Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-4700 Lenguajes de programación
Profesor: Ignacio Trejos Zelaya

Asignación #3 [Proyecto]: concurrencia y comunicación del lenguaje Go

Integrantes
Juan Sebastián Gamboa Botero - 2020030303
David Suárez Acosta - 2020038304

Fecha de entrega:
18 de Noviembre, 2021

Índice

Introducción

Dentro del lenguaje de programación Go (Golang) se puede programar de manera concurrente por medio del uso de “Gorrutinas” (goroutines) y de canales utilizados por estas para transportar información. En este trabajo los estudiantes aprenderán a aplicar ambos conceptos por medio de la graficación cuasi-paralela de 5 algoritmos de ordenamiento.

Gorrutinas

Las gorrutinas son funciones o métodos que se ejecutan independiente y simultáneamente del programa en sí. El uso de estas es bastante común en programas de Go por la facilidad de manejarlas y por su poco costo en comparación con threads (Ramanathan, 2021).

Comunicación mediante canales

Para la comunicación entre gorrutinas y con el programa en sí, se utiliza el concepto de canales (channels), estos se usan para enviar y recibir información. Los canales pueden ser “unbuffered” o “buffered”, en donde la diferencia entre estos es que en el buffered hay una capacidad máxima de datos que pueden estar enviando y recibiendo en el canal mientras que en el unbuffered, se bloquean con más facilidad los canales por exceso de información (Kutty, 2021).

Estrategia general para modificación de algoritmos de ordenamiento

Comunicar la función de ordenamiento con su función graficadora asociada

Para realizar esta comunicación, dentro de cada una de las funciones graficadoras se creó un canal con 1000 items de buffer que transfiere arreglos de enteros (los cuales son pares más específicamente) que se ingresan dentro de la función de ordenamiento asociada por medio de su llamada como gorrutina.

Provocar la graficación del arreglo base

Cada algoritmo de ordenamiento se ejecuta de manera normal con el único cambio de que al hacerse una modificación en el slice ingresado, ya sea un intercambio de valores entre dos índices o la asignación de un valor a un índice en específico, al canal creado se envía un arreglo con los índices a intercambiar sus valores o el índice y el valor a asignar a este.

Muestreo del tiempo al iniciar y al terminar un proceso de ordenamiento

Es un proceso bastante directo, por medio de la biblioteca de time, al inicio de cada algoritmo de ordenamiento se consigue el tiempo actual en la variable de initTime, y se vuelve a conseguir esto al final con la variable endTime, con estos 2 valores se le asigna a la variable respectiva de cada algoritmo la diferencia entre estos dos tiempos, y al momento de mostrar los datos este valor es mostrado en milisegundos. En algunos casos, debido a la velocidad de los algoritmos, el tiempo de ejecución es mínimo por lo que no se logra apreciar este en milisegundos.

Instrumentar el conteo de las operaciones elementales (intercambio de valores entre dos posiciones (índices) del arreglo, comparación entre valores, evaluación de condiciones de ciclos / pasadas enumerativas)

Se crearon sus respectivas variables para cada una de las funciones de ordenamiento, estas son (sin sus respectivos prefijos que dependen del ordenamiento): Swaps, Comparisons e Iterations. Cada una de estas es inicializada en 0.

- En el caso de Swaps (intercambio de valores), claramente este incrementa con cada vez que se haga un intercambio en el arreglo, por lo que se puede observar esto siempre que se manda información por el canal asociado.
- Para Comparisons, con cualquier “if” o statement que se pregunta antes de realizar una modificación en el arreglo ingresado, esta variable incrementa.
- Finalmente con Iterations, al inicio o final de cada ciclo utilizado esta variable se incrementa, esto incluye a todos los ciclos de for dobles que son utilizados en varias funciones.

Generación del gráfico de barras

Estrategia desarrollada para comunicarse con el algoritmo de ordenamiento

Como se mencionó en el primer punto de la sección anterior, se creó un canal de arreglos de números enteros (pares) dentro de la función graficadora que dependiendo del ordenamiento los valores que recibe son índices a intercambiar o un índice con su nuevo valor a asignar. En sí para poder actualizar la representación gráfica de los charts utilizados por cada ordenamiento, se realiza un for que por cada arreglo enviado por el canal, hace un swap a los datos del chart por medio de los índices, o solo al valor en el índice dado se le asigna el valor enviado.

Mapeo de valores hacia la región correspondiente al algoritmo

Para mapear los valores de cada copia del arreglo base en los charts de sus respectivos ordenamientos, primero se creó en sí el Chart a utilizar, al cual se le asigna la copia del arreglo al campo de Data. Al haber realizado esto, gracias a la librería utilizada eso es lo necesario para representar cada valor con una barra del tamaño respectivo en comparación a todos los números del arreglo, cada una de las barras está en orden ascendente dependiendo de su índice. Un dato a mencionar es que debido a que cada una de las regiones de los algoritmos no posee el mismo tamaño por la coordenadas (x,y) ingresadas, la altura de las barras puede variar en píxeles al representar su valor asignado.

Uso de bibliotecas de graficación externas o desarrolladas por el grupo

Las bibliotecas utilizadas para graficar fueron "github.com/gizak/termui/v3" y "github.com/gizak/termui/v3/widgets". Estas poseen una gran cantidad de métodos y tipos de datos que fueron utilizadas bastante. Este es el caso con los “Charts” en si, estos poseen el campo de Data que es lo que se grafica como fue mencionado en la parte anterior.
Para realizar esa animación con cada uno de los ordenamientos de manera simultánea, se necesita usar “sync.Mutex” que se utiliza para hacer “Lock” al proceso, luego se actualiza el chart en sí por medio del render, y luego “Unlock”. El Lock y Unlock se utilizan para sincronizar cada actualización a realizar de la terminal.

