Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Práctica 1

Algoritmos de ordenación

Jorge Solán Morote	816259
Francisco Javier Pizarro	821259

11/09/2023

Lenguaje empleado

Para el desarrollo de esta práctica se ha elegido emplear Golang dada su facilidad para la manipulación de vectores por medio de los denominados *Slices* así como la facilidad para implementar la concurrencia en aquellos algoritmos que lo permitan mediante el uso de las *Gorutinas* y de los *channels*. Adicionalmente ofrece otras grandes ventajas como ser un lenguaje fuertemente tipado y ofrecer un alto rendimiento.

Algoritmos empleados

Se ha decidido implementar más algoritmos de los exigidos para realizar una comparación más amplia y exhaustiva de los mismos. A continuación se detalla un listado de los algoritmos empleados y las razones de la elección de cada uno.

Para la generación de datasets aleatorios se ha empleado el lenguaje de scripting Ruby dada su facilidad de lectura.

BogoSort: Si queremos comparar rendimientos, es casi obligado el hecho de comparar las soluciones “buenas” con la solución más simple/”absurda” posible. Este algoritmo sigue la filosofía de la siguiente afirmación. “Si hay un número infinito de monos empleando máquinas de escribir durante un tiempo infinito, eventualmente uno de ellos logrará escribir El Quijote”.

PancakeSort: El algoritmo funciona de la siguiente manera: en cada paso, seleccionamos la tortita más grande que está en el conjunto desordenado y lo colocamos en la parte superior. Luego, damos vuelta a toda la pila de tortitas para que la tortita más grande quede en la parte inferior. Repetimos este proceso para cada tortita, comenzando desde el más grande hasta el más pequeño. Al final, todos las tortitas están ordenadas de manera eficiente.

BubbleSort: Es el algoritmo de ordenamiento más intuitivo, es el primero que se aprende cuando se comienza a programar. Este se basa en 2 bucles anidados, el bucle exterior recorre todos los elementos, mientras que el interior recorre desde el elemento que se encuentra el bucle exterior hasta el final del vector, ordenando dicho elemento respecto a los demás.

HeapSort: Este algoritmo es ligeramente más sofisticado que los anteriores dado que requiere de un TAD tipo Heap para su correcto funcionamiento, este se basa en el principio de que al incorporar un dato ordenado a un Heap este lo añade de forma de que el Heap mantenga un orden de mayor a menor.

TreeSort: Este algoritmo al igual que el anterior se basa en el uso de un TAD, concretamente un árbol binario de búsqueda, dada la definición de dicho TAD, para obtener los elementos ordenados simplemente debemos generar el árbol completo y posteriormente recorrerlo en *postOrder*.

QuickSort: Este es uno de los algoritmos de ordenación más utilizados, dada su eficiencia. Este consiste en elegir el primer elemento de un vector como pivote, posteriormente el nuevo vector ordenado se define como la concatenación de: los resultados de ordenar con dicho algoritmo aquellos elementos que sean menores que el pivote, el pivote y los resultados de ordenar dicho algoritmo aquellos elementos que sean mayores o iguales que el pivote.

MergeSort: Es uno de los algoritmos de referencia respecto a ordenación, se basa en la metodología de Divide y vencerás. Consiste en la mezcla de las dos mitades del vector, después de ordenar estas con dicho algoritmo.

RadixSort: Este es un algoritmo de ordenación más rápidos y eficientes, se basa en la ordenación ordenada de cada una de sus cifras empezando de derecha a izquierda (hay una implementación similar de izquierda a derecha) . En nuestra implementación para hacer la ordenación individual de cada cifra se ha optado por la implementación de un algoritmo de ordenación auxiliar llamado “Counting Sort”, el cual funciona muy bien para números de una sola cifra. Así se ordenan los números de menor a mayor iterando sobre sus cifras, con lo cual funciona mejor para número de pocas cifras y algo peor para números mayores.

