



Tarea 2 Investigación Quick Sort

Asignatura: Estructura de Datos y Algoritmia

Profesor: D. Sc. Gerardo García Gil

2021-B

Alumno: José Rafael Ruiz Gudiño

Ingeniería en Desarrollo de Software

Centro de Enseñanza Técnica Industrial (CETI)

1.-Introducción

Desde que existe la ciencia de la computación, uno de los mayores problemas con los que los ingenieros se encontraban en su día a día, era el de ordenar listas de elementos. Por su causa, diversos algoritmos de ordenación fueron desarrollados a lo largo de los años y siempre existió un intenso debate entre los desarrolladores sobre cuál de todos los algoritmos de ordenación era el más rápido. Pero ¿qué es un método de ordenamiento?

Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo con un criterio de ordenamiento. El ordenamiento se efectúa con base en el valor de algún campo en un registro. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado. El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

El debate mencionado anteriormente finalizó abruptamente en 1960 cuando Sir Charles Antony Richard Hoare, nativo de Sri Lanka y ganador del premio Turing en 1980, desarrolló el algoritmo de ordenación Quick Sort casi por casualidad mientras ideaba la forma de facilitar la búsqueda de palabras en el diccionario.

2.- Desarrollo

¿Qué es el método de ordenamiento Quick Sort?

El ordenamiento por partición (Quick Sort) se puede definir en una forma más conveniente como un procedimiento recursivo. Tiene aparentemente la propiedad de trabajar mejor para elementos de entrada desordenados completamente, que para elementos semiordenados. Esta situación es precisamente la opuesta al ordenamiento de burbuja.

El algoritmo QuickSort se basa en la técnica de "divide y vencerás" por la que en cada recursión, el problema se divide en subproblemas de menor tamaño y se resuelven por separado (aplicando la misma técnica) para ser unidos de nuevo una vez resueltos.

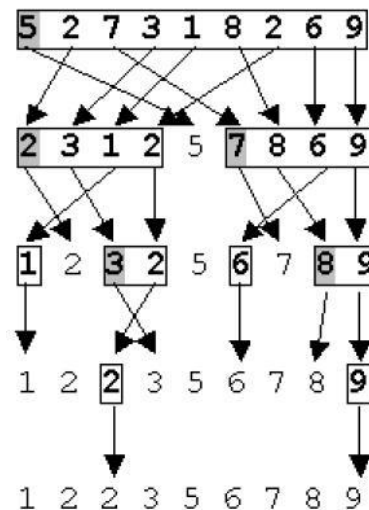
Características

En la práctica, es el algoritmo de ordenación más rápido conocido, su tiempo de ejecución promedio es $O(n \log(n))$, siendo en el peor de los casos $O(n^2)$, caso altamente improbable. El hecho de que sea más

rápido que otros algoritmos de ordenación con tiempo promedio de $O(n \log(n))$ (como SmoothSort o HeapSort) viene dado por que QuickSort realiza menos operaciones ya que el método utilizado es el de partición.

Funcionamiento

1. Se elige un elemento v de la lista L de elementos al que se le llama pivote.
2. Se particiona la lista L en tres listas:
 - a. $L1$ - que contiene todos los elementos de L menos v que sean menores o iguales que v
 - b. $L2$ - que contiene a v
 - c. $L3$ - que contiene todos los elementos de L menos v que sean mayores o iguales que v
3. Se aplica la recursión sobre $L1$ y $L3$
4. Se unen todas las soluciones que darán forma final a la lista L finalmente ordenada. Como $L1$ y $L3$ están ya ordenados, lo único que tenemos que hacer es concatenar $L1$, $L2$ y $L3$



Aunque este algoritmo parece sencillo, hay que implementar los pasos 1 y 3 de forma que se favorezca la velocidad de ejecución del algoritmo.