Bubble Sort

El canal utilizado para comunicar las funciones de abajo fue “pair” en donde se mandan los índices a cambiar a la función que dibuja el gráfico para intercambiar sus valores.

Algoritmo de ordenamiento

```
func bubbleSort(arr *[]float64, pair chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    //Move through all elements
    for i:= 0; i < len; i++ {
        for j := 0; j < len-i-1; j++ {
            // Move from 0 to len-i-1 and swap if element is greater than the next one
            if arr2[j] > arr2[j+1] {
                arr2[j], arr2[j+1] = arr2[j+1], arr2[j]

                pair <- []int{j, j+1} // Channel
                bsSwaps++
            }; bsComparisons++; bsIterations++
        }; bsIterations++
    }

    //*arr = arr2 // Assign changes to original array
    close(pair)

    endTime := time.Now() // Time End
    bsTime = endTime.Sub(initTime) // Total Time
}
```

Función graficadora

```
// / / / / Drawing

func bsChartDrawer(slice []float64){
    // / / bsChart.Data = slice
    bsChart.Data = make([]float64, len(slice))
    copy(bsChart.Data, slice)

    //Copy used in BubbleSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, bsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go bubbleSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
```

```
        swap(&bsChart.Data[pair[0]], &bsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&bsChart)
        m.Unlock()
    }

    playSound()

    //End
    bsChart.Title = "BubbleSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(bsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(bsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(bsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(bsIterations)
    m.Lock()
    ui.Render(&bsChart)
    m.Unlock()
}
```

Selection Sort

El canal utilizado para comunicar las funciones de abajo fue “pair” en donde se mandan los índices a cambiar a la función que dibuja el gráfico para intercambiar sus valores.

Algoritmo de ordenamiento

```
// Selection (changes indexes)

func selectionSort(arr *[]float64, pair chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    for currentIndex := 0; currentIndex < len-1; currentIndex++ { // Done to all the
indexes in the array
        indexMin := currentIndex

        for i := currentIndex + 1; i < len; i++ { // Get the index of the smallest
value from the numbers to the right
            if arr2[i] < arr2[indexMin] {
                indexMin = i
            }; ssComparisons++; ssIterations++
        }

        //Swap numbers
        arr2[currentIndex], arr2[indexMin] = arr2[indexMin], arr2[currentIndex]
        pair <- []int{currentIndex, indexMin} // Channel

        ssSwaps++
        ssIterations++
    }

    *arr = arr2 // Assign changes to original array
    close(pair)

    endTime := time.Now() // Time End
    ssTime = endTime.Sub(initTime) // Total Time
}
```

Función graficadora

```
func ssChartDrawer(slice []float64){
    // / / ssChart.Data = slice
    ssChart.Data = make([]float64, len(slice))
    copy(ssChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, ssChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go selectionSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&ssChart.Data[pair[0]], &ssChart.Data[pair[1]])
```

```
        m.Lock()
        ui.Render(&ssChart)
        m.Unlock()
    }

    playSound()

    //End
    ssChart.Title = "SelectionSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(ssTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(ssSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(ssComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(ssIterations)
    m.Lock()
    ui.Render(&ssChart)
    m.Unlock()
}
```

Insertion Sort

El canal utilizado para comunicar las funciones de abajo fue “oneWay” en donde se manda el índice del arreglo a cambiar y el valor que se le asigna a este.

Algoritmo de ordenamiento

```
// Insertion >> New <<

func insertionSort(arr *[]float64, oneWay chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    for i := 1; i < len; i++ {
        key := arr2[i]
        j := i-1

        //Move greater elements of arr[0 .. i-1] to position ahead of current
        isComparisons++ //isComparison????
        for ; j >= 0 && key < arr2[j]; j--{
            oneWay <- []int{j+1, int(arr2[j])} // Channel
            arr2[j+1] = arr2[j]

            isSwaps++
            isIterations++
        }

        oneWay <- []int{j+1, int(key)} // Channel
        arr2[j+1] = key

        isSwaps++
        isIterations++
    }


    *arr = arr2 // Assign changes to original array
    close(oneWay)

    endTime := time.Now() // Time End
    isTime = endTime.Sub(initTime) // Total Time
}
```

Función graficadora

```
func isChartDrawer(slice []float64){
    // / / isChart.Data = slice
    isChart.Data = make([]float64, len(slice))
    copy(isChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, isChart.Data)

    //Channel
    oneWayChannel := make(chan []int, 1000)
    go insertionSort(&copyArr, oneWayChannel)

    //Update Changes in pairs
```

```
    for oneWay := range oneWayChannel{
        isChart.Data[oneWay[0]] = float64(oneWay[1])//isChart.Data[oneWay[1]]
        m.Lock()
        ui.Render(&isChart)
        m.Unlock()
    }

    playSound()

    //End
    isChart.Title = "InsertionSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(isTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(isSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(isComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(isIterations)
    m.Lock()
    ui.Render(&isChart)
    m.Unlock()
}
```

Quicksort

El canal utilizado para comunicar las funciones de abajo fue “pair” en donde se mandan los índices a cambiar a la función que dibuja el gráfico para intercambiar sus valores.

