Algoritmos de Ordenamiento

Insertion Sort

Este es uno de los métodos más sencillos. Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo. Este es el algoritmo:

Procedimiento ***Insertion Sort***
Este procedimiento recibe el arreglo de datos a ordenar **a[]** y altera las posiciones de sus elementos hasta dejarlos ordenados de menor a mayor. **N** representa el número de elementos que contiene a[].

```
paso 1: [Para cada pos. del arreglo]  For i <- 2 to N do
paso 2: [Inicializa v y j]                    v <- a[i]
                                              j <- i.
paso 3: [Compara v con los anteriores]        While a[j-1] > v AND j>1 do
paso 4: [Recorre los datos mayores]           Set a[j] <- a[j-1],
paso 5: [Decrementa j]                              set j <- j-1.
paso 5: [Inserta v en su posición]      Set a[j] <- v.
paso 6: [Fin]                           End.
```

Ejemplo:
Si el arreglo a ordenar es a = ['a','s','o','r','t','i','n','g','e','x','a','m','p','l','e'],
el algoritmo va a recorrer el arreglo de izquierda a derecha. Primero toma el segundo dato 's' y lo asigna a *v* y *i* toma el valor de la posición actual de *v*.
Luego compara esta 's' con lo que hay en la posición j-1, es decir, con 'a'. Debido a que 's' no es menor que 'a' no sucede nada y avanza i.
Ahora v toma el valor 'o' y lo compara con 's', como es menor recorre a la 's' a la posición de la 'o'; decrementa j, la cual ahora tiene la posición en dónde estaba la 's'; compara a 'o' con a[j-1] , es decir, con 'a'. Como no es menor que la 'a' sale del for y pone la 'o' en la posición a[j]. El resultado hasta este punto es el arreglo siguiente: a = ['a','o','s','r',....]
Así se continúa y el resultado final es el arreglo ordenado :
a = ['a','a','e','e','g','i','l','m','n','o','p','r','s','t','x']

Selection Sort

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo.

Procedimiento ***Selection Sort***

```
paso 1: [Para cada pos. del arreglo]      For i <- 1 to N do
paso 2: [Inicializa la pos. del menor]          menor <- i
paso 3: [Recorre todo el arreglo]               For j <- i+1 to N do
paso 4: [Si a[j] es menor]                            If a[j] < a[menor] then
paso 5: [Reasigna el apuntador al menor]                    min = j
paso 6: [Intercambia los datos de la pos.
          min y posición i]                                 Swap(a, min, j).
paso 7: [Fin]                                         End.
```

Ejemplo:
El arreglo a ordenar es a = ['a','s','o','r','t','i','n','g','e','x','a','m','p','l','e']. Se empieza por recorrer el arreglo hasta encontrar el menor elemento. En este caso el menor elemento es la primera 'a'. De manera que no ocurre ningún cambio. Luego se procede a buscar el siguiente elemento y se encuentra la segunda 'a'. Esta se intercambia con el dato que está en la segunda posición, la 's', quedando el arreglo así después de dos recorridos: a = ['a','a','o','r','t','i','n','g','e','x','s','m','p','l','e'].
El siguiente elemento, el tercero en orden de menor mayor es la primera 'e', la cual se intercambia con lo que está en la tercera posición, o sea, la 'o'. Le sigue la segunda 's', la cual es intercambiada con la 'r'. El arreglo ahora se ve de la siguiente manera: a = ['a','a','e','e','t','i','n','g','o','x','s','m','p','l','r']. De esta manera se va buscando el elemento que debe ir en la siguiente posición hasta ordenar todo el arreglo.
El número de comparaciones que realiza este algoritmo es :
para el primer elemento se comparan n-1 datos, en general para el elemento i-ésimo se hacen n-i comparaciones, por lo tanto, el total de comparaciones es:
la sumatoria para i de 1 a n-1 (n-i) = 1/2 n (n-1).

Shellsort
Denominado así por su desarrollador Donald Shell (1959), ordena una estructura de una manera similar a la del Bubble Sort, sin embargo no ordena elementos adyacentes sino que utiliza una segmentación entre los datos. Esta segmentación puede ser de cualquier tamaño de acuerdo a una secuencia de valores que empiezan con un valor grande (pero menor al tamaño total de la estructura) y van disminuyendo hasta llegar al '1'. Una secuencia que se ha comprobado ser de las mejores es: ...1093, 364, 121, 40, 13, 4, 1. En contraste, una secuencia que es mala porque no produce un ordenamiento muy eficiente es ...64, 32, 16, 8, 4, 2, 1.
Su complejidad es de $O(n^{1.2})$ en el mejor caso y de $O(n^{1.25})$ en el caso promedio.

```
void shellsort ( int a[], int n) {
/* Este procedimiento recibe un arreglo a ordenar a[] y el tamaño del arreglo n. Utiliza en este caso una serie de t=6 incrementos h=[1,4,13,40,121,364] para el proceso (asumimos que el arreglo no es muy grande). */
        int x,i,j,inc,s;

        for(s=1; s < t; s++) {   /* recorre el arreglo de incrementos */
                        inc = h[s];
                for(i=inc+1; i < n; i++) {
                        x = a[i];
                        j = i-inc;
                        while( j > 0 && a[j] > x){
                                a[j+h] = a[j];
                                j = j-h;
                        }
                        a[j+h] = x;
                }
        }
}
```

Bubble Sort

El bubble sort, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Recorre el arreglo tantas veces hasta que ya no haya cambios. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

Procedimiento ***Bubble Sort***

paso 1: [Inicializa i al final de arreglo] For i <- N downto 1 do
paso 2: [Inicia desde la segunda pos.] For j <- 2 to i do
paso 4: [Si a[j-1] es mayor que el que le sigue] If a[j-1] < a[j] then
paso 5: [Los intercambia] Swap(a, j-1, j).
paso 7: [Fin] End.