La velocidad de ejecución del algoritmo depende en gran parte de cómo se implementa este mecanismo, debido a que una mala implementación puede suponer que el algoritmo se ejecute a una velocidad mediocre. La elección del pivote determina las particiones de la lista de datos, por lo tanto, sobra decir que esta es la parte más crítica de la implementación del algoritmo Quick Sort. Es importante intentar que al seleccionar el pivote v las particiones $L1$ y $L3$ tengan un tamaño idéntico dentro de lo posible.

3.- Implementación

El fin de esta implementación de QuickSort es la de crear un algoritmo de ordenación eficiente y rápido, por lo que las listas "auxiliares" que creamos al particionar no son listas reales, es decir, no creamos nuevos elementos de lista para albergar los elementos, sino que situamos el pivote en una posición determinada dentro de la lista para simular las particiones. La mejor opción a la hora de crear las listas que contendrán a $L1$ y $L3$ es reordenar los elementos de forma que los elementos que aparecen antes del pivote sean menores o iguales a él, y los que aparecen después sean mayores o iguales.

Implementación en C

Para implementar este algoritmo debemos tener una función que intercambie valores. No es necesaria, pero ahorra tres líneas de código además de que aumenta la legibilidad.

Después tenemos la función “quicksort” que recibe el arreglo y los índices inicio y fin, o izquierda y derecha. En este caso se usará recursividad así que se divide el arreglo y se invoca a “quicksort” 2 veces.

El algoritmo irá dividiendo el arreglo, pero usará el índice que devuelva el método “partición”; este método es el que realmente ordena el arreglo e intercambia valores.

Código

```
#include<stdio.h>
#include<stdlib.h>
void quicksort(int arreglo[], int izquierda, int derecha);
int particion(int arreglo[], int izquierda, int derecha);
void intercambiar(int *a, int *b);
int main(void) {
    // El arreglo
    int arreglo[] = {28, 11, 96, -5, 21, 18, 17, 12, 7, 22, 30, 97, -3, -7, -13};
    int longitud = sizeof arreglo / sizeof arreglo[0];
    printf("Arreglo desordenado:\n");
    for (int x = 0; x < longitud; x++) {
        printf("%d ", arreglo[x]);
    }
    printf("\n");
    //Invocar a quicksort indicando el arreglo, desde la posicion 0 a n-1
    quicksort(arreglo, 0, longitud - 1);
    printf("Arreglo Ordenado:\n");
    for (int x = 0; x < longitud; x++)
        printf("%d ", arreglo[x]);
    return 0;
}
void intercambiar(int *a, int *b) {
    int temporal = *a;
    *a = *b;
    *b = temporal;
}
// Técnica divide y vencerás
void quicksort(int arreglo[], int izquierda, int derecha) {
    if (izquierda < derecha) {
        int indiceParticion = particion(arreglo, izquierda, derecha);
        quicksort(arreglo, izquierda, indiceParticion);
        quicksort(arreglo, indiceParticion + 1, derecha);
    }
}
int particion(int arreglo[], int izquierda, int derecha) {
    // Elegimos el pivote, es el primero
    int pivote = arreglo[izquierda];
    // Ciclo infinito
    while (1) {
        // Mientras cada elemento desde la izquierda esté en orden (sea menor que el
        // pivote) continúa avanzando el índice
```

