

Algoritmos de Búsqueda y Ordenamiento

Sergio Guzman Camacho | 2-17-1817



Descripción del Proyecto

- Se debe de crear una aplicación donde se ejecuten los diferentes algoritmos de búsqueda que hemos estado investigando en el transcurso del ciclo para ordenar un arreglo único. Se deben ejecutar todos los algoritmos seleccionados al mismo tiempo y se debe calcular el tiempo que tomó cada algoritmo para resolver el problema y apuntar el que menor tiempo duró en realizar la tarea.



Objetivos

- **Objetivo General**

- + Interactuar con los distintos tipos de algoritmos de búsquedas que se han explicado en el transcurso del ciclo y observar su funcionamiento y tiempo de reacción. A la vez que se intenta desarrollar maneras de ordenar arreglos desordenados de manera óptima y en el menor tiempo posible.

- **Objetivo Especificos**

- Programar algoritmos de búsqueda capaces de ordenar arreglos desordenados.
- Cronometrar el tiempo de ejecución de los algoritmos desarrollados.
- Identificar el algoritmo más rápido a la hora de realizar el ordenamiento de arreglos.
- Explicar el por qué el algoritmo más rápido fue el más rápido.



Algoritmos Paralelos

- Un algoritmo paralelo se refiere a un algoritmo que puede ser ejecutado por distintas unidades de procesamiento en un mismo instante de tiempo, para al final unir todas las partes y obtener el resultado correcto.



Técnicas Algorítmicas Paralelas

- ✓ **Técnica List Ranking:** Consiste en indicar la posición de cada elemento de una lista. Por ejemplo, asignarle el número 1 al primer elemento de una lista.
- ✓ **Técnica de Euler:** Se utiliza para resolver ecuaciones diferenciales ordinarias (EDO) dado un valor inicial. Este método sirve para construir métodos más complejos.
- ✓ **Contracción de Árboles:** Este mantiene la estructura de un árbol demasiado grande y dedica sus recursos a asignar una predicción distinta a la frecuencia promedio del nodo.



Modelos de Algoritmos Paralelos

- Un modelo de programación paralela es un modelo para escribir programas paralelos los cuales pueden ser compilados y ejecutados.
- Algunos de los ejemplos de modelos de programación paralela son:
 - ✓ Esqueletos algorítmicos
 - ✓ Componentes
 - ✓ Objetos distribuidos
 - ✓ Invocación de Método de manera remota
 - ✓ Workflows
 - ✓ Máquina de Acceso paralelo Aleatorio
 - ✓ Procesamiento de flujo
 - ✓ Bulk synchronous parallelism



Busqueda Secuencial

- Es el algoritmo de búsqueda más simple, menos eficiente y que menos precondiciones requiere. Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

```
100 go func() {  
101  
102     var a [100]int  
103     var dato int  
104     defer MedirTiempo(time.Now(), "Busqueda Secuencial")  
105  
106     LlenarArrayOrdenado(&a)  
107  
108     dato = 58 //Dato a buscar  
109  
110     i := 0  
111  
112     for (a[i] != dato) && (i < 100) {  
113  
114         i++  
115  
116         if a[i] == dato {  
117  
118             fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, i)  
119  
120         } else {  
121  
122             //fmt.Printf("\nEl Elemento no esta en el array")  
123  
124         }  
125     }  
126 }()
```



```

128 go func() {
129
130     var a [100]int
131     LlenarArrayOrdenado(&a)
132     defer MedirTiempo(time.Now(), "Busqueda Binaria")
133
134     ini := 0
135     fin := 100 - 1
136     var mitad, dato int
137
138     dato = 58 //Dato a Buscar
139
140     mitad = (ini + fin) / 2
141
142     for (ini <= fin) && (a[mitad] != dato) {
143
144         if dato < a[mitad] {
145             fin = mitad - 1
146         } else {
147
148             ini = mitad + 1
149
150         }
151
152         mitad = (ini + fin) / 2
153     }
154
155     if dato == a[mitad] {
156
157         fmt.Printf("\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, mitad)
158
159     } else {
160
161         //fmt.Printf("\nEl Elemento no esta en el array")
162
163     }
164
165 }()
166

```

Busqueda Binaria

- Este algoritmo compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior, respectivamente, al elemento central, y así sucesivamente.



