PROPOSITO: Algoritmos de ordenación: algoritmo de la burbuja implementado con el código de clase y con otro código, y algoritmo de ordenación de la inserción. Gráfica de comparación de los mismos

Práctica de ordenación 1

Sergio Casado López y Sandra Gómez Gálvez
Ingeniería del Software y Matemáticas

Algoritmo burbuja

El siguiente código muestra nuestro código creado para el algoritmo burbuja.

El algoritmo burbuja ordena de menor a mayor un array (o una lista).

La función llamada “ordenar” es la función que nosotros hemos creado para ordenar un vector. Donde compara cada elemento con el siguiente, y si es mayor lo intercambia.

```
################## ALGORITMO BURBUJA 0(n^2) ###################################
    # Queremos ordenar de menor a mayor, comparando cada elemento con el siguiente

############### Nuestro código:
  ordenar<- function(x){
    n<- length(x)           # Cogemos la longitud de x
                                # Recordamos que los Arrays en R comienzan en 1
    for(i in 2:n){          # Realizamos un bucle for anidado desde i=2 hasta n
      for(j in 1:(n-1)){      # y desde j=1 hasta n-1
        if(x[j]>x[j+1]){    # Si x en la posición j es mayor que x en la posición j+1 hacemos:
          v<- x[j];           # v = x en la posición j
          x[j]<-x[j+1];       # x en la posición j será = x en la posición j+1
          x[j+1]<-v;          # x en la posición j+1 será = v
        }
      }
    }
    x                       # Mostramos x
  }
```

Las siguientes funciones son las creadas en clase y utilizadas para realizar una búsqueda por burbuja.

La función intercambio asigna el valor de pair[i] en pair [j].

```
################ Bubblesort de clase:
  # Función intercambio: asigna el valor de pair[i] en pair[j]
  intercambio=function(pair,i,j){
    aux=pair[i];
    pair[j]=aux;
    pair
  }
```

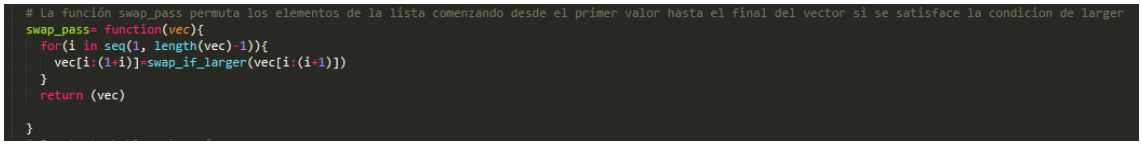
La función larger va a comparar dos elementos de una lista.

```
  # La función larger va a comparar dos elementos de una lista
  larger= function(pair){
    if((pair[1]>pair[2])){
      return(TRUE)
    }else{
      return(FALSE)
    }

  }
```

La función swap_if_larger permuta los elementos de la lista si se satisface la condición de larger.

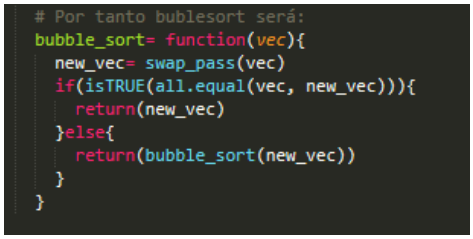
```
  # La función swap_if_larger permuta los elementos de la lista si se satisface la condicion de larger
  swap_if_larger=function(pair){
    if(larger(pair)){
      return(rev(pair)) #Rev le da la vuelta
    }else{
      return(pair)
    }
  }
```

La función swap_pass permuta los elementos de la lista comenzando desde el primer valor hasta el final del vector si se satisface la condición de larger.



```
# La función swap_pass permuta los elementos de la lista comenzando desde el primer valor hasta el final del vector si se satisface la condicion de larger
swap_pass= function(vec){
  for(i in seq(1, length(vec)-1)){
    vec[i:(1+i)]=swap_if_larger(vec[i:(i+1)])
  }
  return (vec)

}
```

Por tanto, la función Bubble_sort será:



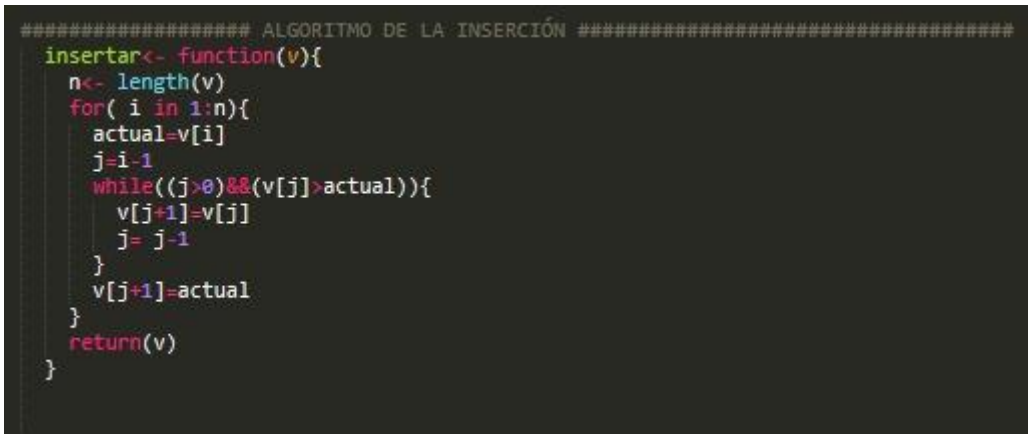
```
# Por tanto bublesort será:
bubble_sort= function(vec){
  new_vec= swap_pass(vec)
  if(isTRUE(all.equal(vec, new_vec))){
    return(new_vec)
  }else{
    return(bubble_sort(new_vec))
  }
}
```

Algoritmo de la inserción

El siguiente código muestra el algoritmo de la inserción creado en la función insertar.

