Algoritmos de ordenamiento Internos

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo. Este informe nos permitirá conocer más a fondo cada método distinto de ordenamiento, desde uno simple hasta el más complejo. Se realizarán comparaciones en tiempo de ejecución, prerrequisitos de cada algoritmo, funcionalidad, alcance, etc.

Tipos de algoritmos

Para poder ordenar una cantidad determinada de números almacenadas en un vector o matriz, existen distintos métodos (algoritmos) con distintas características y complejidad. Existe desde el método más simple, como el Bubblesort (o Método Burbuja), que son simples iteraciones, hasta el Quicksort (Método Rápido), que al estar optimizado usando recursión, su tiempo de ejecución es menor y es más efectivo.

Métodos Internos

BubbleSort

El método de la burbuja es uno de los más simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.

Por ejemplo, imaginemos que tenemos los siguientes valores:

5	6	1	0	3

Lo que haría una burbuja simple, seria comenzar recorriendo los valores de izq. a derecha, comenzando por el 5. Lo compara con el 6, con el 1, con el 0 y con el 3, si es mayor o menor (dependiendo si el orden es ascendiente o

descendiente) se intercambian de posición. Luego continua con el siguiente, con el 6, y lo compara con todos los elementos de la lista, esperando ver si se cumple o no la misma condición que con el primer elemento. Así, sucesivamente, hasta el último elemento de la lista.

Burbuja simple:

Como lo describimos en el ítem anterior, la burbuja más simple de todas es la que compara todos con todos, generando comparaciones extras, por ejemplo, no tiene sentido que se compare con sigo mismo o que se compare con los valores anteriores a él, ya que supuestamente, ya están ordenados.

```
for (i=1; i<LIMITE; i++)
    for j=0 ; j<LIMITE - 1; j++)
          if (vector[j] > vector[j+1])
               temp = vector[j];
               vector[j] = vector[j+1];
               vector[j+1] = temp;
```

Burbuja Mejorada:

Una nueva versión del método de la burbuja seria limitando el número de comparaciones, dijimos que era inútil que se compare consigo misma. Si tenemos una lista de 10.000 elementos, entonces son 10.000 comparaciones que están sobrando. Imaginemos si tenemos 1.000.000 de elementos. El método sería mucho más optimo con “n” comparaciones menos (n = total de elementos).

Burbuja optimizada:

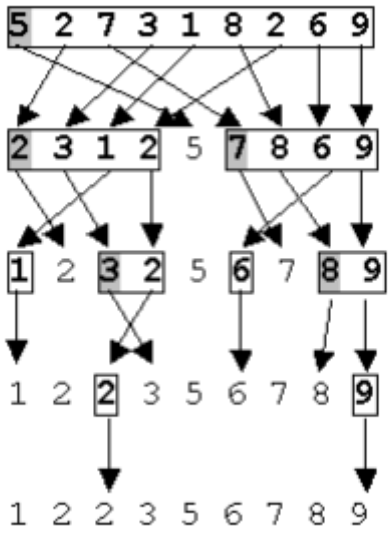
Si al cambio anterior (el de la burbuja mejorada) le sumamos otro cambio, el hecho que los elementos que están detrás del que se está comparando, ya están ordenados, las comparaciones serian aún menos y el método sería aún más efectivo. Si tenemos una lista de 10 elementos y estamos analizando el quinto elemento, qué sentido tiene que el quinto se compare con el primero, el segundo o el tercero, si supuestamente, ¿ya están ordenados? Entonces optimizamos más aun el algoritmo, quedando nuestra versión final del algoritmo optimizado de la siguiente manera:

```
Bubblesort(int matriz[])
{
        int buffer;
        int i,j;
        for(i = 0; i < matriz.length; i++)
        {
                for(j = 0; j < i; j++)
                {
                        if(matriz[i] < matriz[j])
                        {
                                buffer = matriz[j];
                                matriz[j] = matriz[i];
                                matriz[i] = buffer;
                        }
                }
        }
}
```