Algoritmo de ordenamiento

```
// Quicksort [iterative for drawing]:
https://www.geeksforgeeks.org/iterative-quick-sort/

func partition(arr *[]float64, low int, high int, pair chan []int) int { //
    arr2 := *arr
    pivot := arr2[high]

    i := low - 1

    for j := low; j < high; j++ {
        if arr2[j] <= pivot {
            i++
            arr2[i], arr2[j] = arr2[j], arr2[i] //Gets the lesser values to the left
of the pivot
            pair <- []int{i, j} // Channel

            qsSwaps++
        }; qsComparisons++; qsIterations++
    }

    //Swap pivot with the next element to i
    arr2[i+1], arr2[high] = arr2[high], arr2[i+1]
    pair <- []int{i+1, high} // Channel

    qsSwaps++

    *arr = arr2 // Assign changes to original array

    return i + 1 //new pivot
}

func quickSort(arr *[]float64, pair chan []int){
    initTime := time.Now() // Time Start

    low := 0; high := len(*arr) - 1
    stack := make([]int, high+1) //Auxiliary stack

    top := -1 //Top of stack

    //Push high & low
    top++
    stack[top] = low
    top++
    stack[top] = high

    for top >= 0 {
        //Pop high & low
        high = stack[top]
        top--
        low = stack[top]
        top--

        pivot := partition(arr, low, high, pair) //pivot at correct position
```

```
        if pivot-1 > low { //If elements on left push left side to stack
            top++
            stack[top] = low
            top++
            stack[top] = pivot-1
        }; qsComparisons++

        if pivot+1 < high{
            top++
            stack[top] = pivot+1
            top++
            stack[top] = high
        }; qsComparisons++

        qsIterations++
    }

    close(pair)

    endTime := time.Now() // Time End
    qsTime = endTime.Sub(initTime) // Total Time
}
```

Función graficadora

```
func qsChartDrawer(slice []float64){
    // / / qsChart.Data = slice
    qsChart.Data = make([]float64, len(slice))
    copy(qsChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, qsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go quickSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&qsChart.Data[pair[0]], &qsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&qsChart)
        m.Unlock()
    }

    playSound()

    //End
    qsChart.Title = "QuickSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(qsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(qsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(qsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(qsIterations)
    m.Lock()
    ui.Render(&qsChart)
    m.Unlock()
}
```

Heapsort

El canal utilizado para comunicar las funciones de abajo fue “pair” en donde se mandan los índices a cambiar a la función que dibuja el gráfico para intercambiar sus valores. Algoritmo realizado de manera iterativa.

Algoritmo de ordenamiento

```
// Heapsort >> New << [iteraive just in case]:
https://www.geeksforgeeks.org/iterative-heap-sort/

func buildMaxHeap(arr *[]float64, n int, pair chan []int){
    arr2 := *arr
    for i := 1; i < n; i++{
        if arr2[i] > arr2[(i-1)/2]{ // Child bigger than parent
            j := i

            for arr2[j] > arr2[(j-1)/2]{ //Swap until parent is smaller than child
                arr2[j], arr2[(j-1)/2] = arr2[(j-1)/2], arr2[j]
                pair <- []int{j, (j-1)/2} // Channel
                j = (j-1)/2

                hsSwaps++
            }; hsIterations++
        }; hsComparisons++

        hsIterations++
    }

    *arr = arr2
}

func heapSort(arr *[]float64, pair chan []int){
    initTime := time.Now() // Time Start

    n := len(*arr)

    buildMaxHeap(arr, n, pair)
    arr2 := *arr

    for i := n-1; i > 0; i--{
        arr2[0], arr2[i] = arr2[i], arr2[0] //swap first with last
        pair <- []int{0, i} // Channel
        j, index := 0, 0

        hsSwaps++

        for {
            hsIterations++

            index = 2 * j + 1

            if index < (i - 1) && arr2[index] < arr2[index + 1]{
                index++
            }; hsComparisons+=2
            if index < i && arr2[j] < arr2[index]{
                arr2[j], arr2[index] = arr2[index], arr2[j]
                pair <- []int{j, index} // Channel
```

```
                hsSwaps++
            }; j = index
            hsComparisons++
            if index >= i{
                break
            }

        }; hsIterations++
    }

    *arr = arr2
    close(pair)

    endTime := time.Now() // Time End
    hsTime = endTime.Sub(initTime) // Total Time
}
```

Función graficadora

```
func hsChartDrawer(slice []float64){
    // / / hsChart.Data = slice
    hsChart.Data = make([]float64, len(slice))
    copy(hsChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, hsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go heapSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&hsChart.Data[pair[0]], &hsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&hsChart)
        m.Unlock()
    }

    playSound()

    //End
    hsChart.Title = "HeapSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(hsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(hsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(hsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(hsIterations)
    m.Lock()
    ui.Render(&hsChart)
    m.Unlock()
}
```

Dificultades

La primera dificultad con la cual nos encontramos al empezar este proyecto fue la elección de la biblioteca a utilizar. Inicialmente estuvimos buscando en internet distintas porque creiamos que ibamos a estar generando imágenes de cada uno de los gráficos con la biblioteca que se usó en la asignación anterior, pero gracias al ejemplo dado por el profesor, decidimos guiarnos por este y comprender esta biblioteca.

