Maestría en Tecnología de Cómputo

Curso propedéutico de Programación

Tema III: Estructuras de datos.

Conferencia No. 3

Título: Estructuras de datos estáticas.

Objetivo: Identificar y utilizar las estructuras de datos estáticas en el lenguaje C para la implementación de algoritmos sencillos que involucren colecciones de datos.

Sumario:

Introducción.. 2

1. Arreglos Unidimensionales 2

   1.1 Inicialización. 3

   1.2 Tamaño físico y lógico de un arreglo 3

2. Arreglos Multidimensionales 4

   2.1 Arreglos rectangulares 4

   2.2 Inicialización de arreglos rectangulares 6

3. Seudocódigo para arreglos..................................... 6

4. Cadenas de c aracteres en C. 7

   4.1 Conversiones. 7

5. Cadenas de caracteres... 7

   5.1 Ejemplo. 8

6. Estructuras estáticas de C 11

   6.1 Acceso a los miembros de una estructura. 12

   6.2 Matrices de estructuras. 12

7. Métodos recursivos.. 12

   7.1 Ejemplo simple de recursividad: El factorial de un número ... 13

   7.2 Otro ejemplo simple de recursividad: La sucesión Fibonacci... 15

   7.3 Recursividad versus Iteración. 18

8. Algoritmos de búsqueda. 19

   8.1 Búsqueda secuencial en un arreglo. 19

   8.2 Búsqueda binaria en un arreglo. 19

9. Algoritmos de ordenamiento. 22
9.1 Ordenamiento por mínimos sucesivos. 23
9.2 Ordenamiento por burbuja o por comparación de pares consecutivos........................ 25
Conclusiones.. 27

Introducción

Hasta el momento se ha almacenado la información que se desea procesar en variables de algún tipo simple estándar. Pero, qué pasa si se necesita trabajar con una lista de elementos, por ejemplo, una lista de edades de un grupo de personas, o de valores de temperatura máxima de cada día de un período, o de los pasajeros de un vuelo de avión, etc. Nótese que:

- La cantidad de elementos de las listas pudiera ser diferente. Por poner solamente un ejemplo, analícese que no todos los vuelos de avión tienen la misma cantidad de pasajeros.
- Si se necesita hacer varias operaciones con los elementos de la lista, por ejemplo, promediarlos y después contar los que están por encima del promedio. Será obligatorio tenerlos almacenados todos permanentemente, al menos mientras dure el procesamiento.

En esta conferencia se comenzará el estudio de un nuevo tipo de datos que permite dar solución a este problema, el arreglo.

1. Arreglos Unidimensionales

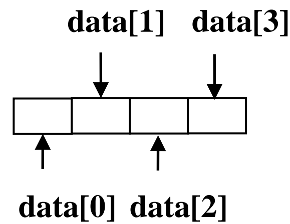
El arreglo (en Inglés *array*) es un tipo de dato que permite almacenar una colección de elementos ***del mismo tipo***, ***bajo un mismo identificador***, y que permite tratar la colección como un todo y también acceder a cada elemento por separado. Los elementos de un arreglo se diferencian entre sí por la posición que ocupan dentro del mismo, la cual se expresa a través de uno o más subíndices. En C los índices comienzan en 0 y terminan en *n-1,* donde *n* es el número de elementos del arreglo.

La razón por la que en este lenguaje, al igual que en C++ y C#, los subíndices comienzan en 0 se debe a que los valores de los arreglos están almacenados en localizaciones de memorias continuas. El nombre del arreglo es visto como una referencia al primer elemento del arreglo. Los valores de los arreglos son almacenados en localizaciones continuas de memoria de manera que para encontrar la dirección en memoria de un elemento específico la computadora adiciona el subíndice a la dirección del primer elemento. En el código que se muestran a continuación se crea un arreglo de enteros de 4 elementos.

```
int data[4];
```

La definición de un arreglo en C se realiza poniendo primeramente el tipo de datos (en este caso int), luego el nombre del arreglo que se define, y entre corchetes la cantidad de elementos que almacenará.