Ordenamiento de Burbuja

```
168 go func() {
169
170     var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}
171     var n int = len(a)
172     defer MedirTiempo(time.Now(), "Ordenamiento de Burbuja")
173     var e, i, auxiliar int
174
175     for e = 0; e < n; e++ {
176
177         for i = 0; i < n-1-e; i++ {
178
179             if a[i] > a[i+1] {
180
181                 auxiliar = a[i+1]
182                 a[i+1] = a[i]
183                 a[i] = auxiliar
184             }
185         }
186     }
187
188     fmt.Println(a)
189 }()
```

- Se basa en recorrer el array un cierto número de veces, comparando pares de valores que ocupan posiciones adyacentes (0-1, 1-2, ...). Si ambos datos no están ordenados, se intercambian.



```

192 go func() {
193
194     var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}
195     var n int = len(a)
196     defer MedirTiempo(time.Now(), "QuickSort")
197
198     b := quicksort(a, 0, n-1)
199     fmt.Println(b)
200
201 }()

```

```

42 func quicksort(a []int, izq int, der int) []int {
43
44     pivote := a[izq]
45     i := izq
46     j := der
47     var aux int
48
49     for i < j {
50
51         for a[i] <= pivote && i < j {
52             i++
53         }
54
55         for a[j] > pivote {
56             j--
57         }
58
59         if i < j {
60             aux = a[i]
61             a[i] = a[j]
62             a[j] = aux
63         }
64     }
65
66     a[izq] = a[j]
67     a[j] = pivote
68
69     if izq < j-1 {
70         quicksort(a, izq, j-1)
71     }
72
73     if j+1 < der {
74         quicksort(a, j+1, der)
75     }
76
77     return a
78 }

```

QuickSort

- El método se basa en la estrategia típica de "divide y vencerás". El array a ordenar se divide en dos partes: una contendrá todos los valores menores o iguales a un cierto valor (que se suele denominar pivote) y la otra con los valores mayores que dicho valor.



- El primer elemento del array (CB [0]) se considerado ordenado (la lista inicial consta de un elemento). A continuación, se inserta el segundo elemento (CB [1]) en la posición correcta (delante o detrás de CB [0]) dependiendo de que sea menor o mayor que CB [0].

Metodo de Insercion

```
04  
05     var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}  
06     var n int = len(a)  
07     defer MedirTiempo(time.Now(), "Insercion")  
08     var auxiliar int  
09  
10     for i := 1; i < n; i++ {  
11  
12         auxiliar = a[i]  
13  
14         for j := i - 1; j >= 0 && a[j] > auxiliar; j-- {  
15  
16             a[j+1] = a[j]  
17             a[j] = auxiliar  
18         }  
19     }  
20  
21     fmt.Println(a)
```



```

PS C:\Users\sergi\OneDrive\Universidad\Clases Virtuales\5. Ciclo 2-2021\5. Algoritmos Paralelos\3. Tercer Parcial> go run .\AlgoritmoFinal.go

Arreglo Desordenado -----
[15 3 8 6 18 1 20 10 37 150]

Arreglo Ordenado -----
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100]

[1 3 6 8 10 15 18 20 37 150]
[1 3 6 8 10 15 18 20 37 150]
Algoritmo: Insercion | Tiempo de ejecucion: 444.6µs

Algoritmo: QuickSort | Tiempo de ejecucion: 444.6µs

[1 3 6 8 10 15 18 20 37 150]
Algoritmo: Ordenamiento de Burbuja | Tiempo de ejecucion: 1.5117ms

```

```

PS C:\Users\sergi\OneDrive\Universidad\Clases Virtuales\5. Ciclo 2-2021\5. Algoritmos Paralelos\3. Tercer Parcial> go run .\AlgoritmoFinal.go

Arreglo Desordenado -----
[15 3 8 6 18 1 20 10 37 150]

Arreglo Ordenado -----
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100]

El dato 58, fue encontrado en la posicion: 57 del arreglo!!
Algoritmo: Busqueda Secuencial | Tiempo de ejecucion: 623.9µs

El dato 58, fue encontrado en la posicion: 57 del arreglo!!
Algoritmo: Busqueda Binaria | Tiempo de ejecucion: 120.8µs

```

Conclusiones

- En esta prueba, nos dimos cuenta como una buena selección de pivotes para los algoritmos de ordenamientos y búsqueda (para los algoritmos que lo utilicen), pueden hacer la diferencia a la hora de optimizar estos algoritmos para sacarle mayor provecho a los mismos. Si no tomamos en cuenta esta selección de pivotes (que muchos los hacen), nos daremos cuenta de que, nuestros algoritmos como QuickSort o búsqueda binaria, resultaran menos eficiente que los algoritmos tradicionales como ordenamiento de burbuja o búsqueda secuencial.