Tiempo de ejecución del bubble sort:

- para el mejor caso (un paso) O(n)
- peor caso n(n-1)/2
- promedio $O(n^2)$

Merge Sort

El método Quicksort divide la estructura en dos y ordena cada mitad recursivamente. El caso del MergeSort es el opuesto, es decir, en éste método de unen dos estructuras ordenadas para formar una sola ordenada correctamente.
Tiene la ventaja de que utiliza un tiempo proporcional a: n log (n), su desventaja radica en que se requiere de un espacio extra para el procedimiento.
Este tipo de ordenamiento es útil cuando se tiene una estructura ordenada y los nuevos datos a añadir se almacenan en una estructura temporal para después agregarlos a la estructura original de manera que vuelva a quedar ordenada.

Procedimiento ***MergeSort***

```
/*recibe el arreglo a ordenar un índice l que indica el límite inferior del arreglo a ordenar y un índice r que indica el límite superior*/
void mergesort(int a[], int l, int r){
        int i,j,k,m,b[MAX];
        if (r > l) {
                m = (r+l) /2;
                mergesort(a, l, m);
                mergesort(a, m+1, r);
                for (i= m+1; i > l;i--)
                        b[i-1] = a[i-1];
                for (j= m; j < r;j++)
                        b[r+m-j] = a[j+1];
                for (k=l ; k <=r; k++)
                        if(b[i] < b[j])
                          a[k] = b[i++];
                        else
                          a[k] = b[j--];
        }
}
```

a = {a,s,o,r,t,i,n,g,e,x,a,m,p,l,e}
{a,s,
o,r,
a,o,r,s,
i,t,
g,n,
g,i,n,t,
a,g,i,n,o,r,s,t,
e,x,
a,m,
a,e,m,x,
l,p,
e,l,p}
a,e,e,l,m,p,x}

a = {a,a,e,e,g,i,l,m,n,o,p,r,s,t,x}

Heap Sort
Este método garantiza que el tiempo de ejecución **siempre** es de:
O(n log n)
El significado de *heap* en ciencia computacional es el de una cola de prioridades (priority queue). Tiene las siguientes características:

- Un heap es un arreglo de n posiciones ocupado por los elementos de la cola. (Nota: se utiliza un arreglo que inicia en la posición 1 y no en cero, de tal manera que al implementarla en C se tienen n+1 posiciones en el arreglo.)
- Se mapea un árbol binario de tal manera en el arreglo que el nodo en la posición i es el padre de los nodos en las posiciones (2*i) y (2*i+1).
- El valor en un nodo es mayor o igual a los valores de sus hijos. Por consiguiente, el nodo padre tiene el mayor valor de todo su sub-árbol.

Heap Sort consiste escencialmente en:

- convertir el arreglo en un heap
- construir un arreglo ordenado de atrás hacia adelante (mayor a menor) repitiendo los siguientes pasos:
  - sacar el valor máximo en el heap (el de la posición 1)
  - poner ese valor en el arreglo ordenado
  - reconstruir el heap con un elemento menos
- utilizar el mismo arreglo para el heap y el arreglo ordenado.

Procedimiento Heapsort

```
/* Recibe como parámetros un arreglo a ordenar y un entero que indica el numero de datos a ordenar */
void heapsort(int a[], int N){
        int k;
        for(k=N/2; k>=1; k--)
                downheap(a,N,k);
        while(N > 1){
                swap(a,1,N);
                downheap(a,--N,1);
```

```
        }
}
```

/* el procedimiento downheap ordena el árbol de heap para que el nodo padre sea mayor que sus hijos */

```
void downheap(int a[], int N, int r){
        int j, v;
        v = a[r];
        while (r <= N/2){
                j = 2*r;
                if(j < N && a[j] < a[j+1]);
                        j++;
                if( v >= a[j])
                        break;
                a[r] = a[j];
                r = j;
        }
        a[r] = v;
}
```

Partition-Exchange Sort o Quicksort

Es un método de ordenamiento recursivo y en lenguajes en donde no se permite la recursividad esto puede causar un retraso significativo en la ejecución del quicksort.

Su tiempo de ejecución es de $n \log_2 n$ en promedio.

```
void quicksort(int a[], int l, int r){
        int i,j,v;
        if(r > l){
                v = a[r];
                i = l-1;
                j = r;
                for(;;){
                        while(a[++i] < v && i < r);
                        while(a[--j] > v && j > l);
                        if( i >= j)
                                break;
                        swap(a,i,j);
                }
                swap(a,i,r);
                quicksort(a,l,i-1);
                quicksort(a,i+1,r);
        }
}
```

```
a = {a,s,o,r,t,i,n}
a i o r t s n
a i n r t s o
a i n o t s r
a i n o r s t
```