Universidad Rey Juan Carlos

Grado en Matemáticas

Algoritmos de ordenación II

Geometría computacional

Autores: Guillermo Grande Santi y

Alejandro López Adrados

Marzo, 2022

Índice

1 Objetivos

El objetivo de esta práctica era la implementación de otros dos algoritmos de ordenación, que en este caso son algoritmos rápidos. Posteriormente, habrá que comparar estos dos algoritmos con los tres algoritmos lentos de la práctica anterior y con el algoritmo Mergesort, implementado en clase. Para esta comparación, hemos calculado el tiempo que tardaban en ejecutarse con una cantidad distinta de elementos al algoritmo.

2 Metodología

2.1 Algoritmo Quicksort

Para este algoritmo seleccionamos en primer lugar la mediana de los valores a ordenar. Tras esto, se divide el array de valores en dos partes generando parte izquierda y parte derecha. Utilizando la recursividad se llama a la función con la parte izquierda y con la parte derecha. Una vez que se completan todos los pasos recursivos se concatenan de la forma parte izquierda-mediana-parte derecha. Devolviendo así todos los valores ordenados tras numerosos pasos recursivos.

2.2 Algoritmo Heapsort

Para este algoritmo empezamos con el vector inicial, intercambiando los valores desde la posición 1 hasta la n de modo que el mayor valor queda en la última posición. Una vez tenemos esto, se realizará de nuevo con el vector de longitud n-1 (método "heapsort"). Una vez hemos llegado a la mediana, la primera posición debe ser menor igual que los valores de sus hijos. Si no lo es, tiene que ser uno de sus hijos, por lo que se selecciona el menor y se intercambia por la primera posición. Repitiendo este proceso sucesivamente obtendremos el vector ordenado, llegando a la condición final de que una posición anterior tiene que ser menor que la siguiente posición.

2.3 Cálculo de tiempos y gráfico

Para calcular los tiempos hemos usado la función "proc.time()", que determina el tiempo real y de la CPU en segundos desde que se inició el proceso en R. Por tanto, por cada vez que se llama al algoritmo se coge el tiempo justo antes de llamarlo, para así restar al tiempo total de ejecución el tiempo que tardó en llegar hasta justo antes de la llamada al algoritmo.

En cuánto a los gráficos, se cogen como variable independiente un vector formado por los diferentes números de elementos que se usarán en el algoritmo. Por otro lado, como variable dependiente se crean seis vectores (uno por cada algoritmo) en los que se guardarán los tiempos de ejecución de cada uno de los algoritmos con el número de elementos que indica la variable independiente. Una vez tenemos estos datos, se usan las funciones "plot()" y "lines()", que se encargarán de dibujar la gráfica. Además se incluirá una leyenda con la función "legend()". En el primer gráfico podemos observar la comparación que se pedía pero con un número pequeño de elementos para que la comparación pueda ser notable. En el segundo gráfico podemos ver la comparación que se pedía con un mayor número de elementos.

3 Conclusiones

Como conclusión, observando los gráficos podemos ver que los algoritmos lentos se quedan muy atrás en cuanto a rendimiento comparándose con Quicksort y Mergesort. El heapsort, aunque sea denominado como algoritmo rápido, no es muy eficiente en R. El primer gráfico nos muestra cómo para valores muy pequeños los algoritmos están bastante igualados, aunque se empiezan a distanciar poco a poco. Especialmente la burbuja, que como vimos en la anterior práctica, con muy pocos elementos es la más ineficiente. El segundo gráfico se ha realizado con valores que suponen un tiempo muy corto para los rápidos y que empiezan a suponer un problema para los lentos. En efecto, con un número de elementos mayor a 50.000, los algoritmos lentos se ralentizaban demasiado

y el programa de R colpasaba. En cuanto a los rápidos, exceptuando el heapsort, sus tiempos son todos muy parecidos, por no decir iguales, para cualquier número de elementos, llegando a ordenar 1.000.000 de elementos en menos de 10 segundos, lo cual resulta increíble. Es por ello que en este gráfico anexado se superponen las dos líneas que corresponden a estos dos algoritmos de ordenación rápidos.