Quicksort

Sin duda, este algoritmo es uno de los más eficientes. Este método es el más rápido gracias a sus llamadas recursivas, basándose en la teoría de divide y vencerás. Lo que hace este algoritmo es dividir recursivamente el vector en partes iguales, indicando un elemento de inicio, fin y un pivote (o comodín) que nos permitirá segmentar nuestra lista. Una vez dividida, lo que hace, es dejar todos los mayores que el pivote a su derecha y todos los menores a su izq. Al finalizar el algoritmo, nuestros elementos están ordenados. Por ejemplo, si tenemos 3 5 4 8 básicamente lo que hace el algoritmo es dividir la lista de 4 elementos en partes iguales, por un lado 3, por otro lado 4 8 y como comodín o pivote el 5. ¿Luego pregunta, 3 es mayor o menor que el comodín? Es menor, entonces lo deja al lado izq. Y como se acabaron los elementos de ese lado, vamos al otro lado. 4 es mayor o menor que el pivote? Menor, entonces lo tira a su izq. Luego pregunta por el 8, al ser mayor lo deja donde esta, quedando algo así: 3 4 5 8

En esta figura se ilustra de mejor manera un vector con más elementos, usando como pivote el primer elemento:


5 2 7 3 1 8 2 6 9
2 3 1 2 5 7 8 6 9
1 2 3 2 5 6 7 8 9
1 2 2 3 5 6 7 8 9
1 2 2 3 5 6 7 8 9

El algoritmo es el siguiente:

```
public void _Quicksort(int matrix[], int a, int b)
{
	this.matrix = new int[matrix.length];
	int buf;
	int from = a;
	int to = b;
	int pivot = matrix[(from+to)/2];
	do
	{
		while(matrix[from] < pivot)
		{
			from++;
		}
		while(matrix[to] > pivot)
		{
			to--;
		}
		if(from <= to)
		{
			buf = matrix[from];
			matrix[from] = matrix[to];
			matrix[to] = buf;
			from++; to--;
		}
	}while(from <= to);
	if(a < to)
	{
		_Quicksort(matrix, a, to);
	}
	if(from < b)
	{
		_Quicksort(matrix, from, b);
	}
	this.matrix = matrix;
}
```

ShellSort

Este método es una mejora del algoritmo de ordenamiento por Inserción (Insertsort). Si tenemos en cuenta que el ordenamiento por inserción es mucho más eficiente si nuestra lista de números esta semi-ordenada y que desplaza un valor una única posición a la vez. Durante la ejecución de este algoritmo, los números de la lista se van casi-ordenando y finalmente, el último paso o función de este algoritmo es un simple método por inserción que, al estar casi-ordenados los números, es más eficiente.

El algoritmo:

```
Bubblesort(int matriz[])
{
        int buffer;
        int i,j;
        for(i = 0; i < matriz.length; i++)
        {
                for(j = 0; j < i; j++)
                {
                        if(matriz[i] < matriz[j])
                        {
                                buffer = matriz[j];
                                matriz[j] = matriz[i];
                                matriz[i] = buffer;
                        }
                }
        }
}
```

Radix

Este ordenamiento se basa en los valores de los dígitos reales en las representaciones de posiciones de los números que se ordenan.

Por ejemplo, el número 235 se escribe 2 en la posición de centenas, un 3 en la posición de decenas y un 5 en la posición de unidades.

Reglas para ordenar.

- Empezar en el dígito más significativo y avanzar por los dígitos menos significativos mientras coinciden los dígitos correspondientes en los dos números.

- El número con el dígito más grande en la primera posición en la cual los dígitos de los dos números no coinciden es el mayor de los dos (por supuesto sí coinciden todos los dígitos de ambos números, son iguales).

Este mismo principio se toma para Radix Sort, para visualizar esto mejor tenemos el siguiente ejemplo. En el ejemplo anterior se ordenó de izquierda a derecha. Ahora vamos a ordenar de derecha a izquierda.