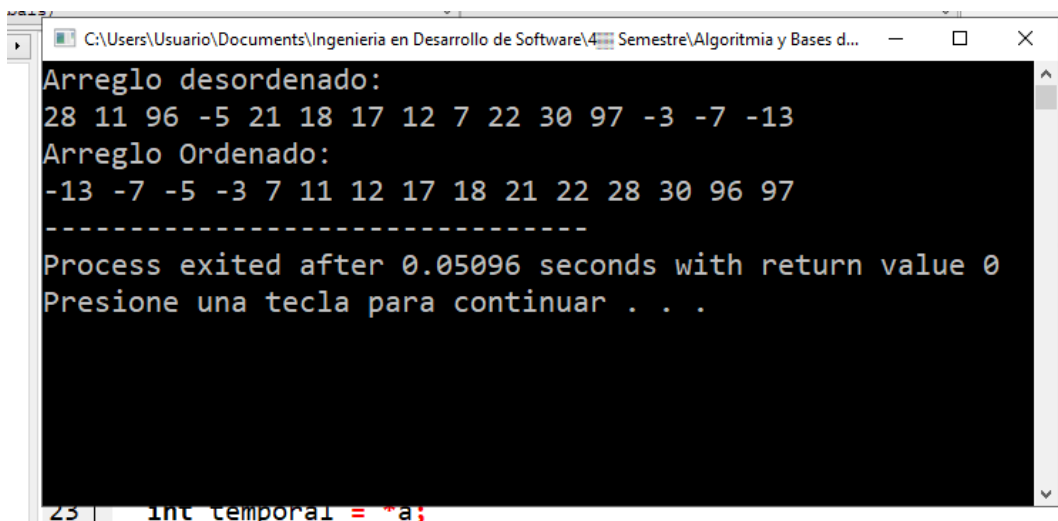
```

while (arreglo[izquierda] < pivote) {
    izquierda++;
}
// Mientras cada elemento desde la derecha esté en orden (sea mayor que el
// pivote) continúa disminuyendo el índice
while (arreglo[derecha] > pivote) {
    derecha--;
}
/*Si la izquierda es mayor o igual que la derecha significa que no
necesitamos hacer ningún intercambio de variables, pues los elementos ya están
en orden (al menos en esta iteración)*/
if (izquierda >= derecha) {
    // Indicar en dónde nos quedamos para poder dividir el arreglo de nuevo y ordenar los demás elementos
    return derecha;
} else {
    /*Si las variables quedaron "lejos" (es decir, la izquierda no superó ni
    alcanzó a la derecha) significa que se detuvieron porque encontraron un valor que no estaba
    en orden, así que lo intercambiamos */
    intercambiar(&arreglo[izquierda], &arreglo[derecha]);
    //Después de intercambiar se continúa avanzando los índices
    izquierda++;
    derecha--;
} // El while se repite hasta que izquierda >= derecha
}
}

```

4.- Resultados

Finalmente se tiene el arreglo de una manera ordenada:



```

Arreglo desordenado:
28 11 96 -5 21 18 17 12 7 22 30 97 -3 -7 -13
Arreglo Ordenado:
-13 -7 -5 -3 7 11 12 17 18 21 22 28 30 96 97
-----
Process exited after 0.05096 seconds with return value 0
Presione una tecla para continuar . . .

```

5.- Conclusiones

El método quick sort es un método muy interesante debido a que es muy eficiente al implementarlo, así como también agiliza los tiempos de carga y realiza el ordenamiento de una manera mas óptima. Este método es mas eficiente que el método burbuja, debido a que el método burbuja es efectivo, pero hace más cálculos de lo que se necesitan hasta que se finalicen los ciclos, por lo que termina repitiendo cálculos que ya se habían hecho e innecesarios. Y esto es lo que caracteriza y diferencia al método quick sort ya que el arreglo lo divide en subarreglos y los va dividiendo hasta ordenarlos por lo que solo se realizan los

cálculos necesarios, optimizando así tiempos de carga. Aunque al principio puede ser un poco complejo de leer el código y su función, una vez que se entiende cómo funciona el método es mas sencillo de lo que parece.

6.-Referencias

Campos, O. (2011, 14 junio). *Implementando el algoritmo QuickSort*. Genbeta.

<https://www.genbeta.com/desarrollo/implementando-el-algoritmo-quicksort>

P. (2019, 9 octubre). *Quicksort en C: implementación de algoritmo*. Parzibyte's blog.

<https://parzibyte.me/blog/2019/10/08/quicksort-c-algoritmo/>

Universidad Autónoma del Estado de Hidalgo. (s. f.). *Método Quick Sort*. Centro de Innovación para el

Desarrollo y la Capacitación en Materiales Educativos. Recuperado 31 de agosto de 2021, de

http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro9/mtodo_quick_sort.html