Implementaciones concurrentes: Dado que algunos algoritmos son paralelizables, esto es una ventaja muy significativa respecto a los que no, es por ello que se debe analizar también el aumento de rendimiento que dicha cualidad permite. Se han implementado en su versión concurrente los siguientes algoritmos: Bogosort, Mergesort y Quicksort.

Pasos para ejecutarlo

Dado que se han empleado los lenguajes Golang y Ruby, para poder ejecutar esto se debe emplear el servidor central del cps de la universidad de Zaragoza.(accesible vía ssh central.cps.unizar.es)

Desempaquetar el archivo comprimido facilitado dentro de una carpeta que se encuentre montada en el servidor central.cps.unizar.es

Dentro de dicha carpeta ejecutar ./ejecutar.sh

Por motivos de cuota de disco la generación y uso del dataset grande se ha deshabilitado dado que este ocupa más de 1GB de espacio, adicionalmente el dataset pequeño se ha deshabilitado también dado el coste temporal del bogosort.

No se espera que graficas.py sea ejecutado, dado que este fue un script adicional para generar las gráficas de esta memoria.

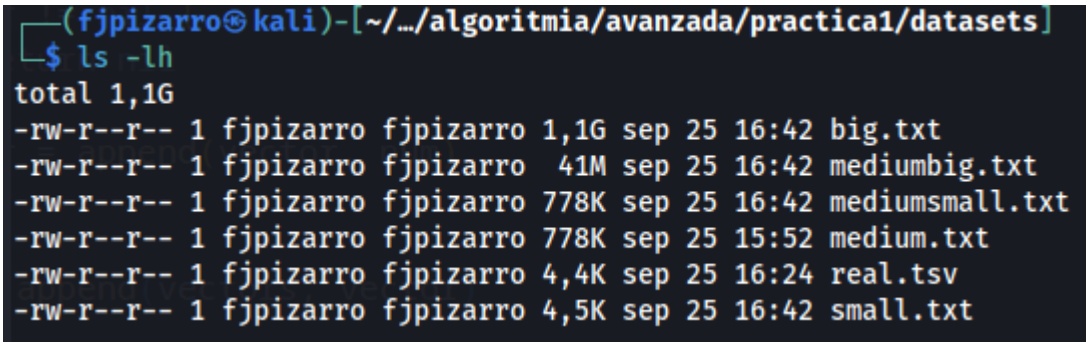
Datasets de rendimiento

Para realizar unas pruebas objetivas de rendimiento se ha optado por emplear datasets generados de forma aleatoria de distintos tamaños, tras una fase de experimentación se ha asociado a cada dataset unos algoritmos concretos para resolverlo de forma que los algoritmos más eficientes son empleados con los datasets más grandes.

Respecto al tamaño de los datasets, el más grande de ellos tiene un vector de **150000000** componentes. Dicho tamaño de dataset ha causado una implicación en el código dado que por defecto el lector de ficheros de golang no está configurado para admitir semejante cantidad de datos por lo que se tuvo que configurar de forma específica que este lector contase con 2 GB de RAM para leer ficheros.

```
var vectors [][]int
scanner := bufio.NewScanner(file)
const maxBufferSize = 1024 * 1024 * 1024 * 2// 2 GB buffer size
buf := make([]byte, maxBufferSize)
scanner.Buffer(buf, maxBufferSize)
```

Otro detalle que cabe resaltar del dataset grande es que este fichero ocupa más de **1GB** en disco.