Con la parte de la graficación, claramente se iba a mandar arreglos copias del arreglo base para que no se modifiquen entre sí, pero al asignar al chart de cada uno de los ordenamientos esta copia como Data y mandar esta Data a ordenarse, claramente al estarse modificando dentro del algoritmo de ordenamiento y en si en la función graficadora, se ordenaba erróneamente. Este error fue detectado y arreglado por medio de la generación de otra copia, pero no fue evidente inicialmente.

En relación a los algoritmos de ordenamiento, el hecho de que preferimos no trabajar con recursividad debido a que en el ejemplo dado, el estudiante realiza el quicksort de manera iterativa, por lo que nosotros lo hicimos de la misma manera y también aplicamos esto al heapsort por si acaso.

Finalmente, se decidió utilizar la biblioteca de “math/rand” para generar el arreglo de números aleatorios, esto se debe a que al utilizar el método de congruencia lineal multiplicativa, se debían introducir muchas variables como parámetros y que dependiendo de la combinación utilizada, resultaba en un patrón diferente. El patrón más grande de números que se encontró con el método de congruencia lineal multiplicativa fue de nueve números distintos, lo cual evita que se visualicen correctamente los gráficos, resultando en una escalera con muy pocos escalones al terminar de ordenarse y menos dificultad al ordenarse por la repetición de una sola secuencia de números a lo largo del arreglo.

Código del Método de Congruencia Lineal Multiplicativa:

```
func randomSlice(size int, seed int, k int, period int) []float64 {

    // Validating "size"

    if size < 10 || size > 100 {
        fmt.Println("El valor size es incorrecto")
        return nil
    }

    // Validating Seed
    if seed < 11 || seed > 101 {
        fmt.Println("El valor de la semilla es incorrecto")
        return nil
    }

    for i := 2; i < seed; i++ { // Prime Number
        if seed % i == 0 {
            fmt.Println("El valor de la semilla es incorrecto")
```

```
            return nil
        }
    }

    // Validating "k"
    if k < 0 {
        fmt.Println("El valor k es incorrecto")
        return nil
    }

    // Validating "period"
    if period < 2048 {
        fmt.Println("El valor m es incorrecto")
        return nil
    }

    var slice = make([]float64, size)
    multiplier := 8*k + 3   // 8k + 5 can also be used

    /*
    first := (multiplier * seed) % period
    first = first % 30
    */

    for i := 0; i < size; i++ { // Generating the Array
        num := (multiplier * seed) % period // Main Algorithm, X = (multiplier * [seed
or previous number]) % period
        num = num % 30  // Changed to 0..29

        /*
          if num == first && i != 0{  // We can stop the loop to avoid repeating the
pattern
            slice = slice[:i]
            break
        }
        */

        slice[i] = float64(num)
        seed = num  // Seed is now the previous number
    }

    fmt.Println("Resultado: ", slice)

    return slice
}
```

Análisis de resultados

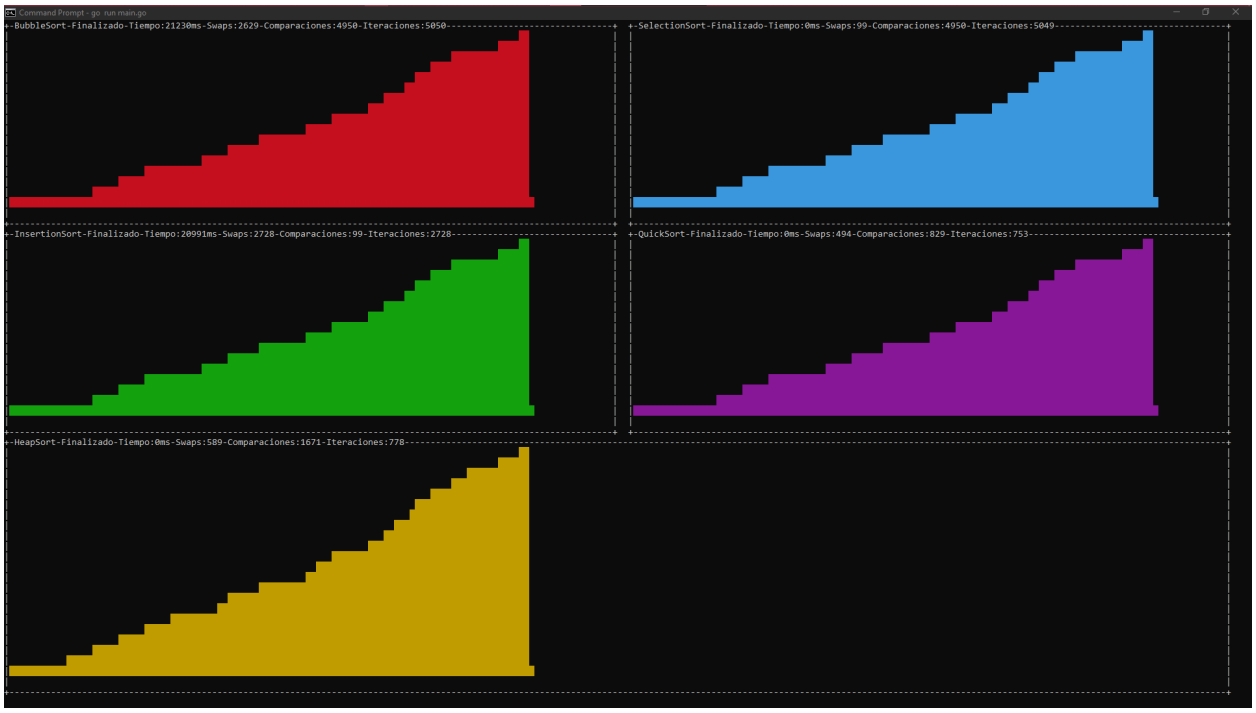

Figura 1: Captura de Pantalla de resultados obtenidos en el video Resultados.mkv

Como se puede observar en la figura 1, los resultados obtenidos consisten en:

	Tiempo (Milisegundos)	Swaps	Comparaciones	Iteraciones
BubbleSort	21230	2629	4950	5050
SelectionSort	0	99	4950	5049
InsertionSort	20991	2728	99	2728
QuickSort	0	494	829	753
HeapSort	0	589	1671	778

Cuadro 1: Resultados de figura 1

Como se puede observar, los mejores en tiempo son Selection, Quick y Heap, mientras que el peor es BubbleSort. El que tiene la menor cantidad de swaps/intercambios es el SelectionSort, mientras que el InsertionSort contiene la mayor cantidad. En menor cantidad de comparaciones es InsertionSort, mientras que los que tienen la mayor cantidad de comparaciones son ambos BubbleSort y SelectionSort. El que tiene la menor cantidad de iteraciones es QuickSort mientras que la mayor BubbleSort.

Con estos datos capturados, podemos decir con confianza que el algoritmo de ordenamiento menos eficiente y más costoso es el BubbleSort. Es el que más tiempo dura y realiza demasiadas comparaciones e iteraciones para poder ordenar un arreglo. En el caso del algoritmo más eficiente, según nuestros datos es el QuickSort al ser tan veloz que no se logra representar en milisegundos el tiempo de ejecución y que posee una poca cantidad de comparaciones e iteraciones.

Un dato que hay que tener en cuenta al observar el video de prueba realizado, es que la velocidad en que se realiza la animación no es la misma velocidad a la que se ejecutan los ordenamientos, esto es gracias a que se están mostrando los cambios visualmente a una velocidad comprensible mientras que si fueran en la velocidad del algoritmo no se podrían observar las modificaciones al gráfico efectivamente.

Link video “Resultados.mkv”:
https://drive.google.com/file/d/1kBpAOnDZdp5V9mbzAqYGmaFi6UZoX7fi/view?usp=sharing

Aportes por cada miembro

Juan Sebastian Gamboa Botero: Graficación de los algoritmos de ordenamiento y modificación de algoritmos de ordenamiento.
David Suarez Acosta: Algoritmos de ordenamiento con modificaciones, función generadora de números aleatorios y sonidos al finalizar animación.

Reflexión y Conclusiones

Gracias a este trabajo, los conceptos de gorrutinas y canales fueron comprendidos y puestos a prueba, esto se puede observar con el gran uso de las gorrutinas en varias secciones del código. Dentro del main se utilizan para correr simultáneamente cada una de las funciones graficadoras, mientras que dentro de estas funciones, se utilizan para llamar a los algoritmos de ordenamiento y se utilizan canales para pasar datos de una manera eficiente entre estas funciones.

El uso de las gorrutinas resultó bastante fácil de entender e implementar, y sobre todo, resultó bastante útil debido a que permite que se corran diferentes procesos al mismo tiempo y se pueden sincronizar con bastante facilidad en comparación a threads y corrutinas en otros lenguajes.

Finalmente, al poder graficar los algoritmos, se pudo observar claramente el funcionamiento de cada uno de estos y fue interesante graficarlos dentro de la terminal. Ninguno de los estudiantes había realizado algo tan peculiar en una terminal anteriormente y personalmente les gustó el proyecto.

Referencias

Iterative heapsort. GeeksforGeeks. (2019, July 31). Retrieved November 9, 2021, from https://www.geeksforgeeks.org/iterative-heap-sort/.

Iterative quick sort. GeeksforGeeks. (2021, September 6). Retrieved November 9, 2021, from https://www.geeksforgeeks.org/iterative-quick-sort/.

Kutty, A. (2021, March 25). *Concurrency in go: Goroutines and channels*. Go Chronicles. Retrieved November 12, 2021, from https://gochronicles.com/concurrency-in-go/.

Ramanathan, N. (2021, June 19). *Goroutines - concurrency in Golang*. Go Tutorial - Learn Go from the Basics with Code Examples. Retrieved November 12, 2021, from https://golangbot.com/goroutines/.