Archivo original.

25 57 48 37 12 92 86 33

Asignamos colas basadas en el dígito menos significativo.

Parte delantera Parte posterior

0

1

2 12 92

3 33

4

5 25

6 86

7 57 37

8 48

9

10

Después de la primera pasada:

12 92 33 25 86 57 37 48

Colas basadas en el dígito más significativo.

Parte delantera Parte posterior

0

1 12

2 25

3 33 37

4 48

5 57

6

7

8 86

9 92

10

Archivo ordenado: 12 25 33 37 48 57 86 92

A S O R T I N G E X A M P L E

A E O L M I N G E A X T P R S

A E A E G I N M L O

A A E E G

A A

A A

E E G

E E

I N M L O

L M N O

L M

N O

S T P R X

S R P T

P R S

R S

```
#include
#include
#define NUMELTS 20

void radixsort(int x[], int n)
{
  int front[10], rear[10];
  struct {
    int info;
    int next;
  } node[NUMELTS];
  int exp, first, i, j, k, p, q, y;

  /* Inicializar una lista viculada */
  for (i = 0; i < n-1; i++) {
    node[i].info = x[i];
    node[i].next = i+1;
  } /* fin del for */
  node[n-1].info = x[n-1];
  node[n-1].next = -1;
  first = 0; /*first es la cabeza de la lista vinculada */
  for (k = 1; k < 5; k++) {
    /* Suponer que tenemos n£meros de cuatro dÃgitos */
```

```
    for (i = 0; i < 10; i++) {
      /*Inicializar colas */
      rear[i] = -1;
      front[i] = -1;
    } /*fin del for */
    /* Procesar cada elemento en la lista */
    while (first != -1) {
      p = first;
      first = node[first].next;
      y = node[p].info;
      /* Extraer el kâsimo dÁgito */
      exp = pow(10, k-1);        /* elevar 10 a la (k-1)âsima potencia */
      j = (y/exp) % 10;
      /* Insertar y en queue[j] */
      q = rear[j];
      if (q == -1)
           front[j] = p;
      else
           node[q].next = p;
      rear[j] = p;
    } /*fin del while */

    /* En este punto, cada registro est  en su cola bas ndose en el dÁgito k
       Ahora formar una lista £nica de todos los elementos de la cola. Encontrar
       el primer elemento. */
    for (j = 0; j < 10 && front[j] == -1; j++);
      ;
    first = front[j];

    /* Vincular las colas restantes */
    while (j <= 9) {  /*Verificar si se ha terminado */
      /*Encontrar el elemento siguiente */
      for (i = j+1; i < 10 && front[i] == -1; i++);
           ;
      if (i <= 9) {
           p = i;
           node[rear[j]].next = front[i];
      } /* fin del if */
      j = i;
    } /* fin del while */
    node[rear[p]].next = -1;
  } /* fin del for */

  /* Copiar de regreso al archivo original */
  for (i = 0; i < n; i++) {
    x[i] = node[first].info;
    first = node[first].next;
  } /*fin del for */
} /* fin de radixsort*/


int main(void)
{
  int x[50] = {NULL}, i;
  static int n;

  printf("\nCadena de n£meros enteros: \n");
  for (n = 0;; n++)
    if (!scanf("%d", &x[n])) break;
  if (n)
    radixsort (x, n);
  for (i = 0; i < n; i++)
    printf("%d ", x[i]);
  return 0;
}
```

Referencias

1. Fernando A. Lagos B. (2007). Algoritmos de ordenamiento. México: Copyleft.
2. Ricardo Ferris. (1996). Algoritmos y estructuras de datos 1. México: McGraw-Hill.
3. Robert Sedgewick, "Algorithms in C", (third edition), Addison-Wesley, 2001
4. Thomas Cormen et al, “Algorithms”, (eighth edition), MIT Press, 1992.
5.