4 Anexos

4.1 Código de algoritmos

```
quicksort<-function(x)
{
  #Caso base
  if(length(x)<=1){
    return(x)
  }
  #Inicializamos un array que guardará todos los valores menos
  #la mediana
  resto<-c()
  #Elegimos la mediana del vector
  mediana<-length(x)%/%2
  #Se elige el pivote
  pivote<-x[mediana]
  #Bucle que inserta en resto hasta la posición mediana-1
  for(i in 1:mediana-1){
    resto[i]<-x[i]
  }
  longitud<-length(x)
  siguiente_mediana<-mediana+1
```

```
  #Bucle que inserta en resto desde la posición mediana+1
  for(i in siguiente_mediana:longitud){
    resto[i-1]<-x[i]
  }
  #Recursividad sobre el vector resto:
    #Parte izquierda del Árbol (menores)
    L1<-quicksort(resto[resto<pivote])
    #Parte derecha del Árbol (mayores)
    L3<-quicksort(resto[resto>=pivote])
  #Se devuelve la concatenación de parte
  #izquierda-pivote-parte derecha
  return(c(L1,pivote,L3))
}
#-------------------------------------------------------------
heapify<-function(vec, n, i)
{
  #Seleccionamos al padre y a los dos hijos
  padre<-i
  hijo_izq<-2*(i-1) + 1
  hijo_der<-2*(i-1) + 2
  if ((hijo_izq < n) & (vec[padre] < vec[hijo_izq]))
  {
    padre<-hijo_izq
  }
  if ((hijo_der < n) & (vec[padre] < vec[hijo_der]))
  {
    padre<-hijo_der
  }
  if (padre != i) {
    vec<-replace(vec, c(i, padre), vec[c(padre, i)])
```

```
    vec<-heapify(vec, n, padre)
  }
  vec
}

heapsort<-function(vec)
{
  n<-length(vec)
  #Seleccionamos la mediana
  mediana<-n%/%2
  for (i in mediana:1) {
    vec<-heapify(vec, n, i)
  }
  for (i in n:1) {
    vec<-replace(vec, c(i, 1), vec[c(1, i)])
    vec<-heapify(vec, i, 1)
  }
  vec
}
#-----------------------------------------------------------------
merge <- function(izq, der) {
  resultado <- numeric(length(izq) + length(der))
  i <- 1 # Contador de la parte izqierda
  j <- 1 # Contador de la parte derecha
  r <- 1 # Contador del resultado
  for(r in 1 : length(resultado)) {
    if((i <= length(izq) && izq[i] < der[j]) || j > length(der)) {
      resultado[r] <- izq[i]
      i <- i + 1
    } else {
```

```
      resultado[r] <- der[j]
      j <- j + 1
    }
  }
  resultado
}

mergesort <- function(l) {
  if(length(l) > 1) {
    mediana <- ceiling(length(l) / 2)
    izq <- mergesort(l[1 : mediana])
    der <- mergesort(l[(mediana + 1) : length(l)])
    merge(izq, der)
  } else {
    l
  }
}

#Vector de números aleatorios y llamada a los algoritmos
#Variable auxiliar
aux<-0
#Para cambiar el número de elementos se debe modificar la
#siguiente variable
elementos<-40
vec<-runif(elementos,0,1000)

#Cálculo de los tiempos que tarda cada uno de los algoritmos
tiempoQuick<-proc.time()
quicksort(vec)

##  [1]  25.83859  32.18012  33.11035  77.22245 111.38119 144.15091 146.25904
```

```
##  [8] 180.04342 200.99753 217.38872 231.34620 265.73190 310.67947 319.82866
## [15] 374.70832 379.59289 420.26439 441.54701 456.81126 499.95117 526.37655
## [22] 530.60281 562.20467 576.80274 582.60123 661.06784 679.22343 699.50651
## [29] 704.18795 707.84543 726.05267 742.65987 759.03601 787.63228 815.20226
## [36] 826.56881 840.63756 846.85121 859.43740 960.70712
```

```
proc.time()-tiempoQuick
```

```
##    user  system elapsed
##   0.049   0.005   0.055
```

```
tiempoHeap<-proc.time()
heapsort(vec)
```

```
##  [1]  25.83859  32.18012  33.11035  77.22245 111.38119 144.15091 146.25904
##  [8] 180.04342 200.99753 217.38872 231.34620 265.73190 310.67947 319.82866
## [15] 374.70832 379.59289 420.26439 441.54701 456.81126 499.95117 526.37655
## [22] 530.60281 562.20467 576.80274 582.60123 661.06784 679.22343 699.50651
## [29] 704.18795 707.84543 726.05267 742.65987 759.03601 787.63228 815.20226
## [36] 826.56881 840.63756 846.85121 859.43740 960.70712
```

```
proc.time()-tiempoHeap
```

```
##    user  system elapsed
##   0.034   0.000   0.034
```

```
tiempoMerge<-proc.time()
mergesort(vec)
```