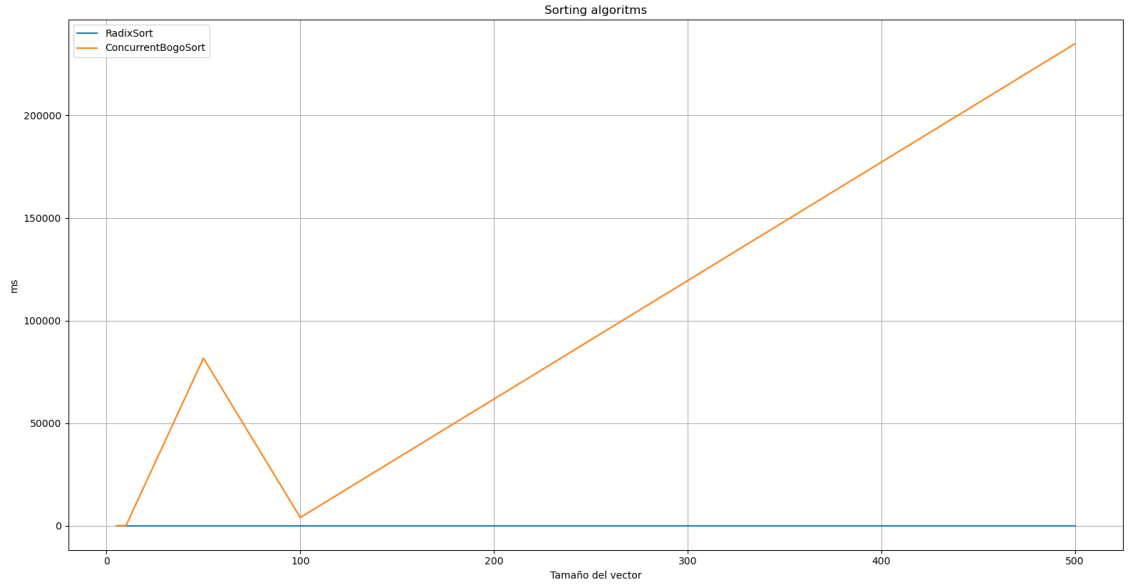


Para contrastar la veracidad de la eficiencia de los algoritmos con datos reales se ha recurrido a ordenar datos demográficos de europa, dicho dataset se encuentra disponible en https://ec.europa.eu/eurostat/web/population-demography/population-housing-censuses/database

Resultados experimentales

Dataset pequeño

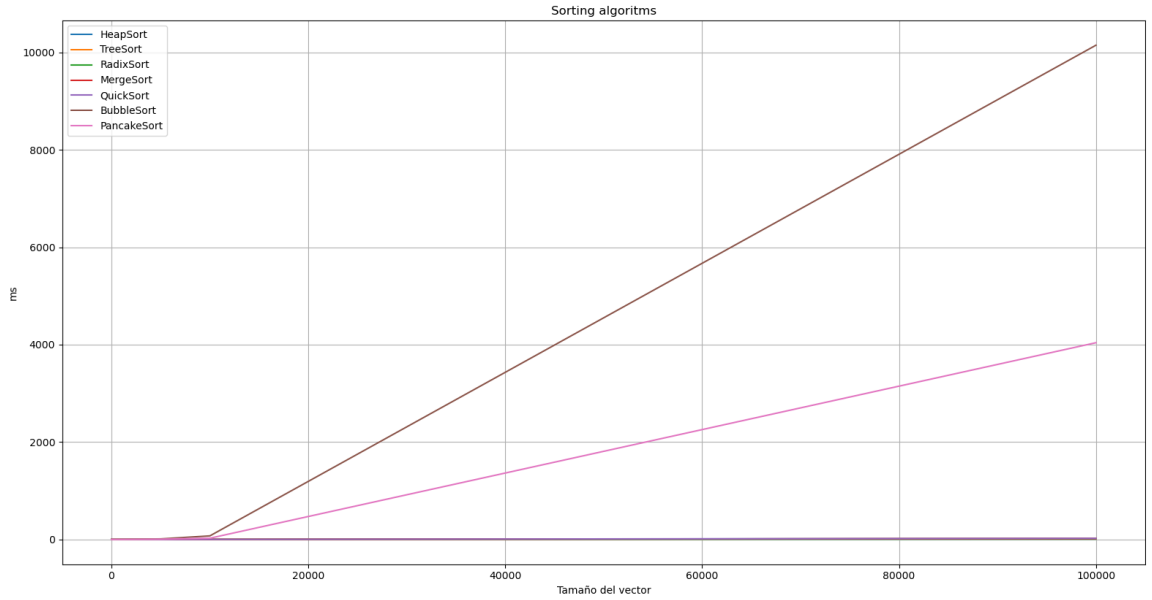
Como cabía esperar, el radixSort mantiene su coste temporal por debajo de 0.1 ms mientras que el concurrentBogosort incrementa mucho su coste temporal a pesar del muy reducido tamaño del vector



