ts - TiempoFinal

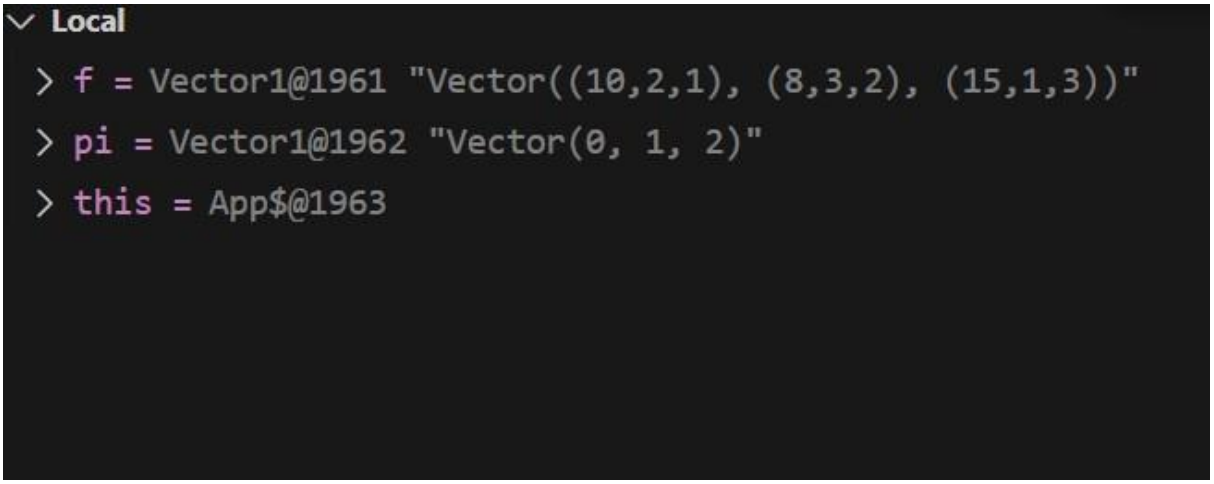


1.1.3 costoRiegoFinca

Se implementó la Función "costoRiegoFinca" que recibe una finca de "n" tamaño y una programación de riego pi, y esta función sirve mediante una sumatoria o sea un bucle "for". En este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:

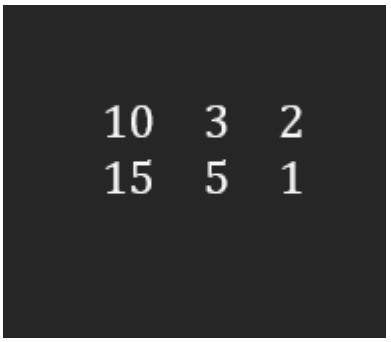


Cuya pi será tal como en varios anteriores [0, 1, 2] y luego de este punto lo único que se puede explicar de esta función es que realiza un bucle "for" que va probando con cada tablon llamando la función previa costoRiegoTablon, usando map para poder al final realizar una suma de cada uno de los resultados retornando un Int.

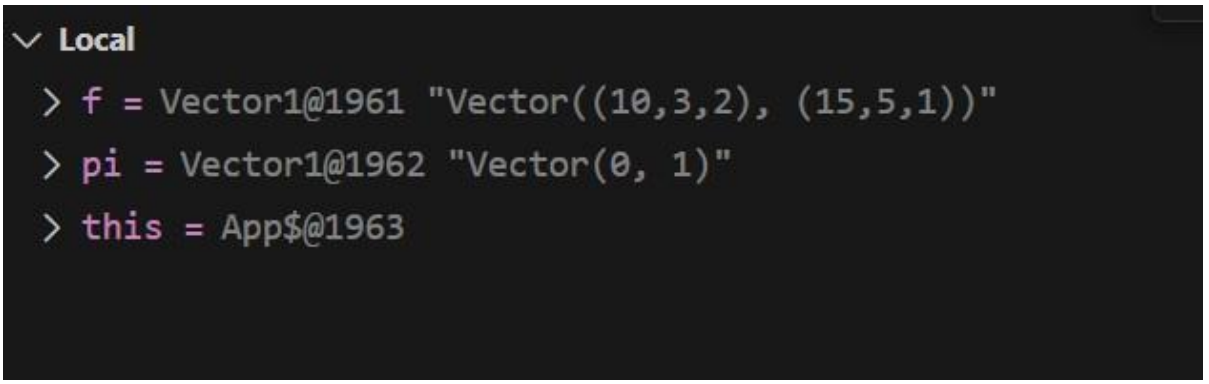


1.1.4 costoRiegoFincaPar

Se implementó la Función "costoRiegoFincaPar" que recibe una finca de "n" tamaño y una programación de riego pi, y esta función sirve mediante una sumatoria o sea un bucle "for". Solo que vamos a usar la paralelización mediante par.map, en este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:

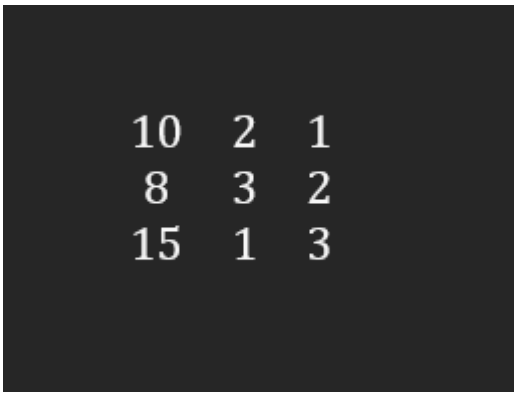


Cuya pi será [0, 1] y tal como en el caso anterior utiliza un bucle for para ir probando en cada tablón, solo que ahora lo que se usa par.map lo que permite a cada uno de los elementos de la finca que se le aplique la función anterior, por lo que cada una se hace por separado durante el bucle for.

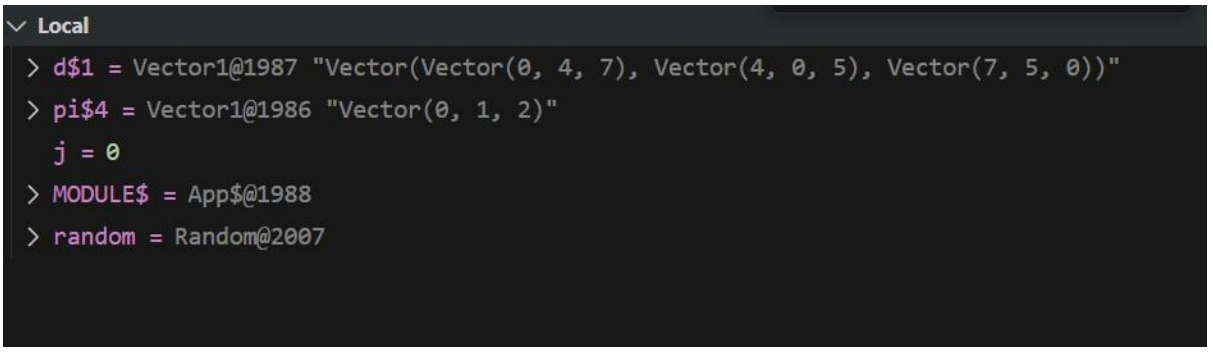


1.1.5 costoMovilidad

Se implementó la Función "costoMovilidad" que recibe una finca de "n" tamaño, una programación de riego pi y una "d" distancia, esta función sirve mediante una sumatoria o sea un bucle "for". En este caso vamos a usar una finca y una distancia que corresponde a vectores de la siguiente manera:



Una vez que se ha llamado la función está al igual que en el caso anterior utiliza un bucle for en el cual suma cada uno de los valores de la distancia, usando el pi para revisar en cada una de las posiciones de la distancia. Para al final retorna un Int que corresponde a lo siguiente:



1.1.6 costoMovilidadPar

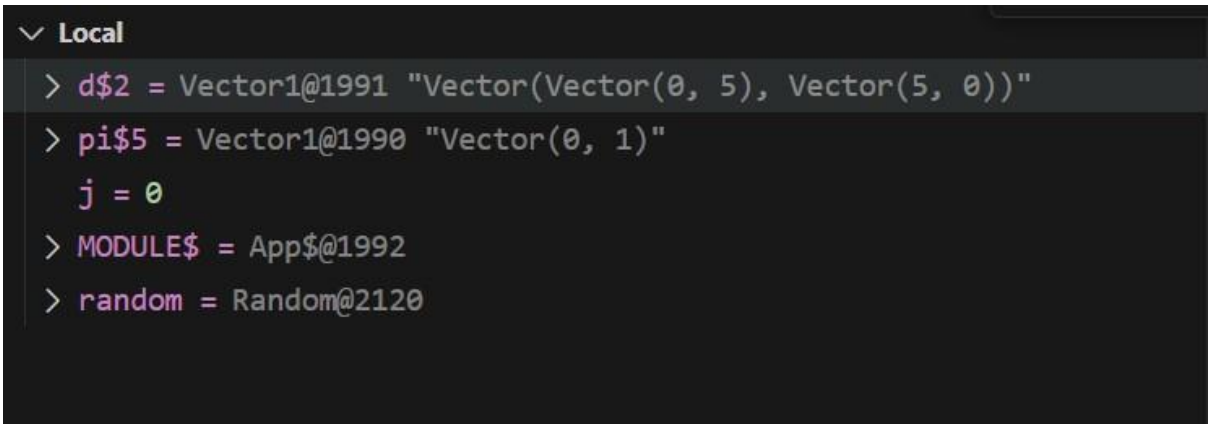
Se implementó la Función "costoMovilidadPar" los mismos valores que costoMovilidad los cuales son f, pi y d, esta función sirve mediante una sumatoria

o sea un bucle “for”. En este caso vamos a usar una finca y una distancia que corresponde a vectores de la siguiente manera:

```
10  3  2
15  5  1
```

```
0  5
5  0
```

Tal como en el caso anterior lo que se realizará ahora será un bucle for usando map para afectar a cada uno de los valores en la distancia con una suma, solo que ahora que estamos usando “.par” esto se realizará en paralelo, haciendo que cada función se resuelva independiente de las otras.

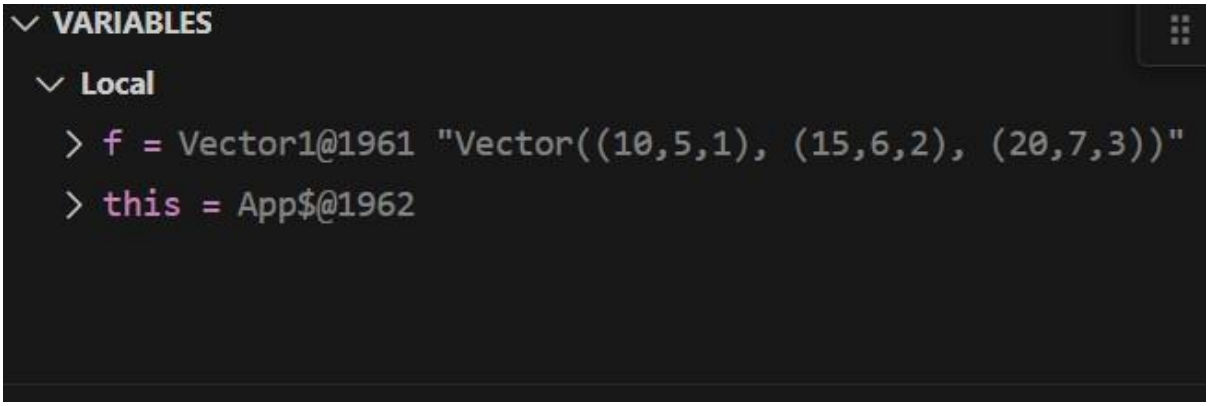


1.2 Generando programaciones de riego

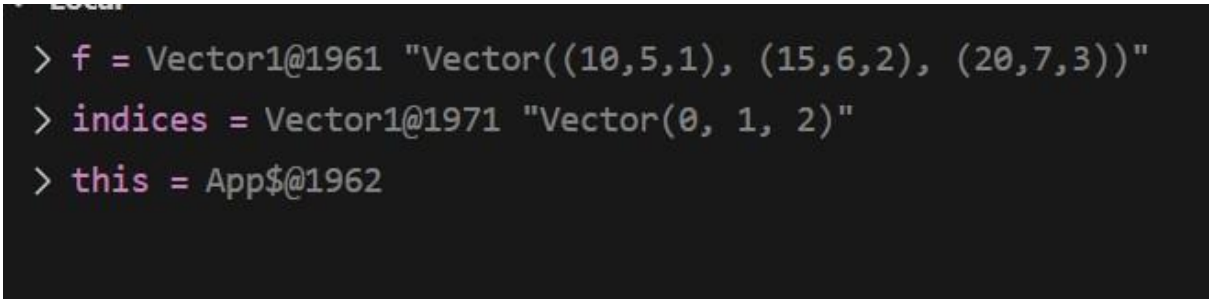
Se implementó la Función “generarProgramacionesRiego” que recibe una finca de “n” tamaño la cual es lo único que necesita la función al ser usada, en este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:

```
10  5  1
15  6  2
20  7  3
```

Una vez tenemos eso establecido para probar la función, lo siguiente que hace la función es recibir los valores y asignarlo a f, esto justo antes de usar la función ".length" para medir el límite de un bucle "for".



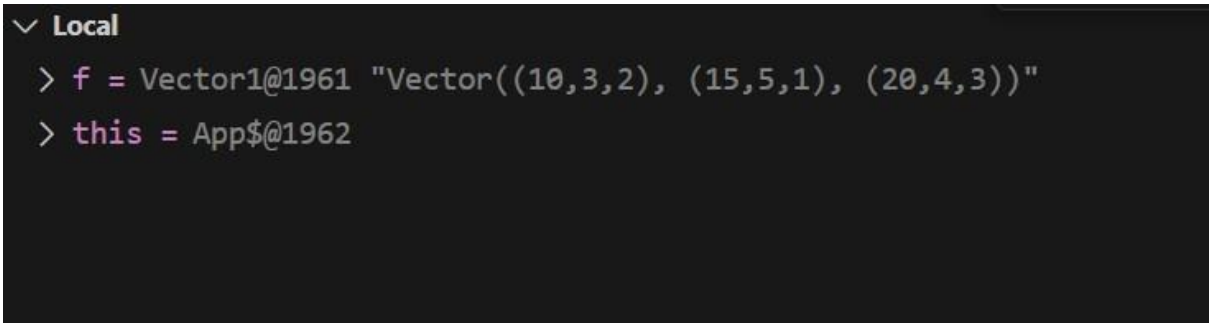
Posteriormente se crea la variable "indices" el cual recibe la cuenta del length como un vector el cual va a retornar una vez termine el ciclo, lo único que debe hacer luego de recibir los valores del ciclo es permutarlos para que retorne ese Vector[ProgRiego] la cual será un Vector de [0, 1, 2] al ser solo un tablero de tamaño 3.