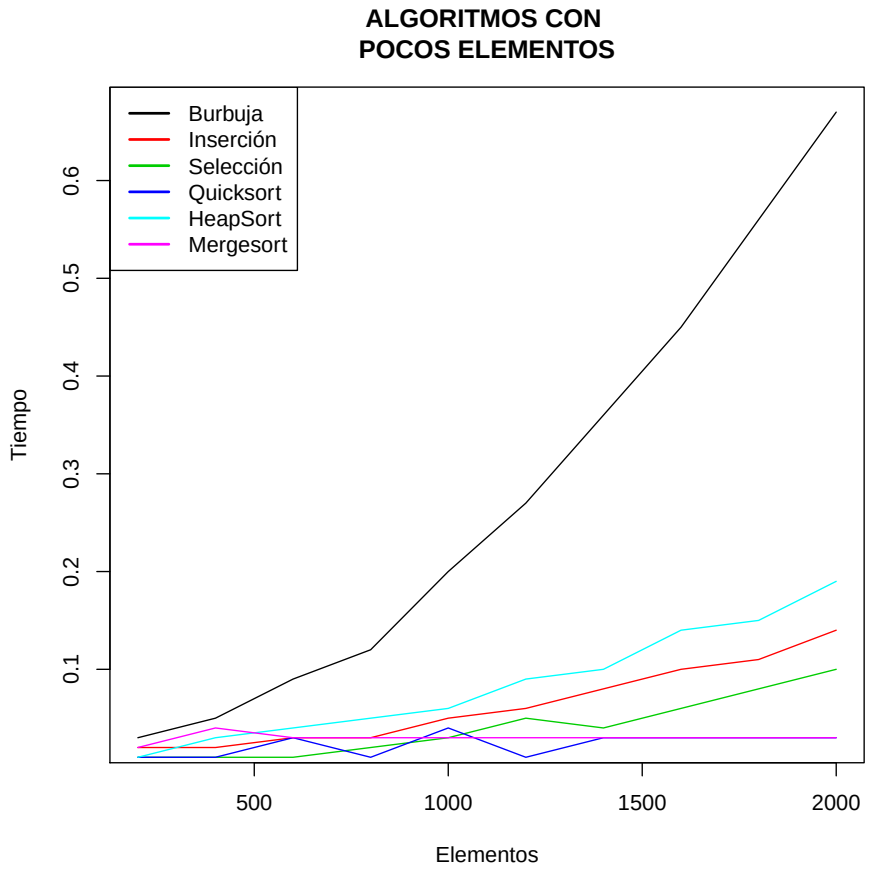
```
##  [1]  25.83859  32.18012  33.11035  77.22245 111.38119 144.15091 146.25904
##  [8] 180.04342 200.99753 217.38872 231.34620 265.73190 310.67947 319.82866
## [15] 374.70832 379.59289 420.26439 441.54701 456.81126 499.95117 526.37655
## [22] 530.60281 562.20467 576.80274 582.60123 661.06784 679.22343 699.50651
## [29] 704.18795 707.84543 726.05267 742.65987 759.03601 787.63228 815.20226
## [36] 826.56881 840.63756 846.85121 859.43740 960.70712
```

```
proc.time()-tiempoMerge

##    user  system elapsed
##   0.024   0.000   0.024
```