Dataset medio-pequeño

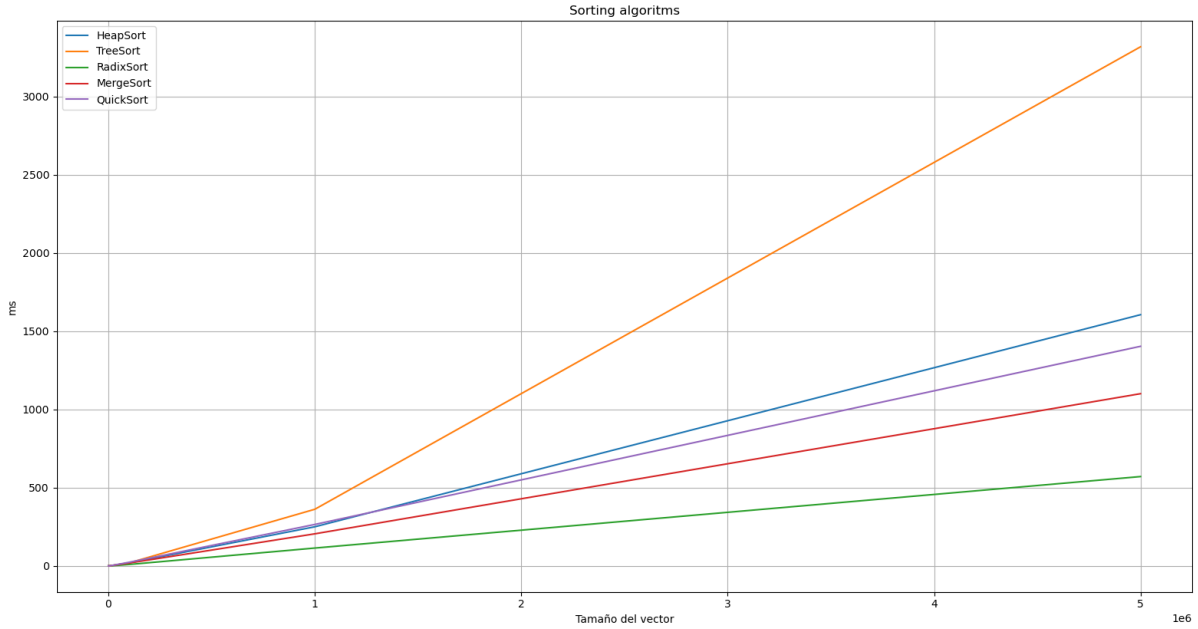
Lo primero que se percibe de forma casi inmediata es que a pesar del tamaño, el tiempo de todos los algoritmos es infinitamente mejor que el del previo concurrentBogosort.

Asimismo se aprecia que los algoritmos que peor gestiona el incremento de tamaño del vector son el bubbleSort y el pancakeSort