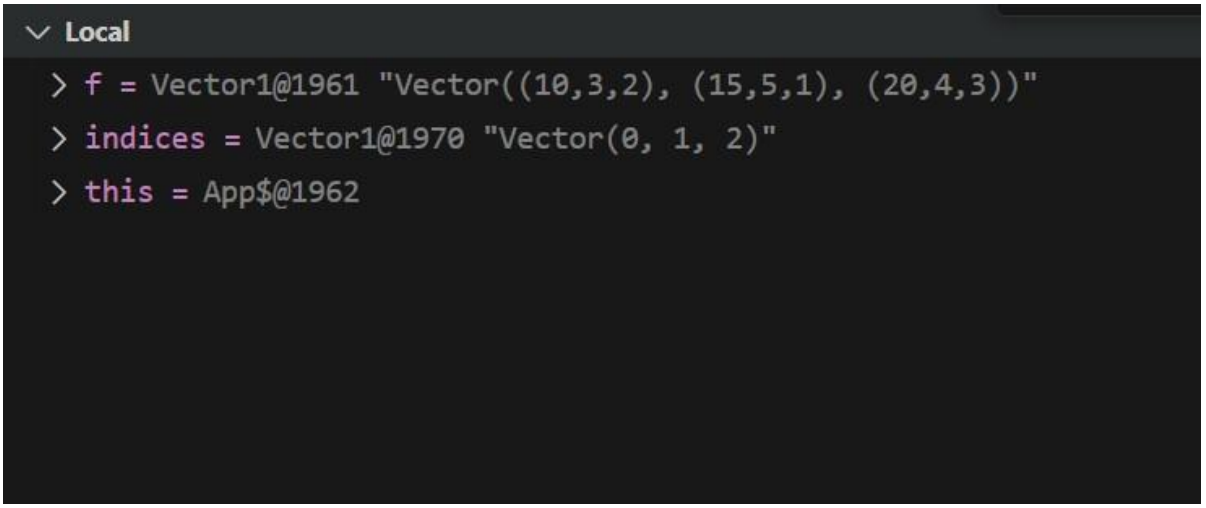
1.3 Generando programaciones de riego Paralelizada

Se implementó la Función "generarProgramacionesRiegoPar" que recibe una finca de "n" tamaño que será lo único que se usará en esta función, en este caso vamos a usar una finca que corresponde a un vector de la siguiente manera:

Lo primero que hace la función tal como antes es recoger los datos presentados para la realización de la operación necesaria, por ahora no se realiza paralelización.

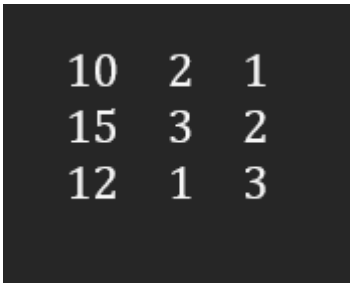


Lo que sigue es lo mismo que anteriormente que es usar un bucle para poder crear índices en forma de vectores, los cuales posteriormente gracias a la paralelización son permutados en un solo vector, algo que para nada parece muy efectivo con unas operaciones tan simples y tan pequeñas.



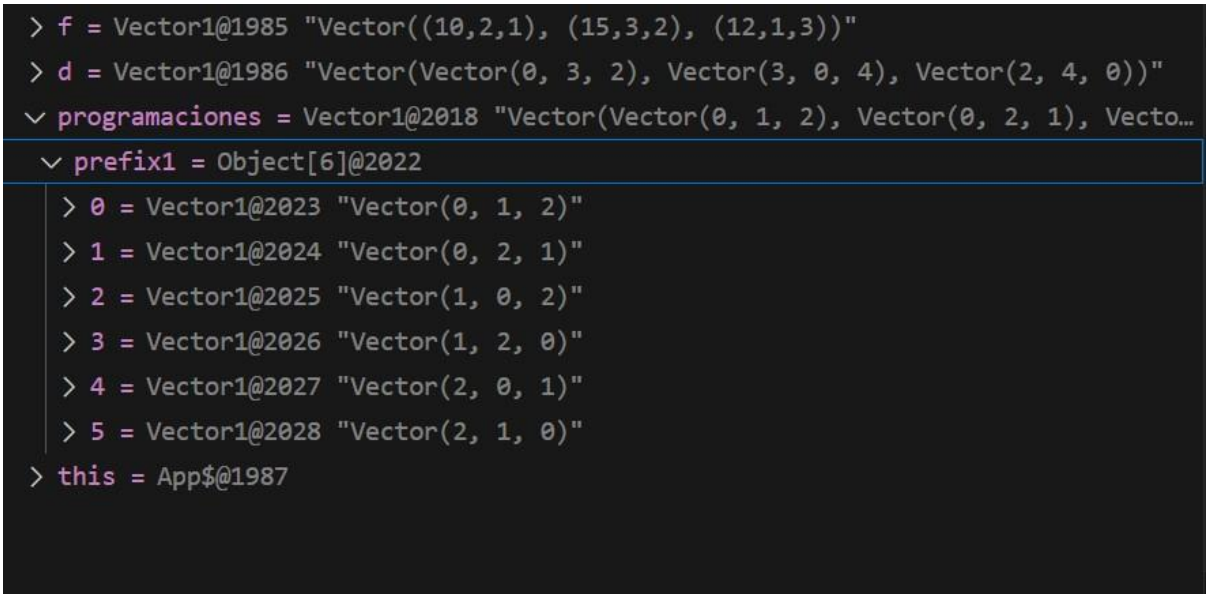
1.3.1 programaciónRiegoOptimo

Se implementó la Función “programaciónRiegoOptimo” que recibe una finca de “n” y una “d” distancia, en este caso vamos a usar una finca y distancia que corresponde a un vector de la siguiente manera:

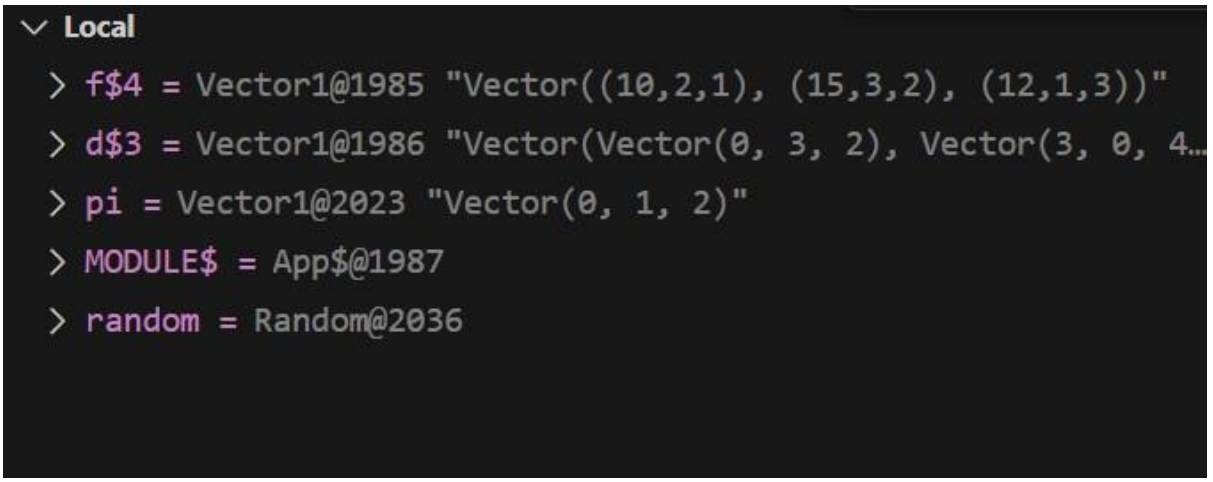


0	3	2
3	0	4
2	4	0

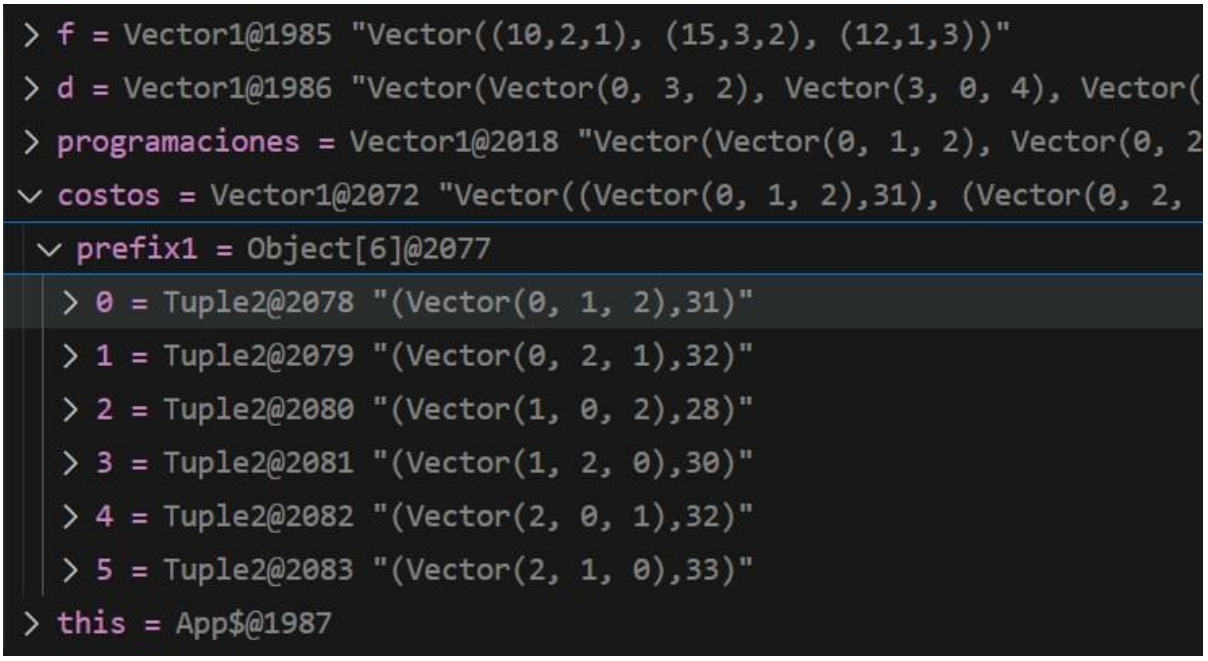
El programa usa entonces la “f” otorgada para poder crear la variante “programaciones” que reúne cada una de las posibles programaciones de riego, en este caso se crearon unas 6 para ser usadas posteriormente.



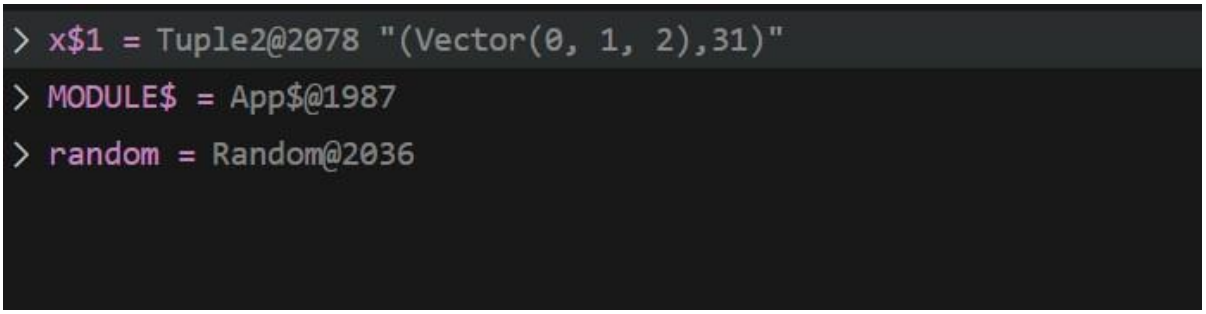
Lo que sigue es la realización de un map en el que a cada uno de los elementos de programaciones se le suman los costos de riego de la finca con los de movilidad, usando pi que es tomado de programaciones para la operación.



Todo esto para crearnos una nueva variable llamada “costos” que nos retorna una serie de vectores compuestos del pi al que realizamos la operación anterior y su resultado, esto para realizar una última operación que nos retornará el valor final de este riego óptimo.

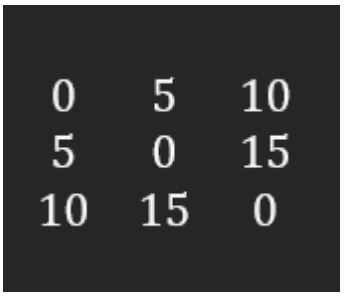


Lo último que se realiza es esta operación de aplicar un mínimo a los vectores, solo que esta escrita de la siguiente manera costos.minBy(_._2) lo que permite que lo retornado sean los dos pares menores, o sea el más óptimo.