4.2 Representación de las gráficas

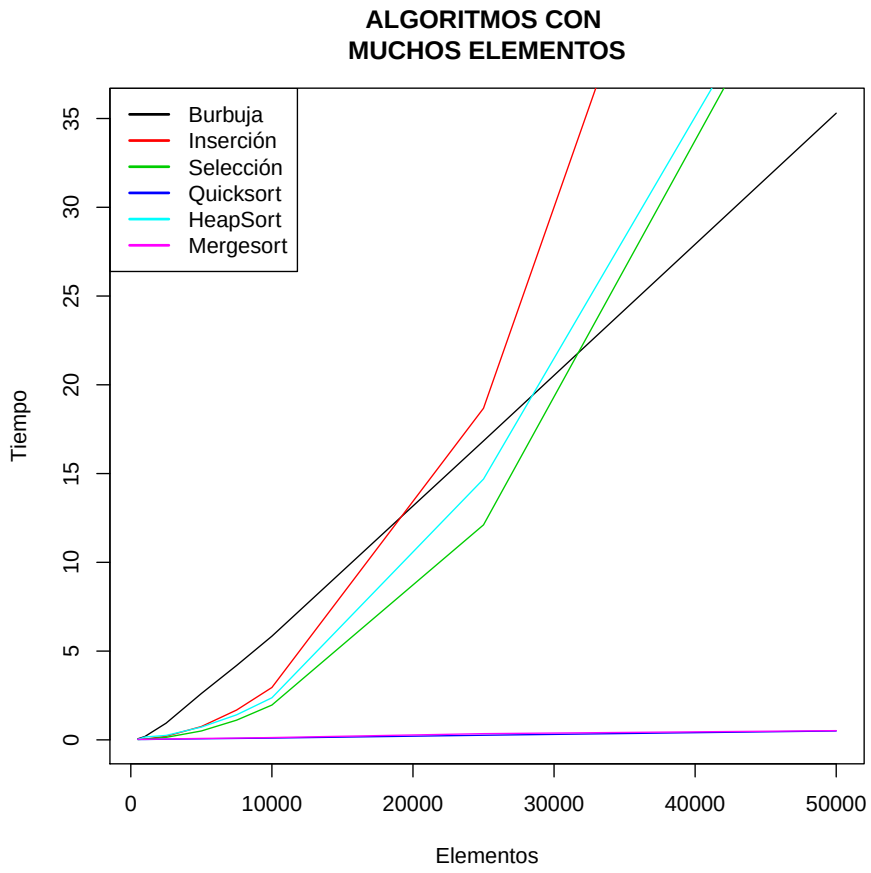
```
# Datos
Elementos<-c(200,400,600,800,1000,1200,1400,1600,1800,2000)
#Burbuja
Tiempo<-c(0.03,0.05,0.09,0.12,0.2,0.27,0.36,0.45,0.56,0.67)
#Inserción
y2<-c(0.02,0.02,0.03,0.03,0.05,0.06,0.08,0.1,0.11,0.14)
#Selección
y3<-c(0.01,0.01,0.01,0.02,0.03,0.05,0.04,0.06,0.08,0.1)
#Quicksort
y4<-c(0.01,0.01,0.03,0.01,0.04,0.01,0.03,0.03,0.03,0.03)
#Heapsort
y5<-c(0.01,0.03,0.04,0.05,0.06,0.09,0.1,0.14,0.15,0.19)
#Mergesort
y6<-c(0.02,0.04,0.03,0.03,0.03,0.03,0.03,0.03,0.03,0.03)

# Gráfico
plot(Elementos, Tiempo, type = "l", main="ALGORITMOS CON
POCOS ELEMENTOS")
legend(x = "topleft",          # Posición
       legend = c("Burbuja", "Inserción","Selección","Quicksort",
       "HeapSort","Mergesort"), # Textos de la leyenda
       lty = c(1),           # Tipo de líneas
       col = c(1, 2, 3, 4, 5, 6),          # Colores de las líneas
```

```
       lwd = 2)                  # Ancho de las líneas
lines(Elementos,y2, type ="l", col=2)
lines(Elementos,y3, type ="l", col=3)
lines(Elementos,y4, type ="l", col=4)
lines(Elementos,y5, type ="l", col=5)
lines(Elementos,y6, type ="l", col=6)
```




```
# Datos
Elementos<-c(500,1000,2500,5000,7500,10000,25000,50000)
#Burbuja
Tiempo<-c(0.06,0.19,0.94,2.61,4.19,5.84,16.84,35.3)
```

```
#Inserción
y2<-c(0.03,0.04,0.2,0.75,1.68,2.95,18.68,75.32)
#Selección
y3<-c(0.02,0.04,0.14,0.5,1.11,1.96,12.11,48.2)
#Quicksort
y4<-c(0.02,0.02,0.04,0.06,0.08,0.1,0.26,0.5)
#Heapsort
y5<-c(0.03,0.16,0.25,0.72,1.41,2.37,14.69,48.7)
#Mergesort
y6<-c(0.03,0.03,0.05,0.08,0.1,0.13,0.35,0.52)

# Gráfico
plot(Elementos, Tiempo, type = "l", main="ALGORITMOS CON
MUCHOS ELEMENTOS")
legend(x = "topleft",          # Posición
       legend = c("Burbuja", "Inserción","Selección","Quicksort",
       "HeapSort","Mergesort"), # Textos de la leyenda
       lty = c(1),           # Tipo de líneas
       col = c(1, 2, 3, 4, 5, 6),          # Colores de las líneas
       lwd = 2)                # Ancho de las líneas
lines(Elementos,y2, type ="l", col=2)
lines(Elementos,y3, type ="l", col=3)
lines(Elementos,y4, type ="l", col=4)
lines(Elementos,y5, type ="l", col=5)
lines(Elementos,y6, type ="l", col=6)
```


ALGORITMOS CON
MUCHOS ELEMENTOS
Burbuja
Inserción
Selección
Quicksort
HeapSort
Mergesort
Tiempo
0
5
10
15
20
25
30
35
0
10000
20000
30000
40000
50000
Elementos