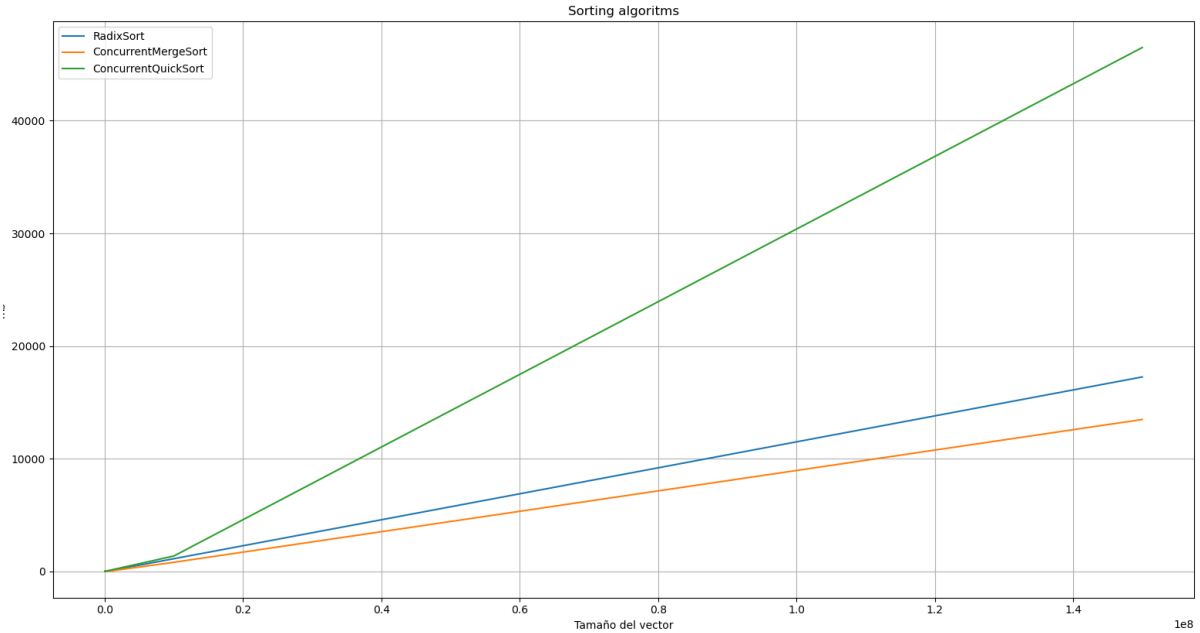
Dataset medio-grande

La conclusión que se extrae de este dataset es que aquellos algoritmos que dependen de estructuras de datos, escalan peor que aquellos que no. Asimismo los algoritmos más eficientes son el radixSort y el mergeSort.



Dataset grande

Cómo nos fijamos más en aumentar el tamaño del vector y no el de los números el algoritmo de radixSort sigue siendo de los más eficientes. No obstante, como era de esperar, la implementación concurrente del mergeSort es la que mejor escala en tiempo conforme al tamaño del vector.



Dataset real

En este caso el vector es tan pequeño(612 componentes), que todos los algoritmos empleados en este terminan un tiempo inferior a 0.1 ms por lo que no tiene sentido alguno mostrar una gráfica de este dataset.