Básicamente lo que se hace en esta línea es crear una variable de referencia o puntero a un arreglo. En la siguiente figura se muestra el segmento de memoria que corresponde al arreglo data.



1.1 Inicialización.

Si el arreglo que se define corresponde al ámbito global, será inicializado automáticamente con 0, si es de tipo numérico, o con el carácter 0, si es de tipo char. Por el contrario, si se trata de un arreglo de punteros, se inicializará con el valor null.

Sin embargo, si el arreglo se define en un ámbito global, no será inicializado automáticamente, sino que almacenará valores indeterminados (basura).

C permite declarar, crear e inicializar un arreglo en una sola sentencia.

```
int data[2] = { 10, 20 }; // forma extendida.
```

En este caso el arreglo es inicializado por una lista de expresiones separadas por comas y encerradas entre llaves. Esta línea también se puede escribir omitiendo el tamaño del arreglo de la siguiente manera:

```
int data[] = { 10, 20 };  // forma abreviada.
```

En este caso el compilador usa el número de expresiones como el tamaño del arreglo.

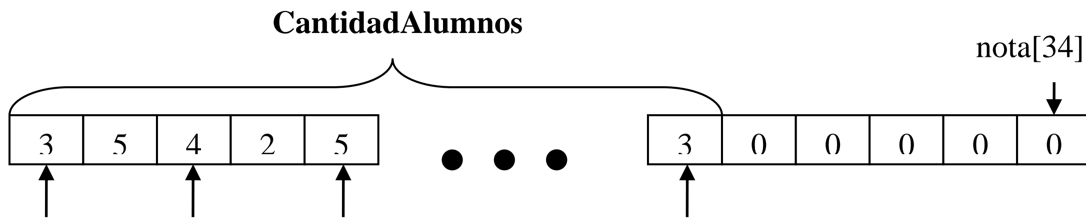
1.2 Tamaño físico y lógico de un arreglo

El **tamaño físico** de un arreglo es el tamaño que éste ocupa físicamente en la memoria, que como vimos anteriormente depende del tipo y cantidad de elementos.

El **tamaño lógico** es la cantidad de elementos del arreglo que realmente se usan durante la ejecución. Por ejemplo, si se declara:

```
int nota[35]; // para almacenar las notas de estudiantes de una brigada
```

Según la brigada que se procese en cada ejecución del programa la cantidad de estudiantes podrá ser menor o igual a 35. Si se trata de una brigada de 30 estudiantes los últimos 5 elementos del arreglo no se usarán (aunque la memoria se mantiene reservada para él durante toda la ejecución). En ese caso será necesario tener una variable auxiliar donde se mantenga la cantidad de alumnos de la brigada, por ejemplo **CantidadAlumnos.** De esta forma se garantiza siempre acceder a los elementos desde nota[0] hasta **nota[CantidadAlumnos - 1].**



En el siguiente ejemplo se inicializa un arreglo, se imprimen sus valores, se calcula la suma de sus elementos y se calcula el valor promedio.

```
#include <stdio.h>

main()
{
  int data[] = { 11, 12, 13, 14, 15, 16, 17 };
  int sum = 0;
  for (int i = 0; i < 7; i ++)
  {
    sum = sum + data[i];
    printf("%d , ", data[i]);
  }
  double average = sum / 7.0;
  printf("\n sum = %d \n average = %d", sum, average);
}
```

```
11, 12, 13, 14, 15, 16, 17,
sum = 98
average = 14
```

2. Arreglos Multidimensionales

Un arreglo puede imaginarse como una fila de "celdas" consecutivas. Puede ser útil imaginar también un arreglo como una tabla (con filas y columnas) de celdas, por ejemplo para representar los píxeles o puntos de una imagen bidimensional. Este es el clásico arreglo de dos dimensiones o bidimensional, también conocido como tabla o matriz.

