ALGORITMOS DE ORDENACIÓN Y DE BÚSQUEDA

1. Algoritmos de Ordenación

Para poder ordenar una cantidad determinada de datos existen distintos métodos (algoritmos) con distintas caracterésticas y complejidad. De cada método, es posible encontrar distintas implementaciones, propuestas, mejoras, etc.

Estos métodos sirven para ordenar una lista de elementos, en los ejemplos, los elementos se encuentran almacenados en arrays unidimensionales (vectores). Los elementos pueden ser de cualquier tipo (siempre que sean comparables y que las relaciones $<$ y $>$ tengan sentido). Además, cada uno puede ser utilizado para ordenar los elementos de menor a mayor o de mayor a menor, cambiando en cada caso la forma de comparar.

Existen métodos iterativos y métodos recursivos. Los métodos iterativos son simples de entender y de programar ya que son iterativos: simples ciclos y sentencias que hacen que el vector pueda ser ordenado. Dentro de los algoritmos iterativos encontramos:

- Bubble Sort o método de la Burbuja.
- Insertion Sort o método de Inserción.
- Selection Sort o método de Selección.
- Shellsort.

A su vez, veremos el algoritmo recursivo Quicksort, el cual es un algoritmo rápido, elegante y corto de codificar.

1.1. Bubble Sort o Método de la Burbuja

El método Bubble Sort es uno de los más simples: compara 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 y se los desea ordenar de menor a mayor:
5 6 1 0 3

El método de la burbuja comienza recorriendo los valores de izquierda a derecha, comenzando por la posición 0 (valor 5). Lo compara con el resto de las posiciones a su derecha: con el 6, con el 1, con el 0 y con el 3, cuando encuentra uno menor, los intercambia de posición. Luego continua con el siguiente (posición 2), comparando con todos los elementos de la lista a su derecha (los anteriores ya han sido ordenados, por lo que compararlos no tiene sentido), esperando ver si se cumple o no la misma condición que con el primer elemento. Asi, sucesivamente, hasta el último elemento de la lista.

Algorithm 1: BubbleSort.

```
1  #define SIZE 100
2  int swap(int *x, int *y)
3  {
4      int aux = *x;
5      *x = *y;
6      *y = aux;
7      return(0);
8  }
9  int BubbleSort()
10 {
11     int vector[SIZE], i, j, aux;
12     // inicializacion del vector
13
14     for(i=0; i <SIZE; i++) {
15         for(j=i+1; j <SIZE; j++) {
16             if (vector[j] <vector[i]) {
17                 swap(&vector[i], &vector[j]);
18             }
19         }
20     }
21     return 0;
22 }
```

1.2. Ordenación por inserción

La iteración principal de la ordenación por inserción examina sucesivamente todos los elementos de la lista, desde el segundo hasta el último, e inserta cada uno en el lugar adecuado entre sus predecesores dentro del vector. Antes de insertar cada elemento donde va, realiza un corrimiento hacia la derecha (para dejar libre la posición del vector donde se insertará el elemento).

Algorithm 2: InsertionSort.

```
1  #define SIZE 100
2  int InsertionSort()
3  {
4      int vector[SIZE], i, j, aux;
5      // inicializacion del vector
6      for(i=0; i <SIZE; i++) {
7          aux = vector[i];
8          j = i - 1;
9          while ((j>=0) && (vector[j] >aux)) {
10             vector[j+1] = vector[j];
11             j–; //doble simbolo menos
12         }
13         vector[j+1] = aux;
14     }
15     return 0;
16 }
```

Entonces, si el vector tiene N elementos (con índices de 0..N-1), el algoritmo consiste en recorrer todo el vector comenzando desde la segunda posición (i=1) y terminando en i=N-1. Para cada i, se trata de ubicar en el lugar correcto el elemento vector[i] entre los elementos anteriores: vector[0]..vector[i-1]. Dada la posición actual i, el algoritmo se basa en que los elementos vector[0], vector[1], ..., vector[i-1] ya están ordenados.

1.3. Ordenación por Selección

La ordenación por selección selecciona el menor elemento del vector y lo lleva al principio, a continuación selecciona el siguiente menor y lo pone en la segunda posición del vector y asi sucesivamente.
Si el tamaño del vector es N, lo que hace el algoritmo es: para cada i de [0..N-2] intercambia vector[i] con el mínimo elemento del subarreglo [vector[i+1], ..., vector[N]].

Algorithm 3: SelectionSort.

```
1  #define SIZE 100
2  int swap(int *x, int *y);
3  int SelectionSort()
4  {
5      int vector[SIZE], i, j, k, p, aux, limite = SIZE-1;
6      // inicializacion del vector
7      for(k=0; k <limite; k++) {
8          p = k;
9          for (i = k + 1; i <= limite; i++) {
10             if (vector[i] <vector[p]) {
11                 p = i;
12             }
13         }
14         if (p != k) {
15             swap (&vector[p], &vector[k]);
16         }
17     }
18     return 0;
19 }
```

1.4. ShellSort

Este método es una mejora del algoritmo de ordenamiento por Inserción: el ordenamiento por Inserción es mucho más eficiente si nuestra lista de números está 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.