Anexo I

Implementación algoritmos

```
func RadixSort(ints IntVector, verbose bool) {
max := getMax(ints)

for exp := 1; max/exp > 0; exp *= 10 {
ints = countingSort(ints, exp)
}
if (verbose) {
fmt.Println(ints)
}
return
}
```

```
func QuickSort(ints IntVector, verbose bool) {
result := recQuickSort(ints)
if (verbose) {
fmt.Println(result)
}
return
}

func recQuickSort(ints IntVector) IntVector {
if ints != nil && len(ints) > 1 {
pivote := ints[0]
menoresIguales, mayores := divideInLowersAndGreaters(pivote, ints[1:])
return append(append([]int(recQuickSort(menoresIguales)), ints[:1]...),
[]int(recQuickSort(mayores))...)
} else {
if len(ints) == 1 {
return ints
} else {
return nil
}
}
}

func ConcurrentQuickSort(ints IntVector, verbose bool) {
retChan := make(chan IntVector)
go concurrentRecQuickSort(ints,retChan,4)
```

```
resultado := <- retChan

if (verbose) {
fmt.Println(resultado)
}
}

func concurrentRecQuickSort(ints IntVector, ret chan IntVector, w int)
{
N := len(ints)
var a,b IntVector
if N > 1 {
pivote := ints[0]
menoresIguales , mayores := divideInLowersAndGreaters(pivote, ints[1:])
if (w > 0) {
lowerRes, higherRes := make(chan IntVector), make(chan IntVector)
go concurrentRecQuickSort(menoresIguales, lowerRes, w - 1)
go concurrentRecQuickSort(mayores, higherRes, w - 1)
a = <- lowerRes
b = <- higherRes
} else {
a = recQuickSort(menoresIguales)
b = recQuickSort(mayores)
}
finalVec := append(append(a, pivote), b...)
ret <- finalVec
} else {
ret <- ints
}
}
```

```
func bogoSortInstance(seguir, encontrado chan bool, ints IntVector, res
chan IntVector) {
sigo := true
rand.Seed(time.Now().UnixNano())

for sigo {
newVec := shuffle(ints)
sorted := isSorted(newVec)
encontrado <- sorted
sigo = <-seguir
```

```
if (sorted) {
res <- newVec
}
}
}

func ConcurrentBogoSort(ints IntVector, verbose bool) {
keepSearching := true
nWorkers := 10
seguir := make(chan bool)
encontrado := make(chan bool)
res := make(chan IntVector)
for I := 0; I < nWorkers; I++ {
go bogoSortInstance(seguir, encontrado, ints, res)
}

for keepSearching {
keepSearching = !<-encontrado
seguir <- keepSearching
}
resultado := <- res
if (verbose) {
fmt.Println(resultado)
}
nRestantes := nWorkers - 1
for nRestantes > 0 {
<-encontrado
seguir <- keepSearching
nRestantes--
}
return
}
```

```
func recMergeSort(ints IntVector) IntVector {
N := len(ints)
if N > 1 {
firstHalf := recMergeSort(ints[:N/2])
secondHalf := recMergeSort(ints[N/2:])
mergedVec := merge(firstHalf, secondHalf)
return mergedVec
} else {
```

```
return ints
}
}

func MergeSort(ints IntVector, verbose bool) {
resultado := recMergeSort(ints)
if (verbose) {
fmt.Println(resultado)
}
}

func ConcurrentMergeSort(ints IntVector, verbose bool) {
retChan := make(chan IntVector)
go concurrentRecMergeSort(ints,retChan,4)
resultado := <- retChan

if (verbose) {
fmt.Println(resultado)
}
}

func concurrentRecMergeSort(ints IntVector, ret chan IntVector, w int)
{
N := len(ints)
var a,b IntVector
if N > 1 {
if (w > 0) {
resultados := make(chan IntVector)
go concurrentRecMergeSort(ints[:N/2], resultados, w - 1)
go concurrentRecMergeSort(ints[N/2:], resultados, w - 1)
a = <- resultados
b = <- resultados
} else {
a = recMergeSort(ints[:N/2])
b = recMergeSort(ints[N/2:])
}
mergedVec := merge(a, b)
ret <- mergedVec
} else {
ret <- ints
}
}
```

```
func BubbleSort(ints IntVector, verbose bool) {
N := len(ints)
aux := 0
for i := range ints {
for j := i; j < N; j++ {
if ints[i] > ints[j] {
aux = ints[i]
ints[i] = ints[j]
ints[j] = aux
}
}
}
if (verbose) {
fmt.Println(ints)
}

return
}
```

```
func HeapSort(ints IntVector, verbose bool) {
h := &IntHeap{}
heap.Init(h)
heap.Push(h, 3)
for _, v := range ints {
heap.Push(h, v)
}
for h.Len() > 0 {
if (verbose) {
fmt.Printf("%d ", heap.Pop(h))
} else {
heap.Pop(h)
}
}
return
}
```

```
func PancakeSort(ints IntVector, verbose bool) {
N := len(ints)
currSize := N
for currSize > 1 {
maxIndex := findMaxIndex(ints[:currSize])
```

```
if (maxIndex != (currSize - 1)) {
flip(ints,maxIndex)
flip(ints, currSize - 1)
}
currSize--
}

if (verbose) {
fmt.Println(ints)
}
return
}
```

```
func TreeSort(ints IntVector, verbose bool) {
var t Tree
for _, v := range ints {
t.insert(v)
}
if (verbose) {
fmt.Print("[")
}
postOrder(t.root , verbose)
if (verbose) {
fmt.Print("]")
}
return
}
```