Del mismo modo podría imaginarse un arreglo de tres dimensiones, por ejemplo para representar imágenes espaciales. Un lector más familiarizado con la matemática o la física también podría pensar en la utilidad de arreglos de más dimensiones.

2.1 Arreglos rectangulares

Un arreglo rectangular es un arreglo de dos o más dimensiones. El clásico arreglo de dos dimensiones se expresa con la sintaxis:

<tipo de los elementos> <nombre del arreglo>[][]

Donde los dos corchetes indican que es de dos dimensiones. Similarmente un arreglo de tres dimensiones se expresaría:

<tipo de los elementos><nombre del arreglo>[][][]

y de forma similar para cualquier cantidad de dimensiones.

Por ejemplo:

```
double matrix[][]; //dos dimensiones
```

```
int plane[][][]; //tres dimensiones
```

Para asignar memoria se introduce el valor deseado entre corchetes. Por ejemplo si declaramos:

```
int data [3][5];
```

se podría representar el arreglo de elementos como muestra la siguiente tabla:

	Columna 0	**Columna 1**	**Columna 2**	**Columna 3**	**Columna 4**
Fila 0	a[0,0]	a[0,1]	a[0,2]	a[0,3]	a[0,4]
Fila 1	a[1,0]	a[1,1]	a[1,2]	a[1,3]	a[1,4]
Fila 2	a[2,0]	a[2,1]	a[2,2]	a[2,3]	a[2,4]

Para ilustrar estas ideas, escribiremos un programa para llenar un arreglo de dos dimensiones con valores generados internamente.

```
#include <stdio.h>
main()
{
  int data[3][5];
  int sum = 0;
  for (int i = 0; i < 3; ++i)
  {
    printf("Row %d:, ", i);
    for (int j = 0; j < 5; ++j)
    {
      data[i][j] = i * j;
      printf("%d , ", data[i][j]);
    }
    printf("\n");
  }
}
```

```
Row 0: 0, 0, 0, 0, 0,
Row 1: 0, 1, 2, 3, 4,
Row 2: 0, 2, 4, 6, 8,
```

2.2 Inicialización de arreglos rectangulares

Los arreglos multidimensionales pueden ser inicializados con un conjunto de valores, al igual que los de una sola dimensión como por ejemplo:

```
int a[][][] = {{1,2,3,4}, {4,5,6,7},{8,9,10,11}}; // arreglo de dimension 3 x 4.

int b[][][] = {{{1,2,3,4}, {4,5,6,7},{8,9,10,11}},{{12,13,14,15},
{16,17,18,19},{20,21,22,23}}}; // arreglo de dimension 2 x 3 x 4.
```

3. Seudocódigo para arreglos.

Unidimensionales

seudocódigo	
DIMENSIONAR nombreArreglo(<tamaño>)	Crear un vector
nombreArreglo(indice)←<valor>	Asignar un elemento
nombreArreglo ← {valor1,valor2,…}	Inicializar un arreglo
Londitud ← EXTRAER DIMENSIÓN (nombreArreglo)	Extraer dimensión

Bidimensionales.

seudocódigo	
DIMENSIONAR nombreMatriz(<#filas>,<#columnas>)	Crear una matriz
nombreArreglo(fila,columna)←<valor>	Asignar un elemento
nombreArreglo ← {valor11,valor12,…;valor21,valor22…, valor31,valor32…}	Inicializar un arreglo completo
(#filas, #columnas) ← EXTRAER DIMENSIÓN (nombreArreglo)	Extraer dimensión del arreglo

En programación, una **cadena de caracteres**, **palabra**, **ristra de caracteres** o **frase** (*string* en inglés) es una secuencia ordenada de longitud arbitraria (aunque finita) de elementos que pertenecen a un cierto lenguaje formal o alfabeto análogas a una frase o a una oración. En general, una **cadena de caracteres** es una sucesión de caracteres (letras, números u otros signos o símbolos).