Algorithm 4: ShellSort.

```
1  #define SIZE 100
2  int swap(int *x, int *y);
3  int ShellSort()
4  {
5      int vector[SIZE], i, j,aux, incr = SIZE;
6      // inicializacion del vector
7      do {
8          for (i = incr; i <SIZE; i++) {
9              for (j = i; (j>=incr) && (vector[j-incr]>vector[j]); j -= incr) {
10                 swap(&vector[j], &vector[j-incr]);
11             }
12         }
13         incr = incr / 2;
14     }while (incr >0);
15     return 0;
16 }
```

1.5. QuickSort - Ordenación rápida [1]

Se basa en la división en particiones de la lista a ordenar, aplica la técnica "divide y vencerás". El método es pequeño en código, rápido, elegante y eficiente.

El método se basa en dividir los n elementos de la lista a ordenar en dos partes separadas por un elemento: una partición izquierda, un elemento central llamado *pivote* y una partición derecha: la partición o división se hace de tal forma que todos los elementos de la partición o sublista izquierda son menores al pivote y todos los elementos de la partición o sublista derecha son mayores al pivote. Luego las dos sublistas se ordenan de forma independiente (y recurrente).

Primero se elige un pivote, y luego se utiliza dicho pivote para ordenar el resto de la lista en dos sublistas: una tiene todas los elementos menores y la otra todos los elementos mayores o iguales al pivote (o al revés). Estas dos sublistas se ordenan de forma recursiva utilizando el mismo algoritmo. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda sublista.

1.5.1. Algoritmo Quicksort

La primera etapa del algoritmo consiste en elegir el pivote, y luego se ha de buscar la forma de situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote.

Los pasos del algoritmo son:

- *Seleccionar* el elemento central de a[0..n-1] como pivote.
- *Dividir* los elementos restantes en particiones izquierda y derecha, de modo que ningún elemento de la izquierda tenga un elemento mayor que el pivote y ninguno de la derecha tenga un elemento menor al pivote.
- *Ordenar* la sublista izquierda usando quicksort recursivamente.
- *Ordenar* la sublista derecha usando quicksort recursivamente.

- La solución es partición izquierda seguida del pivote y a continuación partición derecha.

1.5.2. Ejemplo

Se ordena una lsita de números enteros aplicando el algoritmo quicksort. Como pivote se elige el elemento central de la lista.

Lista original: 8 1 4 9 6 3 5 2 7 0
pivote (elemento central): 6

La etapa 2 requiere mover los elementos mayores a la derecha y los menores a la izquierda. Para ello se recorre la lista de izquierda a derecha utilizando un índice i, que se inicializa en la posición más baja, buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice j inicializado a la posición más alta.

```
8   1   4   9     6     3   5   2   7   0
↑                                     ↑
→ i                                   j ←
```

Ahora se intercambian 8 y 0 para que estos dos elementos se sitúen correctamente en cada sublista y se incrementa el índice i, y se decrementa j para seguir los intercambios.

```
0   1   4   9     6     3   5   2   7   8
    ↑                             ↑
    i                             j
```

A medida que el algoritmo continúa, i se detiene en el elemento mayor, 9, y j se detiene en el elemento menor, 2.

```
0   1   4   9     6     3   5   2   7   8
        ↑                 ↑
        i                 j
```

Se intercambian los elementos mientras i y j no se crucen. En el momento en que se cruzan los índices se detiene este bucle. En el caso anterior se intercambian 9 y 2.

```
0   1   4   2     6     3   5   9   7   8
            ↑         ↑
            i         j
```

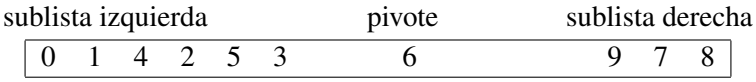
Continúa la exploración y ahora el contador i se detiene en el elemento con valor 6 (que es el pivote) y el índice j se detiene en el elemento menor, valor 5. Se intercambian los valores 6 y 5, se incrementa i y se decrementa j.

```
0   1   4   2   5   3     6     9   7   8
                ↑
               i=j
```

Los índices ahora son i = 5, j = 5. Continua la exploración hasta que i >j, acaba con índices i = 6, j = 5.

```
0   1   4   2   5   3     6     9   7   8
                ↑     ↑
                j     i
```

En esta posición los índices i y j han cruzado posiciones en el arreglo. En este caso se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede j está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