Apéndice con código fuente

```
package main

//go mod init main.go
//go mod tidy

import (
    "fmt"
    "log"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"

    ui "github.com/gizak/termui/v3"
    "github.com/gizak/termui/v3/widgets"
    "github.com/lxn/win"

    "github.com/faiface/beep/mp3"
    "github.com/faiface/beep/speaker"
)

// Variables
const (
    BAR_WIDTH = 1
    FONT_WIDTH = 8 //8
    FONT_HEIGHT = 16
    MAX_NUMBER_SIZE = 32
)

var (
    width int = int(win.GetSystemMetrics(win.SM_CXSCREEN) / FONT_WIDTH)
    height int = int(win.GetSystemMetrics(win.SM_CYSCREEN) /
(FONT_HEIGHT*3))
    m sync.Mutex

    // Charts
```

```
    bsChart widgets.BarChart
    ssChart widgets.BarChart
    isChart widgets.BarChart
    qsChart widgets.BarChart
    hsChart widgets.BarChart

    //Values per algorithm
    bsTime time.Duration
    bsSwaps = 0
    bsComparisons = 0
    bsIterations = 0

    ssTime time.Duration
    ssSwaps = 0
    ssComparisons = 0
    ssIterations = 0

    isTime time.Duration
    isSwaps = 0
    isComparisons = 0
    isIterations = 0

    qsTime time.Duration
    qsSwaps = 0
    qsComparisons = 0
    qsIterations = 0

    hsTime time.Duration
    hsSwaps = 0
    hsComparisons = 0
    hsIterations = 0
)

func main(){
    fmt.Print("Indique la cantidad de numeros a generar dentro de un
intervalo de 10 a 100 incluyendolos: ")

    var size int
    fmt.Scanln(&size)
    if err := ui.Init(); err != nil{
```

```
        log.Fatalf("failed to initialize termui: %v", err)
    }

    //Slice
    arregloBase := temporalRANDOM(size) // CAMBIAR TEMPORAL

    //Init
    initBSChart(arregloBase)
    initSSChart(arregloBase)
    initISChart(arregloBase)
    initQSChart(arregloBase)
    initHSChart(arregloBase)

    ui.Render(&bsChart)
    ui.Render(&ssChart)
    ui.Render(&isChart)
    ui.Render(&qsChart)
    ui.Render(&hsChart)

    //Goroutines & Drawing Start
    go bsChartDrawer(arregloBase)
    go ssChartDrawer(arregloBase)
    go isChartDrawer(arregloBase)
    go qsChartDrawer(arregloBase)
    hsChartDrawer(arregloBase)

    //Ending
    fmt.Scanln()
    ui.Close()
}

func temporalRANDOM(n int) []float64{
    arr := make([]float64, n)
    for i := 0; i < n; i++{
        arr[i] = float64(rand.Intn(30))
    }; return arr
}

// / / / / / / Sorting \ \ \ \ \ \
```

```
// Bubblesort >> New <<

func bubbleSort(arr *[]float64, pair chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    //Move through all elements
    for i:= 0; i < len; i++ {
        for j := 0; j < len-i-1; j++ {
            // Move from 0 to len-i-1 and swap if element is greater than
the next one
            if arr2[j] > arr2[j+1] {
                arr2[j], arr2[j+1] = arr2[j+1], arr2[j]

                pair <- []int{j, j+1} // Channel
                bsSwaps++
            }; bsComparisons++; bsIterations++
        }; bsIterations++
    }

    //*arr = arr2 // Assign changes to original array
    close(pair)

    endTime := time.Now() // Time End
    bsTime = endTime.Sub(initTime) // Total Time
}

// Selection (changes indexes)

func selectionSort(arr *[]float64, pair chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    for currentIndex := 0; currentIndex < len-1; currentIndex++ { // Done
to all the indexes in the array
        indexMin := currentIndex
```

```
        for i := currentIndex + 1; i < len; i++ { // Get the index of the
smallest value from the numbers to the right
            if arr2[i] < arr2[indexMin] {
                indexMin = i
            }; ssComparisons++; ssIterations++
        }

        //Swap numbers
        arr2[currentIndex], arr2[indexMin] = arr2[indexMin],
arr2[currentIndex]
        pair <- []int{currentIndex, indexMin} // Channel

        ssSwaps++
        ssIterations++
    }

    *arr = arr2 // Assign changes to original array
    close(pair)

    endTime := time.Now() // Time End
    ssTime = endTime.Sub(initTime) // Total Time
}

// Insertion >> New <<

func insertionSort(arr *[]float64, oneWay chan []int) {
    initTime := time.Now() // Time Start

    arr2 := *arr
    len := len(arr2)

    for i := 1; i < len; i++ {
        key := arr2[i]
        j := i-1

        //Move greater elements of arr[0 .. i-1] to position ahead of
current
        isComparisons++ //isComparison????
        for ; j >= 0 && key < arr2[j]; j--{
```

```
            oneWay <- []int{j+1, int(arr2[j])} // Channel
            arr2[j+1] = arr2[j]

            isSwaps++
            isIterations++
        }

        oneWay <- []int{j+1, int(key)} // Channel
        arr2[j+1] = key

        isSwaps++
        isIterations++
    }


    *arr = arr2 // Assign changes to original array
    close(oneWay)

    endTime := time.Now() // Time End
    isTime = endTime.Sub(initTime) // Total Time
}

// Quicksort [iterative for drawing]:
https://www.geeksforgeeks.org/iterative-quick-sort/

func partition(arr *[]float64, low int, high int, pair chan []int) int {
//
    arr2 := *arr
    pivot := arr2[high]

    i := low - 1

    for j := low; j < high; j++ {
        if arr2[j] <= pivot {
            i++
            arr2[i], arr2[j] = arr2[j], arr2[i] //Gets the lesser values
to the left of the pivot
            pair <- []int{i, j} // Channel

            qsSwaps++
```

```
        }; qsComparisons++; qsIterations++
    }

    //Swap pivot with the next element to i
    arr2[i+1], arr2[high] = arr2[high], arr2[i+1]
    pair <- []int{i+1, high} // Channel