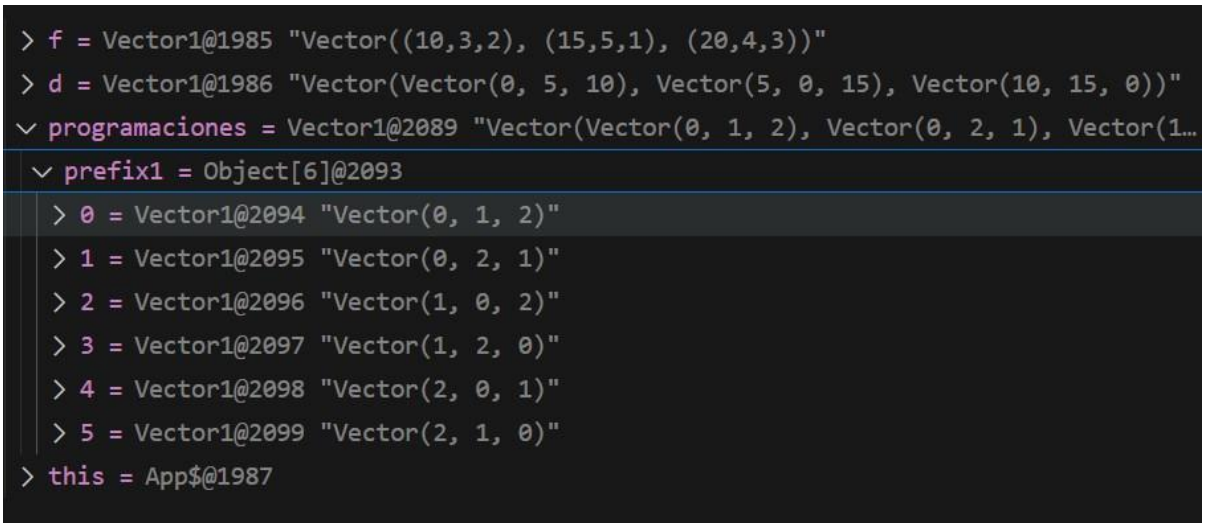


1.3.2 programaciónRiegoOptimoPar

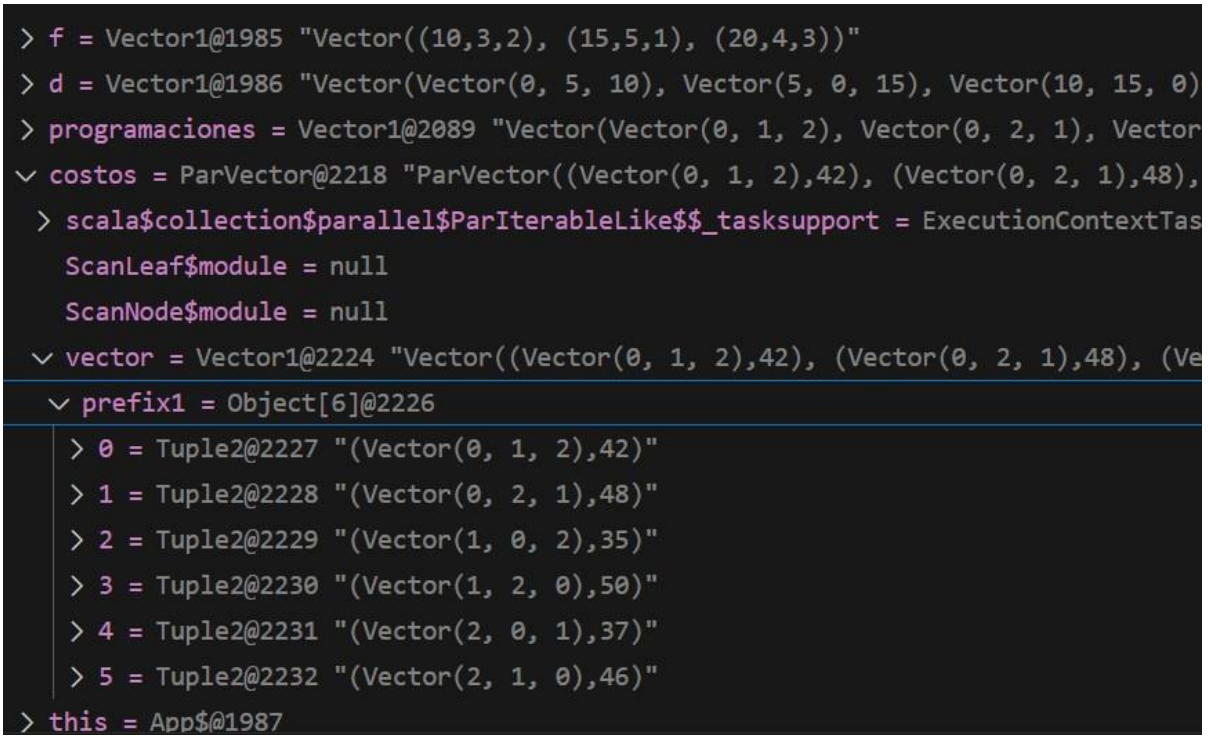
Se implementó la Función “programaciónRiegoOptimoPar” que recibe una finca de “n” y una “d” distancia, en este caso vamos a usar una finca y distancia que corresponde a un vector de la siguiente manera:



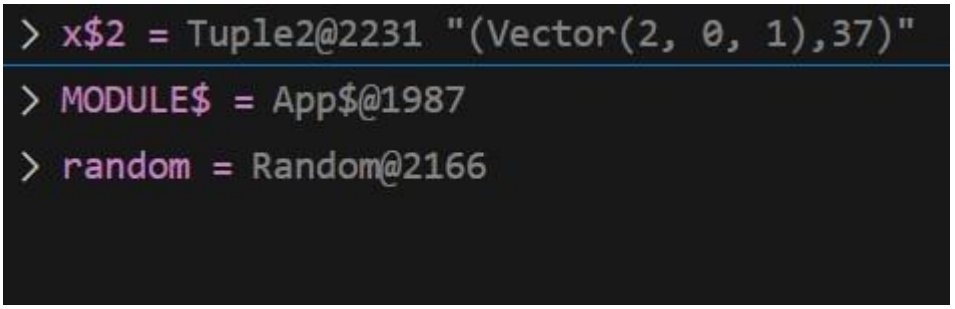
Se procede a crear programaciones usando el f otorgado al llamar la función, creando 6 que serán usadas más adelante, hasta ahora no se ha usado la paralelización.



Tal como en el caso anterior se vuelve a crear costos el cual reunió las sumas del costo de movilidad y costo de riego luego de que dicha operación se aplicará a cada un dirección de programaciones, solo que en este caso se usó la paralelización al hacer el map. Lo que nos permitió calcular cada una de las operaciones de manera casi simultánea a pesar de que no se pudo registrar en el screenshot.



Finalmente sucede tal como con la función anterior y solo se vuelven a llamar los costos para poder hallar el par menor, cosa para la que no sé usó la paralelización y al final nos retorna el valor adecuado.