El algoritmo de la inserción ordena de menor a mayor un array (o una lista).

Este algoritmo recorrerá un bucle for de 1 hasta n, y por cada elemento k lo comparará con los siguientes k+1 elementos hasta que encuentra un elemento de valor mayor al elemento k y se detiene, en esa posición se insertará.



```
#################### ALGORITMO DE LA INSERCIÓN ####################################
insertar<- function(v){
  n<- length(v)
  for( i in 1:n){
    actual=v[i]
    j=i-1
    while((j>0)&&(v[j]>actual)){
      v[j+1]=v[j]
      j= j-1
    }
    v[j+1]=actual
  }
  return(v)
}
```

Gráfica de tiempos

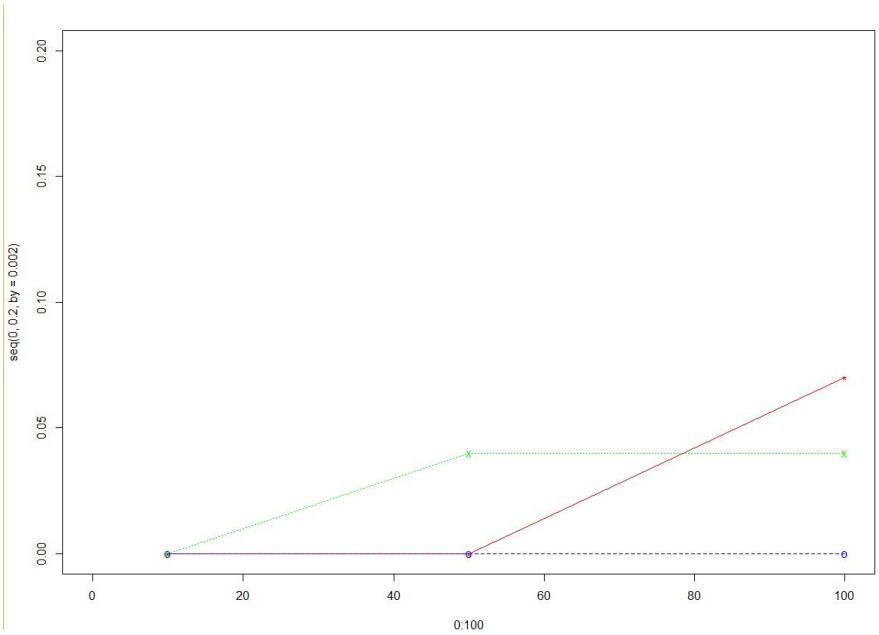
Crearemos una gráfica de tiempos comparando las diferentes funciones ordenar, bubble_sort, e insertar.

Cambiaremos los tamaños de los vectores a ordenar para así poder comparar los tiempos entre ambos algoritmos y saber cuál es más eficiente.

Con el siguiente código podemos dibujar una grafica para vectores de tamaño 10, 50, y 100. Primero generamos un vector de dicho tamaño con números aleatorios, calculamos su ordenación y medimos su tiempo. Vamos dibujando los tiempos en función del tamaño del vector para ambos algoritmos de ordenación, para poder ir viendo la evolución y diferencia entre ambos.

```
############ Gráfica de tiempos:
# Crearemos una gráfica comparando los códigos con vectores de distintos tamaños:

# Para 10 elementos:
vec=round(runif(10,0,100))
bubble_sort(vec)
t<-system.time(bubble_sort(vec))
ordenar(vec)
t2<-system.time(ordenar(vec))
insertar(vec)
t4<-system.time(insertar(vec))

plot(0:100,seq(0,0.2,by=0.002) , col="white")
points(10, t2[1] , col="blue", pch="o" )

points(10, t4[1], col="green", pch="x")

points(10, t[1], col="red", pch="*")

# Para 50 elementos:
vec=round(runif(50,0,100))
bubble_sort(vec)
t1<-system.time(bubble_sort(vec))
ordenar(vec)
t3<-system.time(ordenar(vec))
insertar(vec)
t5<-system.time(insertar(vec))

points(50, t5[1], col="green", pch="x")
x=c(10,50)
y=c(t4[1],t5[1])
lines(x, y, col="green",lty=3)

points(50, t1[1], col="red", pch="*")
x=c(10,50)
y=c(t[1],t1[1])
lines(x, y, col="red",lty=1)

points(50, t3[1], col="blue", pch="o")
x=c(10,50)
y=c(t2[1],t3[1])
lines(x, y, col="blue",lty=2)

# Para 100 elementos:
vec=round(runif(100,0,100))
bubble_sort(vec)
t<-system.time(bubble_sort(vec))
ordenar(vec)
t2<-system.time(ordenar(vec))
insertar(vec)
t4<-system.time(insertar(vec))

points(100, t4[1], col="green", pch="x")
x=c(50,100)
y=c(t5[1],t4[1])
lines(x, y, col="green",lty=3)

points(100, t[1], col="red", pch="*")
x=c(50,100)
y=c(t1[1],t[1])
lines(x, y, col="red",lty=1)

points(100, t2[1], col="blue", pch="o")
x=c(50,100)
y=c(t3[1],t2[1])
lines(x, y, col="blue",lty=2)

# El de clase funciona más lento que el nuestro porque el suyo tiene más comparaciones
```

Donde nos muestra la gráfica:



Red: bubble_sort | Blue: ordenar | Green: insertar

Podemos observar como el algoritmo de la función bubble_sort (rojo) es más lento para un mayor conjunto de números, el segundo más lento es el algoritmo de la función insertar (verde), y el código más rápido es el código de la función ordenar (azul) creado por nosotros.