    qsSwaps++

    *arr = arr2 // Assign changes to original array

    return i + 1 //new pivot
}

func quickSort(arr *[]float64, pair chan []int){
    initTime := time.Now() // Time Start

    low := 0; high := len(*arr) - 1
    stack := make([]int, high+1) //Auxiliary stack

    top := -1 //Top of stack

    //Push high & low
    top++
    stack[top] = low
    top++
    stack[top] = high

    for top >= 0 {
        //Pop high & low
        high = stack[top]
        top--
        low = stack[top]
        top--

        pivot := partition(arr, low, high, pair) //pivot at correct
position

        if pivot-1 > low { //If elements on left push left side to stack
            top++
```

```
            stack[top] = low
            top++
            stack[top] = pivot-1
        }; qsComparisons++

        if pivot+1 < high{
            top++
            stack[top] = pivot+1
            top++
            stack[top] = high
        }; qsComparisons++

        qsIterations++
    }

    close(pair)

    endTime := time.Now() // Time End
    qsTime = endTime.Sub(initTime) // Total Time
}

// Heapsort >> New << [iteraive just in case]:
https://www.geeksforgeeks.org/iterative-heap-sort/

func buildMaxHeap(arr *[]float64, n int, pair chan []int){
    arr2 := *arr
    for i := 1; i < n; i++{
        if arr2[i] > arr2[(i-1)/2]{ // Child bigger than parent
            j := i

            for arr2[j] > arr2[(j-1)/2]{ //Swap until parent is smaller
than child
                arr2[j], arr2[(j-1)/2] = arr2[(j-1)/2], arr2[j]
                pair <- []int{j, (j-1)/2} // Channel
                j = (j-1)/2

                hsSwaps++
            }; hsIterations++
        }; hsComparisons++
```

```
        hsIterations++
    }

    *arr = arr2
}

func heapSort(arr *[]float64, pair chan []int){
    initTime := time.Now() // Time Start

    n := len(*arr)

    buildMaxHeap(arr, n, pair)
    arr2 := *arr

    for i := n-1; i > 0; i--{
        arr2[0], arr2[i] = arr2[i], arr2[0] //swap first with last
        pair <- []int{0, i} // Channel
        j, index := 0, 0

        hsSwaps++

        for {
            hsIterations++

            index = 2 * j + 1

            if index < (i - 1) && arr2[index] < arr2[index + 1]{
                index++
            }; hsComparisons+=2
            if index < i && arr2[j] < arr2[index]{
                arr2[j], arr2[index] = arr2[index], arr2[j]
                pair <- []int{j, index} // Channel

                hsSwaps++
            }; j = index
            hsComparisons++
            if index >= i{
                break
            }
```

```
        }; hsIterations++
    }

    *arr = arr2
    close(pair)

    endTime := time.Now() // Time End
    hsTime = endTime.Sub(initTime) // Total Time
}


// / / / / / / Graphic \ \ \ \ \ \

// / / / / Initialize

func initBSChart(arr []float64){
    bsChart = *widgets.NewBarChart()
    bsChart.Data = arr
    bsChart.BarWidth = BAR_WIDTH
    bsChart.BarGap = 0

    //Changes per Chart
    bsChart.Title = "BubbleSort"
    bsChart.SetRect(0, 0, width/2 - 2, height-2)
    bsChart.BarColors = []ui.Color{ui.ColorRed}
    bsChart.NumStyles = []ui.Style{ui.NewStyle(ui.ColorRed)} // Can't be
seen

    // Indexes
    //bsChart.Labels = generateLabels(arr)
    //bsChart.LabelStyles = []ui.Style{ui.NewStyle(ui.ColorWhite)}
}

func initSSChart(arr []float64){
    ssChart = *widgets.NewBarChart()
    ssChart.Data = arr
    ssChart.BarWidth = BAR_WIDTH
    ssChart.BarGap = 0

    //Changes per Chart
```

```
    ssChart.Title = "SelectionSort"
    ssChart.SetRect(width/2, 0, width - 4, height-2)
    ssChart.BarColors = []ui.Color{ui.ColorCyan}
    ssChart.NumStyles = []ui.Style{ui.NewStyle(ui.ColorCyan)} // Can't be
seen
}

func initISChart(arr []float64){
    isChart = *widgets.NewBarChart()
    isChart.Data = arr
    isChart.BarWidth = BAR_WIDTH
    isChart.BarGap = 0

    //Changes per Chart
    isChart.Title = "InsertionSort"
    isChart.SetRect(0, height-2, width/2 - 2, height*2-4)
    isChart.BarColors = []ui.Color{ui.ColorGreen}
    isChart.NumStyles = []ui.Style{ui.NewStyle(ui.ColorGreen)} // Can't be
seen
}

func initQSChart(arr []float64){
    qsChart = *widgets.NewBarChart()
    qsChart.Data = arr
    qsChart.BarWidth = BAR_WIDTH
    qsChart.BarGap = 0

    //Changes per Chart
    qsChart.Title = "QuickSort"
    qsChart.SetRect(width/2, height-2, width - 4, height*2-4)
    qsChart.BarColors = []ui.Color{ui.ColorMagenta}
    qsChart.NumStyles = []ui.Style{ui.NewStyle(ui.ColorMagenta)} // Can't
be seen
}

func initHSChart(arr []float64){
    hsChart = *widgets.NewBarChart()
    hsChart.Data = arr
    hsChart.BarWidth = BAR_WIDTH
    hsChart.BarGap = 0
```

```
    //Changes per Chart
    hsChart.Title = "HeapSort"
    hsChart.SetRect(0, height*2-4, width-4, height*3-1)
    hsChart.BarColors = []ui.Color{ui.ColorYellow}
    hsChart.NumStyles = []ui.Style{ui.NewStyle(ui.ColorYellow)} // Can't
be seen
}

// / / / / Drawing

func bsChartDrawer(slice []float64){
    // / / bsChart.Data = slice
    bsChart.Data = make([]float64, len(slice))
    copy(bsChart.Data, slice)

    //Copy used in BubbleSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, bsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go bubbleSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&bsChart.Data[pair[0]], &bsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&bsChart)
        m.Unlock()
    }

    playSound()

    //End