2. Informe de paralización

2.1 generarProgramacionesRiegoPar

```
def generarProgramacionesRiegoPar(f:Finca) : Vector[ProgRiego] = {
    val indices = (0 until f.length).toVector
    indices.permutations.toVector.par.toVector
}
```

Ley de Amdahl

$S = 1/(1 - P) + P/N$

$P = 0,95$ *//Puesto que la paralelización está aplicada para que se aplique a todas las permutaciones*

$N = 8$ *//Cantidad estimada de procesadores*

$S = 1/(1 - 0,95) + 0,95/8$

$S = 6,71$

```
Iteracion: {1}List(0.0432, 0.0748, 0.5775401069518716)

Iteracion: {2}List(0.0611, 0.0374, 1.6336898395721924)

Iteracion: {3}List(0.032, 0.0399, 0.8020050125313284)

Iteracion: {4}List(0.0296, 0.0307, 0.9641693811074918)

Iteracion: {5}List(0.0471, 0.2209, 0.2132186509732911)

Iteracion: {6}List(0.2404, 0.2297, 1.046582498911624)

Iteracion: {7}List(1.7098, 0.885, 1.9319774011299435)

Iteracion: {8}List(5.5799, 7.0684, 0.7894148605059137)
```

2.2 costoRiegoFincaPar

```
def costoRiegoFincaPar(f:Finca, pi:ProgRiego) : Int = {
    (0 until f.length).par.map(i => costoRiegoTablon(i, f, pi)).sum
}
```

Ley de Amdahl

$S = 1/(1 - P) + P/N$

$P = 0,85$ *//Puesto que la paralelización está aplicada para que se aplique a todas los valore del map, menos a la suma final*

$N = 8$ *//Cantidad estimada de procesadores*

$S = 1/(1 - 0,85) + 0,85/8$

$S = 3,9$

```
Iteracion: {1}List(0.0078, 0.1167, 0.066838046272493 57)

Iteracion: {2}List(0.0052, 0.3252, 0.015990159901599015)

Iteracion: {3}List(0.0163, 0.3461, 0.0470962149667726 04)

Iteracion: {4}List(0.0131, 0.3647, 0.035919934192486 97)

Iteracion: {5}List(0.0084, 0.4372, 0.01921317474839 89)

Iteracion: {6}List(0.0194, 0.2671, 0.072631973043803 82)

Iteracion: {7}List(0.0251, 0.2527, 0.09932726553225 17)

Iteracion: {8}List(0.0143, 0.293, 0.048805460750853 24)
```

2.3 costoMovilidadPar

```
def costoMovilidadPar(f:Finca,pi:ProgRiego, d:Distancia) : Int = {
    (0 until pi.length - 1).par.map(j => d(pi(j))(pi(j+1))).sum
}
```

Ley de Amdahl

$S = 1/(1 - P) + P/N$

$P = 0,85$ *//Puesto que la paralelización está aplicada para que se aplique a todas los valore del map, menos a la suma final*

$N = 8$ *//Cantidad estimada de procesadores*

$S = 1/(1 - 0,85) + 0,85/8$

$S = 3,9$

```
Iteracion: {1}List(0.0044, 0.0891, 0.049382716049382716)

Iteracion: {2}List(0.005, 0.0742, 0.06738544474393533)

Iteracion: {3}List(0.0042, 0.1301, 0.032282859338970002)

Iteracion: {4}List(0.0049, 0.1131, 0.043324491600353666)

Iteracion: {5}List(0.0063, 0.1993, 0.031610637230306070)

Iteracion: {6}List(0.0054, 0.2161, 0.024988431281813977)

Iteracion: {7}List(0.0046, 0.3244, 0.014180024660912453)

Iteracion: {8}List(0.0037, 0.3339, 0.011081162024558252)
```

2.4 ProgramacionRiegoOptimoPar

```
def ProgramacionRiegoOptimoPar(f:Finca, d:Distancia) : (ProgRiego, Int) = {
    val programaciones = generarProgramacionesRiegoPar(f)
    val costos = programaciones.par.map(pi =>
        (pi, costoRiegoFincaPar(f, pi) + costoMovilidadPar(f, pi, d))
    )
    costos.minBy(_._2)
}
```

Ley de Amdahl

$S = 1/(1 - P) + P/N$

$P = 0,85$ *//Puesto que la paralelización está aplicada para que se aplique a todas los valore del map, menos al minBy final*

$N = 8$ *//Cantidad estimada de procesadores*

$S = 1/(1 - 0,85) + 0,85/8$

$S = 3,9$

```
Iteracion: {1}List(0.0183, 0.0975, 0.187692307692307 69)

Iteracion: {2}List(0.0227, 0.242, 0.0938016528925 62)

Iteracion: {3}List(0.0486, 0.484, 0.100413223140495 86)

Iteracion: {4}List(0.0889, 0.9379, 0.094786224544194 48)

Iteracion: {5}List(0.4991, 3.8967, 0.128082736674622 1)

Iteracion: {6}List(1.5958, 5.9659, 0.267486883789537 2)

Iteracion: {7}List(4.4547, 17.3055, 0.257415272601196 2)

Iteracion: {8}List(45.9976, 134.3175, 0.342454259497087 13)
```

3. Informe de corrección

Antes de revisar las funciones principales, miramos los generadores de entradas aleatorias, para la finca y para la matriz de distancia

3.1 fincaAlAzar

$F = \langle T_0, \dots, T_{n-1} \rangle$

$T = \langle ts^F_i, tr^F_i, p^F_i \rangle$

```
def FincaALAzar(long: Int): Finca = {

    val vec = Vector.fill(long)(

        (random.nextInt(long*2)+1, random.nextInt(long*2)+1, random.nextInt(4)+1)
    )
    vec
}
```

$v = Vector.fill(long)$

(

($random.nextInt(long * 2) + 1$, // ts^F_i

$random.nextInt(long) + 1$, // tr^F_i

$random.nextInt(4) + 1$) // p^F_i

)

$long = cantidad\ de\ Tablones\ en\ la\ finca$

3.2 distanciaAlAzar

$Matriz\ (long\ x\ long),\ diagonal\ de\ 0\ e\ igual\ a\ su\ transpuesta$

```scala
def DistanciaAlAzar(long: Int): Distancia = {

    val vec = Vector.fill(long, long)(

        random.nextInt(long*3) + 1
    )
    Vector.tabulate(long, long)((i, j) =>
        if(i < j) vec(j)(i)
        else if(i == j) 0
        else vec(i)(j))
}
```

*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))

else if (i == j) 0 // Diagonal de 0
Vector.tabulate(long , long) // long x long

3.3 tIR (Generador de tiempos de riego)

Dada una programación de riego Π, se puede calcular, para cada tablón T_i, el tiempo en que se iniciará su riego t_i^{Π} según:

$$t^{\Pi}_{\pi_0} = 0,$$
$$t^{\Pi}_{\pi_j} = t^{\Pi}_{\pi_{j-1}} + tr^{F}_{\pi_{j-1}}, \quad j = 1, \ldots, n-1.$$

```scala
def tIR(f: Finca, pi:ProgRiego) : TiempoInicioRiego = {
    val tiempos = Array.fill(f.length)(0)
    for (j <- 1 until pi.length) {
        val prevTablon = pi(j-1)
        val currTablon = pi(j)
        tiempos(currTablon) = tiempos(prevTablon) + treg(f, prevTablon)
    }
    tiempos.toVector
}
```

$val\ tiempos\ =\ Array.fill(\ f.length)(0)$
$for\ (j\ <-\ 1\ until\ pi.length\ )\ \{$
$val\ prevTablon\ =\ pi\ (\ j\ -\ 1\ )$
$val\ currTablon\ =\ pi\ (\ j\ )$
$tiempos(currTablon)\ =\ tiempos(\ prevTablon\ )\ +\ treg(\ f,\ prevTablon)\ \}$

$val\ tiempos\ =\ Array.fill(\ f.length)(0)\ //Se\ crea\ un\ vector\ de\ 0\ de\ longitud\ f\ (finca.length)$
$//\ Para\ cada\ tablón\ "j",\ se\ calcula\ el\ tiempo\ de\ inicio\ de\ riego\ con\ el\ valor\ de\ riego\ del\ tablón$
$y\ el\ tiempo\ de\ riego\ acumulado$