Desde un punto de vista de la programación, si no se ponen restricciones al alfabeto, una cadena podrá estar formada por cualquier combinación finita de todo el juego de caracteres disponibles (las letras de la 'a' a la 'z' y de la 'A' a la 'Z', los números del '0' al '9', el espacio en blanco ' ', símbolos diversos '!', '@', '%', etc). En este mismo ámbito (el de la programación), se utilizan normalmente como un tipo de dato predefinido, para palabras, frases o cualquier otra sucesión de caracteres.

Los primeros ejemplos del uso de cadenas se vieron desde el Conferencia 1 cuando se usaron las cadenas de caracteres para dar la salida de los resultados visualizadas en forma de un texto. Este fue el caso del primer ejemplo, imprimir “Hola Mundo”:

La secuencia de símbolos `"Hola mundo!!!"` que debe escribirse entre doble comillas, es una cadena.

4. Cadenas de c aracteres en C.

La palabra clave **char** se utiliza para declarar un carácter Unicode en el intervalo indicado en la siguiente tabla. Los caracteres Unicode son caracteres de 16 bits que se utilizan para representar la mayoría de los lenguajes escritos de todo el mundo.

Tipo	Intervalo	Size
char	U+0000 a U+ffff	Carácter Unicode de 16 bits

Las constantes de tipo **char** se pueden escribir como literales de cadena, secuencias de escape hexadecimales o representaciones Unicode.

4.1 Conversiones.

Un tipo **char** se puede convertir implícitamente en **ushort, int, uint, long, ulong, float, double o decimal**. Sin embargo, no existen conversiones implícitas desde otros tipo al tipo **char.**

5. Cadenas de caracteres

Las cadenas de caracteres, expresadas en C como arreglos de char son secuencias de símbolos (caracteres) que se emplean para representar la información simbólica (no numérica) involucrada en una aplicación.

Para poder leer de teclado, es necesario primeramente definir el arreglo donde será almacenada, y posteriormente leerla. Por ejemplo:

```
#include <stdio.h>
main()
{
  char cadena[50];
  scanf(“%s”, cadena);
  printf(“%s\n”, cadena);
}
```

Por defecto, la función scanf lee datos delimitados por espacios en blanco. En el caso de las cadenas de caracteres, si se desea leer datos con un formato especificado (por ejemplo una línea completa) es necesario utilizar una especificación de formato especializada para la función scanf.

Otra opción para leer caracteres desde teclado es utilizar la función getchar. Nótese que en cualquier caso es necesario definir a priori la cantidad de caracteres a almacenar.

Nótese además, que para leer los datos, no es necesaria la utilización del operador &, dado que el nombre de un arreglo apunta a la dirección de memoria donde se almacenará. Las cadenas de caracteres en C finalizan con el carácter \0.

Para leer cadenas de caracteres como un todo, y no carácter a carácter, puede utilizarse también la función gets. Esta función devuelve un puntero a la cadena de caracteres leída. Si hubiese ocurrido un error, o se llegase al fin del fichero (EOF), la cadena tendrá un valor a un puntero nulo, es decir, un valor NULL.

C posee en sus bibliotecas funciones que permiten realizar operaciones diversas con cadenas de caracteres, tales como conocer la cantidad de caracteres de una cadena, comparar dos cadenas, copiarlas, convertirlas a otros tipos de datos, entre otras.

5.1 Ejemplo.

Escriba un programa que lea varias líneas de texto hasta que el usuario tecle la marca de fin de fichero (EOF, formada por la combinación de techas [Ctrl] + [z]) e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras y así sucesivamente.

Solución.

Para resolver este problema se deben considerar varios aspectos:

- Cada palabra debe terminar con un espacio, excepto la última palabra que puede o no tenerlo.

Me_gusta_la_ _programacion

Simboliza el espacio

- La separación entre dos palabras puede tener uno o más espacios.

Me_gusta_la_ _programacion

- Una línea puede terminar o no en espacios.