    bsChart.Title = "BubbleSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(bsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(bsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(bsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(bsIterations)
```

```
    m.Lock()
    ui.Render(&bsChart)
    m.Unlock()
}

func ssChartDrawer(slice []float64){
    // / / ssChart.Data = slice
    ssChart.Data = make([]float64, len(slice))
    copy(ssChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, ssChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go selectionSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&ssChart.Data[pair[0]], &ssChart.Data[pair[1]])
        m.Lock()
        ui.Render(&ssChart)
        m.Unlock()
    }

    playSound()

    //End
    ssChart.Title = "SelectionSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(ssTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(ssSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(ssComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(ssIterations)
    m.Lock()
    ui.Render(&ssChart)
    m.Unlock()
}

func isChartDrawer(slice []float64){
```

```
    // / / isChart.Data = slice
    isChart.Data = make([]float64, len(slice))
    copy(isChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, isChart.Data)

    //Channel
    oneWayChannel := make(chan []int, 1000)
    go insertionSort(&copyArr, oneWayChannel)

    //Update Changes in pairs
    for oneWay := range oneWayChannel{
        isChart.Data[oneWay[0]] =
float64(oneWay[1])//isChart.Data[oneWay[1]]
        m.Lock()
        ui.Render(&isChart)
        m.Unlock()
    }

    playSound()

    //End
    isChart.Title = "InsertionSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(isTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(isSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(isComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(isIterations)
    m.Lock()
    ui.Render(&isChart)
    m.Unlock()
}

func qsChartDrawer(slice []float64){
    // / / qsChart.Data = slice
    qsChart.Data = make([]float64, len(slice))
    copy(qsChart.Data, slice)

    //Copy used in SelectionSort
```

```
    copyArr := make([]float64, len(slice))
    copy(copyArr, qsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go quickSort(&copyArr, pairsChannel)

    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&qsChart.Data[pair[0]], &qsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&qsChart)
        m.Unlock()
    }

    playSound()

    //End
    qsChart.Title = "QuickSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(qsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(qsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(qsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(qsIterations)
    m.Lock()
    ui.Render(&qsChart)
    m.Unlock()
}

func hsChartDrawer(slice []float64){
    // / / hsChart.Data = slice
    hsChart.Data = make([]float64, len(slice))
    copy(hsChart.Data, slice)

    //Copy used in SelectionSort
    copyArr := make([]float64, len(slice))
    copy(copyArr, hsChart.Data)

    //Channel
    pairsChannel := make(chan []int, 1000)
    go heapSort(&copyArr, pairsChannel)
```

```
    //Update Changes in pairs
    for pair := range pairsChannel{
        swap(&hsChart.Data[pair[0]], &hsChart.Data[pair[1]])
        m.Lock()
        ui.Render(&hsChart)
        m.Unlock()
    }

    playSound()

    //End
    hsChart.Title = "HeapSort-Finalizado-" +
        "Tiempo:"+strconv.FormatInt(hsTime.Milliseconds(),10)+"ms-" +
        "Swaps:"+strconv.Itoa(hsSwaps)+"-" +
        "Comparaciones:"+strconv.Itoa(hsComparisons)+"-"+
        "Iteraciones:"+strconv.Itoa(hsIterations)
    m.Lock()
    ui.Render(&hsChart)
    m.Unlock()
}

// / / / / / / Extra \ \ \ \ \ \

func swap (a *float64, b *float64){
    temp := *a
    *a = *b
    *b = temp
}

func playSound(){
    f, _ := os.Open("w.mpeg")
    streamer, format, _ := mp3.Decode(f)
    speaker.Init(format.SampleRate, format.SampleRate.N(time.Second/10))
    speaker.Play(streamer)
}

func randomSlice(size int, seed int, k int, period int) []float64 {

    // Validating "size"
```

```
if size < 10 || size > 100 {
    fmt.Println("El valor size es incorrecto")
    return nil
}

// Validating Seed
if seed < 11 || seed > 101 {
    fmt.Println("El valor de la semilla es incorrecto")
    return nil
}

for i := 2; i < seed; i++ { // Prime Number
    if seed % i == 0 {
        fmt.Println("El valor de la semilla es incorrecto")
        return nil
    }
}

// Validating "k"
if k < 0 {
    fmt.Println("El valor k es incorrecto")
    return nil
}

// Validating "period"
if period < 2048 {
    fmt.Println("El valor m es incorrecto")
    return nil
}

var slice = make([]float64, size)
multiplier := 8*k + 3   // 8k + 5 can also be used

/*
first := (multiplier * seed) % period
first = first % 30
*/

for i := 0; i < size; i++ { // Generating the Array
```

```
        num := (multiplier * seed) % period // Main Algorithm, X =
(multiplier * [seed or previous number]) % period
        num = num % 30  // Changed to 0..29

        /*
        if num == first && i != 0{  // We can stop the loop to avoid
repeating the pattern
            slice = slice[:i]
            break
        }
        */

        slice[i] = float64(num)
        seed = num  // Seed is now the previous number
    }

    fmt.Println("Resultado: ", slice)

    return slice
}
```