3.4 generarProgramacionesRiego

$f:\ FINCA\ ->\ Vector(\ Vector(\ Int\ ))$

```
def generarProgramacionesRiego(f:Finca) : Vector[ProgRiego] = {
    val indices = (0 until f.length).toVector
    indices.permutations.toVector
}
```

P_f :
$f\ =\ Finca$
$val\ indices\ =\ (0\ until\ f.length).toVector\quad //\ Se\ crea\ el\ vector\ con\ los\ índices\ de\ los\ tablones$
$indices.permutations.toVector\qquad //\ Se\ crean\ todas\ las\ permutaciones\ de\ los\ índices$

Caso Base:

$Finca\ =\ (T_1,\ T_2)$
$Finca.length\ =\ 2$
$indices\ =\ Vector(0,\ 1)$
$indices.permutations.toVector\ =\ Vector(\ Vector(0,1),\ Vector(1,0))$

Caso Inductivo:

$Finca\ =\ (T_1,\ ...,\ T_n)$
$Finca.length\ =\ n$
$indices\ =\ Vector(0,\ 1,\ 2,\ ...,\ n\ -\ 1)$
$indices.permutations.toVector\ =\ Vector(\ Vector_1(0,\ 1,\ ...,\ n\ -\ 1),\ Vector_2(1,\ 2,\ ...,\ n\ -\ 1,\ 0),$
$.....,\ Vector_{n!}\ (n\ -\ 1,\ ...,\ 0,\ 1,\ 2)$

3.5 costoRiegoTablon

f:

El costo de riego de un tablón T_i de la finca F, dada una programación de riego Π, se define como:

$$CR_F^{\Pi}[i] = \begin{cases} ts_i^F - (t_i^{\Pi} + tr_i^F), & \text{si } ts_i^F - tr_i^F \geq t_i^{\Pi}, \\ p_i^F \cdot ((t_i^{\Pi} + tr_i^F) - ts_i^F), & \text{de lo contrario.} \end{cases}$$

```
def costoRiegoTablon(i:Int, f:Finca, pi:ProgRiego) : Int = {
    val tiempoInicio = tIR(f, pi)(i)
    val tiempoFinal = tiempoInicio + treg(f, i)
    if (tsup(f,i) - treg(f, i) >= tiempoInicio) {
        tsup(f,i) - tiempoFinal
    } else  {
        prio(f,i) * (tiempoFinal - tsup(f,i))
    }
}
```

P_f:

$val\ tiempoInicio\ =\ tIR(f\ ,\ pi)(i)$
$val\ tiempoFinal\ =\ tiempoInicio\ +\ treg\ (\ f\ ,\ i\ )$

$if\ (tsup(f\ ,\ i)\ -\ treg(f\ ,\ i)\ >=\ tiempoInicio)\ \{$
$\quad tsup(f\ ,\ i)\ -\ tiempoFinal$
$\}\ else\ \{$
$\quad tprio(f\ ,\ i)\ *\ (tiempoFinal\ -\ tsup(f\ ,\ i))\ \}$

$tiempoInicio\ =\ generar\ Tiempos\ de\ Inicio\ de\ Riego\ para\ cada\ tablón$
$tiempoFinal\ =\ tiempoInicio\ +\ tiempo\ de\ regado\ del\ tablón\ actual$

Caso Base:

$Finca\ (\ T_1\ =\ (3, 2, 4),\ T_2\ =\ (1, 1, 4))$
$tsup_1\ =\ 3,\ \ treg_1\ =\ 2,\ tprio_1\ =\ 4$
$tsup_2\ =\ 1,\ \ treg_2\ =\ 1,\ tprio_2\ =\ 4$
$ProgRiego\ =\ (0,\ 1)$
$tiempoInicio\ =\ (0\ ,\ 2)$

$Costo\,T_1 =$

$tsup(f, i) - treg(f, i) >= tiempoInicio$

$3 - 2 >= 0 \quad //true$

$Costo\,T_1 = tsup(f, i) - tiempoFinal$

$Costo\,T_1 = 3 - (0 + 2) = 1$

Caso inductivo:

$Costo\,T_2 =$

$tsup(f, i) - treg(f, i) >= tiempoInicio$

$1 - 1 >= 2 \quad //false$

$Costo\,T_2 = tprio(f, i) * (tiempoFinal - tsup(f, i))$

$Costo\,T_2 = 4 * ((2 + 1) - 1) = 4 * 2 = 8$

3.6 costoRiegoFinca

$f: costoRiegoFinca = costoT_1 + costoT_2 + \ldots + costoT_n$

```
def costoRiegoFinca(f:Finca, pi:ProgRiego) : Int = {
    (0 until f.length).map(i => costoRiegoTablon(i, f, pi)).sum
}
```

$P_f:$

$(0\ until\ f.length).map(i => costoRiegoTablon(i, f, pi)).sum$

$//\ Aplica\ a\ cada\ tablón\ un\ costo\ de\ tablon\ y\ luego\ los\ suma$

Caso base:

$Finca\,(T_1 = (3, 2, 4), T_2 = (1, 1, 4))$

$tsup_1 = 3, \quad treg_1 = 2, tprio_1 = 4$

$tsup_2 = 1, \quad treg_2 = 1, tprio_2 = 4$

$ProgRiego = (0, 1)$

$tiempoInicio = (0, 2)$

$Costo\,T_1 = 1$

$Costo\,T_2 = 8$

$CostoRiegoFinca = CT_1 + CT_2 = 1 + 8 = 9$

Caso inductivo:

$Finca\ (T_1, T_2, \ldots, T_n)$

$ProgRiego = (0, 1, 2, \ldots, n-1)$

$tiempoInicio = (tIT_1, tIT_2, \ldots, tIT_n)$

$CostoRiegoFinca = CT_1 + CT_2 + \ldots + CT_n$

3.7 costoMovilidad

f:

El costo de movilidad del sistema móvil de riego está dado por una matriz de distancias D_F, donde $D_F[i, j]$ representa el costo de mover el sistema móvil de T_i a T_j. Este costo es proporcional a la distancia entre los tablones. Para este ejercicio, se asume que $D_F[i, j] = D_F[j, i]$ y $D_F[i, i] = 0$.