Me_gusta_la_ _programacion_

En un arreglo podemos guardar los contadores de palabras por cantidad de letras. Para este caso los elementos del arreglo tendrían los siguientes valores:

Índice del arreglo

0 1 2 3 n-3 n-2 n-1

0	2	0	1	• • •	1		

1 2 3 4 n-2 n-1 n

Cantidad de letras

Cantidad de palabras

donde además, n es el tamaño real del arreglo y n – 2 su tamaño lógico que coincide también con la cantidad de letras de la palabra más larga.

Seudocódigo.

```
INICIO                                   Arreglo
    DIMENSIONAR cantidad(1000)
    contador ← 0
    mayorPalabra ← 0  ←——  Tamaño lógico del arreglo
    HACER
        LEER Linea
        longitudLinea  ← EXTRAER DIMENSIÓN(linea)
        SI línea <> "fin" ENTONCES
            DESDE i←0 HASTA longitudLinea
                SI línea(i) <> ' ' ENTONCES
                    contador ← contador + 1          Me_gusta_la__programacion
                    SI( i = longitudLine - 1) ENTONCES ←——
                        cantidad(contador – 1) ← cantidad(contador – 1) + 1
```

```
SI mayorPalabra < contador ENTONCES
    mayorPalabra ← contador
FIN SI
FIN SI
SI NO SI (contador > 0) ENTONCES    ←Me_gusta_la__programacion_
    cantidad(contador – 1) ← cantidad(contador – 1) + 1
    Si  mayorPalabra < contador  ENTONCES
        mayorPalabra ← contador
    FIN SI
    contador ←  0
FIN SI
FIN DESDE
contador ← 0
FIN SI
MIENTRAS (linea <> "fin")
IMPRIMIR ("longitud\tcantidad")
DESDE I = 0 HASTA mayorPalabra
    IMPRIMIR (i + 1) + "\t\t" + cantidad(i)
FIN DEADE
FIN
```

<u>Código en C</u>

```
#include <stdio.h>

main()
{
  int cantidad[1000];
  char linea[1000];
  char *c = NULL;
  int contador = 0;
  int mayorPalabra = 0;

  for (int i = 0; i < 1000; i++)
    cantidad[i] = 0;
```

```
  printf("Introduzca lineas de texto, para finalizar presione las 
teclas [Ctrl] + [z]\n");
  c = gets(linea);
  while (c != NULL)
  {
    for (int i = 0; i < strlen(linea); i++)
      {
        if (linea[i] != ' ')
        {
          contador++;
          if (i == (strlen(linea) - 1))
          {
            cantidad[contador - 1]++;
            if (mayorPalabra < contador)
              mayorPalabra = contador;
          }
        }
        else if (contador > 0)
        {
          cantidad[contador - 1]++;
          if (mayorPalabra < contador)
          mayorPalabra = contador;
          contador = 0;
        }
      }
      contador = 0;
      c = gets(linea);
  }

  printf("Longitud\tCantidad\n");
  for (int j = 0; j < mayorPalabra; ++j)
    printf("%d\t\t%d \n", j + 1, cantidad[j]);
}
```

6. Estructuras estáticas de C

Las estructuras (struct) son tipos estructurados de datos estáticos, que tienen como finalidad agrupar a diversas variables, relacionadas entre sí conceptualmente, para facilitar su manejo. También se les conoce como registros (records) Por ejemplo, de un estudiante se conoce su Nombre y Apellidos, su Edad, Sexo, Dirección particular, Teléfono….

Las estructuras en C se definen de la siguiente forma:

```
struct tipo_estructura
{
  tipo miembro 1;
  tipo miembro 2;
```

```
  …
  tipo miembro n;
}[lista de  variables];
```

Las estructuras no pueden tener calificadores como static, y no pueden ser inicializadas en su definición.

Para poder declarar una variable de tipo estructura, es necesario escribir:

```
struct tipo_estructura nombre_variable;
```

o utilizar typedef, donde se defina el sinónimo de tipos.

6.1 Acceso a los miembros de una estructura.

Para acceder a los miembros de una estructura se utilizará el . de la siguiente forma: variable_estructura.miembro

A su vez, un miembro de una estructura también puede ser de tipo estructura.