1.6. Codificación del algoritmo

A continuación se codifica la funicón recursiva `quicksort()`; a esta función se la llama pasando como argumentos el arreglo `a[]` y los índice que la delimitan `0` y `n-1` (índices inferior y superior). Llamada a la función: `quicksort(a[], 0, n-1)` y la codificación de la función:

Algorithm 5: Quicksort.

```
1  void quicksort(int a[], int primero, int ultimo)
2  {
3      int i, j, central, aux;
4      int pivote;
5      central = (primero + ultimo) / 2;
6      pivote = a[central];
7      i = primero;
8      j = ultimo;
9      do {
10         while (a[i] <pivote) i++;
11         while (a[j] >pivote) j–; // simbolo menos 2 veces
12         if (i <= j) {
13             swap(&a[i], &a[j]);
14             i++;
15             j–; // simbolo menos 2 veces
16         }
17     }while (i <= j)
18     if (primero <j)
19         quicksort(a, primero, j);       /* llamada recursiva con sublista izquierda */
20     if (i <ultimo)
21         quicksort(a, i, ultimo);        /* llamada recursiva con sublista derecha */
22 }
```

2. Búsquedas

Con mucha frecuencia se trabaja con grandes volúmenes de datos, y muchas veces es necesario determinar si un dato se encuentra en un conjunto de datos. Específicamente el proceso de encontrar un elemento específico en un arreglo, lista, etc, se denomina búsqueda. Las técnicas de búsqueda más utilizadas son: *búqueda lineal o secuencial*, la técnica más sencilla, y *búsqueda binaria o dicotómica*, la técnica más eficiente.

2.1. Búsqueda Secuencial [1]

Este algoritmo busca un elemento dado, recorriendo secuencialmente el arreglo desde un elemento al siguiente desde la primer posición y se detiene cuando se encuentra el elemento o bien cuando se alcanza el final del arreglo.

Este algoritmo buscará desde el primer elemento, el segundo, y así sucesivamente hasta encontrar el elemento o llegar al final del arreglo. Se utiliza un *while* para su implementación:

Algorithm 6: Búsqueda secuencial

```
1  #define n
2  int BusquedaSec(int lista[MAX], int elemento)
3  {
4      int encontrado = 0;
5      int i = 0;
6      while ((!encontrado) && (i <= MAX -1)) {
7          encontrado = (lista[i] == elemento);
8          i++;
9      }
10     return encontrado;
11 }
```

Esta búsqueda se aplica a cualquier lista: no hace falta que los datos estén ordenados. En caso de que los datos estén ordenados, analizar cómo se puede mejorar el algoritmo. Esta implementación del algoritmo retorna 1 si el elemento fue encontrado, 0 caso contrario (valor de la variable *encontrado*).

2.2. Búsqueda Binaria o Dicotómica [1]

Si la lista está ordendada, la búsqueda dicotómica proporciona una técnica de búsqueda mejorada.

La búsqueda dicotómica es similar a la búsqueda de una palabra en un diccionario: uno puede abrir el diccionario por la mitad, y verificar si la letra de la página abierta es mayor o menor a la letra inicial de la palabra buscada. Si es mayor, se descarta la mitad mayor y se utilizan las páginas con letras menores para continuar con el mismo proceso. Al contrario, se continua el proceso con la segunda mitad en que fue dividido el diccionario. El proceso implementado por la búsqueda binaria es similar, y se fundamenta en que los datos de la lista están ordenados.

Suponiendo que la lista está almacenada en el arreglo `a[]`, los índices de la lista son inf = 0 y sup = n - 1 donde n es la cantidad de elementos del arreglo. Los pasos a seguir son:

1. Calular el índice del punto central del arreglo: `central = (inf + sup) / 2` (división entera)

2. Comparar el valor de este elemento central con la clave:

- Si `a[central] < clave`, la nueva sublista de búsqueda está en el rango
`central + 1 .. sup`
- Si `clave < a[central]`, la nueva sublista de búsqueda será:
`inf .. central - 1`

Algoritmo de la búsqueda binara: devuelve el índice del alemento buscado, o bien -1 en caso de que el elemento no esté en la lista.

Algorithm 7: Búsqueda Binaria

```
1  int BusquedaBin(int lista[], int n, int clave)
2  {
3      int central, bajo, alto;
4      int valorCentral;
5      int bajo = 0;
6      int alto = n-1;
7      while (bajo <= alto) {
8          central = (bajo + alto) / 2;
9          valorCentral = lista[central];
10         if (clave == valorCentral)
11             return central;
12         else if (clave <valorCentral)
13             alto = central-1;
14         else
15             bajo = central +1;
16     }
17     return 0;
18 }
```

Referencias

[1] Luis Joyanes Aguilar, Ignacio Zahonero Martínez. *Programación en C, C++, JAVA y UML*. McGrawHill. 2010.