El costo de movilidad para una programación Π se define como:

$$CM_F^{\Pi} = \sum_{j=0}^{n-2} D_F[\pi_j, \pi_{j+1}].$$

```
def costoMovilidad(f:Finca, pi:ProgRiego, d:Distancia) : Int = {
    (0 until pi.length - 1).map(j => d(pi(j))(pi(j+1))).sum
}
```

P_f:

$(0\ until\ pi.length - 1).map(j => d(pi(j))(pi(j+1))).sum$

//Aplica para cada par (T_1, T_2) un valor en la matriz de distancia, y luego sumar todos los valores

Caso Base:

distancia = $\begin{pmatrix} 0 & 3 \\ 3 & 0 \end{pmatrix}$

progRiego = (0, 1)

distancia (0, 1) = 3

Caso inductivo:

distancia = $\begin{pmatrix} 0 & V(1,0) & \ldots & V(n,0) \\ V(0,1) & 0 & \ldots & \ldots \\ \ldots & \ldots & 0 & V(n,n-1) \\ V(0,n) & \ldots & V(n-1,n) & 0 \end{pmatrix}$

progRiego = (0, 1, 2, ..., n)

distancia(0,1) = V(0,1)
distancia(1,2) = V(1,2)
…
distancia(n-1, n) = V(n-1, n)

CostoMovilidad = V(0,1) + V(1,2) + … + V(n-1, n)

3.8 ProgramacionRiegoOptimo

f: $Finca,\ distancia\ \ ->\ \ ProgramacionRiego\ mas\ baja$

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

P_f:

$val\ programaciones\ =\ generarProgramacionesRiego(f)$
$val\ costos\ =\ programaciones.map(pi\ =>$
$(pi\ ,\ costoRiegoFincaPar(f\ ,\ pi)\ +\ costoMovilidadPar(f,\ pi\ ,d))\quad)$

$costos.minBy(\_.\_2)$

Caso Inductivo:
$val\ programaciones\ =\ generarProgramacionesRiegoPar(f)$
$//Genera\ todas\ las\ programaciones$
$val\ costos\ =\ programaciones.map(pi\ =>$
$(pi\ ,\ costoRiegoFincaPar(f\ ,\ pi)\ +\ costoMovilidadPar(f,\ pi\ ,d))\quad)$
$//Aplica\ un\ map\ a\ cada\ programación,\ generando\ el\ costo\ de\ cada\ programación\ en\ un\ nuevo\ vector$
$costos.minBy(\_.\_2)\ //Obtiene\ la\ programación\ con\ el\ costo\ mínimo\ o\ más\ bajo$

4. Conclusiones

4.1 Presentación de Resultados de Benchmarks

4.1.1 Benchmark costoRiegoFinca

Tamaño de la finca (tablones)	Tiempo Secuencial (ms)	Tiempo Paralelizado (ms)	Aceleración (%)
6	0.044301	0.657099	0.0674
7	0.0372	0.2742	0.1356
8	0.0229	0.436401	0.0524
9	0.039501	0.466	0.08476

4.1.2 Benchmark costoMovilidad

Tamaño de la finca (tablones)	Tiempo Secuencial (ms)	Tiempo Paralelizado (ms)	Aceleración (%)
6	0.0068	0.3493	0.0194
7	0.0061	0.2815	0.0216
8	0.008001	0.5025	0.01592
9	0.0096	0.4841	0.01983

4.1.3 Benchmark generarProgramacionRiego

Tamaño de la finca (tablones)	Tiempo Secuencial (ms)	Tiempo Paralelizado (ms)	Aceleración (%)
6	0.816	0.6924	1,17964
7	1,58200	1,616090	0.979209
8	13.950	11.578	1,17416
9	121.34	127.008	0.955386

4.1.4 Benchmark riegoOptimo

Tamaño de la finca (tablones)	Tiempo Secuencial (ms)	Tiempo Paralelizado (ms)	Aceleración (%)
6	3,239	4,5439	0.712801
7	11,04	36,093	0.3060009
8	118,199	240,5370	0.491394
9	966,398	1.973,672	0.48964483

4.2 Análisis y Conclusiones

4.2.1 Tamaños de fincas donde el paralelismo genera ganancias significativas

- **costoRiegoFinca:**

  En general no se muestra una aceleración significativa, en la mayoría de los casos el costo de paralelizar es mayor.

- **costoMovilidad:**

  En general no se muestra una aceleración significativa, en la mayoría de los casos el costo de paralelizar es mayor.

- **generarProgramacionesRiego:**

  La paralelización es más efectiva en tamaños intermedios como 8 tablones, con una ganancia de poco mas de un un segundo en su tiempo de ejecucion

- **riegoOptimo:**

  El paralelismo no es beneficioso en ningún tamaño. En todos los casos (7, 8 y 9 tablones), la versión paralelizada introduce una sobrecarga significativa en sus operaciones lo cual consume demasiados recursos.

4.2.2 ¿Las versiones paralelas introducen sobrecarga en casos pequeños?

Sí, en muchos casos las versiones paralelas tienen un tiempo mayor que las secuenciales para tamaños pequeños. Esto puede deberse a la sobrecarga de creación de tareas paralelas: En tamaños pequeños, la gestión del paralelismo (creación de threads, comunicación entre núcleos, etc.) toma más tiempo que la ejecución secuencial directa. También funciones con computación poco costosa: Si la operación no es intensiva en cálculos, el costo de coordinar las tareas paralelas supera cualquier beneficio.

Casos claros donde se introduce sobrecarga:

- **costoRiegoFinca (7 tablones):** La aceleración es apenas del 0.17, lo que sugiere que el paralelismo no aporta valor en este caso.

- **riegoOptimo (todos los tamaños):** El paralelismo introduce una sobrecarga significativa en todos los casos analizados.

4.2.3. Beneficios del paralelismo para diferentes escenarios

Los escenarios donde el paralelismo presenta beneficios surgen en funciones con gran cantidad de cálculos independientes, como GenerandoProgramaciones en fincas medianas (8 tablones). Aquí, el paralelismo aprovecha la división de trabajo entre núcleos para acelerar los cálculos.
Por el contrario, los escenarios donde el paralelismo no es beneficioso se presentan en funciones con tamaños pequeños o con baja complejidad computacional, como costoMovilidad para fincas pequeñas (7 tablones). También en operaciones que involucran gran coordinación o comunicación entre tareas, como riegoOptimo, donde la sobrecarga de paralelización supera los beneficios en cualquier tamaño analizado

4.2.4 Conclusión general del proyecto

Este proyecto ilustra de manera clara las fortalezas y limitaciones del uso del paralelismo en la programación funcional. Los puntos más relevantes incluyen:

- **Mejoras significativas en casos específicos**: El paralelismo resulta particularmente efectivo al procesar grandes volúmenes de datos y realizar cálculos independientes de alta demanda. Ejemplos como las funciones *costoRiegoFinca* y *generarProgramacionRiego* muestran buenos resultados en tamaños de entrada intermedios.
- **Sobrecarga en tareas pequeñas o dependientes**: Cuando se trabaja con entradas pequeñas o tareas que dependen entre sí, la sobrecarga generada por el paralelismo puede conducir a un rendimiento inferior al de una ejecución secuencial, como se observa en *riegoOptimo*.
- **Factores de eficiencia**: La eficiencia del paralelismo depende del equilibrio entre el costo computacional de la tarea y el esfuerzo necesario para coordinar las operaciones paralelas. Aplicar paralelismo de manera indiscriminada puede resultar ineficiente e incluso contraproducente.
- **Lecciones clave**: Es fundamental evaluar la naturaleza del problema antes de optar por una solución paralelizada, ya que no todas las operaciones se beneficiarán de este enfoque. Además, la realización de pruebas exhaustivas, como las ejecutadas en este proyecto, permite identificar los escenarios donde el paralelismo realmente aporta ventajas significativas.

En conclusión, el paralelismo es una herramienta poderosa cuando se implementa de forma adecuada, pero su uso sin un análisis cuidadoso puede generar una sobrecarga considerable. Este estudio subraya la importancia de realizar evaluaciones de rendimiento para optimizar el diseño y la implementación de soluciones concurrentes y funcionales.