6.2 Matrices de estructuras.

Es posible definir un arreglo o matriz de estructuras, lo cual resulta muy provechoso, pues permite manipular los datos de forma más organizada. Para ello, es necesario primeramente declarar un tipo de estructura que va a coincidir con el tipo de los elementos de la matriz. Por ejemplo:

```
typedef struct
{
   char nombre[60];
  float nota;
} testudiante;

testudiante alumnos[100];
```

7. Métodos recursivos

Existen algoritmos que se explican en términos de sí mismos. Un algoritmo que se define parcialmente en términos de sí mismo se denomina recursivo. Al igual que otros lenguajes de programación, C permite definir funciones recursivas, es decir funciones que se llaman directa o indirectamente a sí mismos. Directamente si dentro del propio código de la función aparece una llamada a la función, indirectamente cuando dentro del código de la función se llama a otra función dentro de la cual se llama a otra función que su vez llama a otra función y así hasta que en una de las funciones se llama a la función original.

A simple vista puede parecer que ésta es una lógica circular que puede llevar a un proceso sin fin. ¿Cómo puede una función resolver un problema si éste se llama (o provoca que se termine llamando) a sí mismo? ¿Cuándo y cómo termina entonces? La clave de la respuesta está en que en cada "llamada" el problema se "simplifica" de tal modo que llegará un momento en que durante la ejecución de una llamada no se hará a su vez otra llamada recursiva: la rama, tiene ramas, que a su vez tienen ramas, que a su vez tienen ramas... pero en algún momento se llega a una rama que sólo tiene hojas.

7.1 Ejemplo simple de recursividad: El factorial de un número

El ejemplo más clásico de recursividad es la definición de la función factorial. Los matemáticos suelen decir que factorial de n es n por factorial de n - 1. Es decir $n! = n * (n - 1)!$. Sin embargo, dicho solamente de esta manera, esta definición sería incompleta o infinita porque siempre se puede estar restando 1 hasta el infinito negativo. Para ser más precisos habría que decir factorial de n se define para n positivo o cero, siendo 1 si n es 0, y n por el factorial de n - 1 si n es mayor que 0. Esta función factorial se expresa como sigue:

Factorial recursivo.

```
FUNCIÓN Factorial(numero)
        SI numero >= 0 ENTONCES
                SI numero = 0 ENTONCES
                        Factorial ← 1
                SI NO
                        Factorial ← numero * Factorial( numero - 1)
                FIN SI
        SI NO
                IMPRIMIR "El numero no puede ser negativo"
        FIN SI
FIN FUNCIÓN
```

De modo que el cálculo de Factorial de 4 quedaría

Factorial(4)

4 * Factorial(3)

4*(3*Factorial(2))

4*(3*(2*Factorial(1)))

4*(3*(2*(1*Factorial(0))))

4*(3*(2*(1*1))))

4*(3*(2*1))

4*(3*2)

4*6

24

Código en C

```
#include <stdio.h>
int factorial(int);
main()
{
   int n;
   printf("Entre un numero para calcular su factorial: \n");
   scanf("%d", &n);
   if (n >= 0)
   {
      int f = factorial(n);
      printf("El factorial de %d es: %d", n, f);
   }
   else
      printf("Lo sentimos, el numero debe ser positivo");
}
int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return n*factorial(n-1);
}
```

Si se observa el código de la implementación recursiva de Factorial se puede apreciar que después de regresar de la llamada recursiva a `Factorial(n-1)`, la función Factorial no hace nada más salvo regresar a quien lo llamó. Esto se conoce como recursividad de cola. Para estos casos es también simple encontrar una solución iterativa basada en ciclos.

Este ejemplo de factorial ilustra otra forma en que los matemáticos suelen definir factorial. Factorial de n es 1 si n es 0, y cuando n es mayor que 0 es el producto de n por n-1 por n-2 hasta 1 (es decir el producto de n por todos los números positivos menores que n).

Esta otra solución al cálculo de factorial también se expresa de manera muy simple en C usando un ciclo:

Factorial iterativo.

FUNCIÓN Factorial(numero)

SI numero >= 0 ENTONCES

SI numero = 0 ENTONCES

Factorial ← 1

SI NO

resultado ← 1

```
                    DESDE i ← numero HASTA 1 INCREMENTO -1
                              resultado ← resultado * i
                    FIN DESDE
                    Factorial← resultado
        FIN SI
          SI NO
                  Imprimir "El numero no puede ser negativo"
          FIN SI
FIN FUNCIÓN
```

Para comprobar que un algoritmo recursivo funciona correctamente es necesario que siga las siguientes 3 reglas:

1. Caso base. Siempre tiene que haber al menos un caso en que no se necesite resolver con una llamada recursiva sino que se resuelva directamente. Cuando retornamos 1 al tomar n valor cero.

2. Progresar hacia el caso base. Toda llamada recursiva debe progresar hacia el caso base. Cuando llamamos a factorial decrementando n.

3. Credibilidad. Asuma siempre que la llamada recursiva funcionará correctamente. Esto se fundamenta en la hipótesis de inducción. Esto quiere decir que cuando se diseña una función recursiva no hay que preocuparse (si se cumple con las dos reglas anteriores) de hacer el trazo de todos los largos caminos de llamadas recursivas, lo cual es una labor agobiante y probablemente muy difícil de entender.

Intuitivamente, asuma que es la computadora la que realizará todo el trabajo laborioso que hay detrás de las llamadas recursivas. Este trabajo consume recursos de memoria para ir conservando el rastro de las llamadas pendientes de terminar, es por ello que una función mal diseñada y que no progrese hacia uno de los casos base provocará un proceso aparentemente infinito que terminará por agotar los recursos de la computadora, lo que probablemente se traducirá en un mensaje de excepción de stack overflow (pila desbordada).

Hay una cuarta regla que será enunciada en el próximo epígrafe.

7.2 Otro ejemplo simple de recursividad: La sucesión Fibonacci

La serie Fibonacci

1, 1, 2, 3, 5, 8, 13, …

Empieza con 1 y 1, y tiene la propiedad de que cada número Fibonacci subsecuente es la suma de los dos números Fibonacci previos.

La serie ocurre en la naturaleza y en particular describe una forma de espiral. La relación de números sucesivos Fibonacci converge a un valor constante de 1,618… Este número,

también ocurre de forma repetida en la naturaleza y ha sido llamada la regla áurea o la media áurea.

Los seres humanos tienen tendencia a encontrar la media áurea estéticamente agradable. Los arquitectos a menudo diseñan las ventanas, habitaciones y edificios cuya longitud de ancho están en la relación de la media áurea. Las tarjetas postales a menudo se diseñan en una relación de longitud/ancho de la media áurea.

Los números de Fibonacci tienen otras muchas propiedades, por ejemplo la suma de los cuadrados de dos números de Fibonacci consecutivos es un número de Fibonacci. De hecho existe una revista *The Fibonacci Quaterly* dedicada sólo a publicar resultados sobre los números de Fibonacci.

La serie Fibonacci puede ser definida de forma recursiva como sigue:

$Fibonacci(1) = 1$

$Fibonacci(2) = 1$

$Fibonacci(n) = Fibonacci(n - 1) + Fibonacci(n - 2)$

La función que veremos a continuación calcula el número Fibonacci i-ésimo de forma recursiva.

Fibonacci recursivo

```
FUNCIÓN Fibonacci(numero)
    SI numero < 0 ENTONCES
        IMPRIMIR "El numero no puede ser negativo"
    SI NO
        SI (numero = 1) OR (numero = 2) ENTONCES
            Fibonacci ← 1
        SI NO
            Fibonacci ← Fibonacci(numero - 1) + Fibonacci(numero - 2)
        FIN SI
    FIN SI
FIN FUNCIÓN
```

Código en C

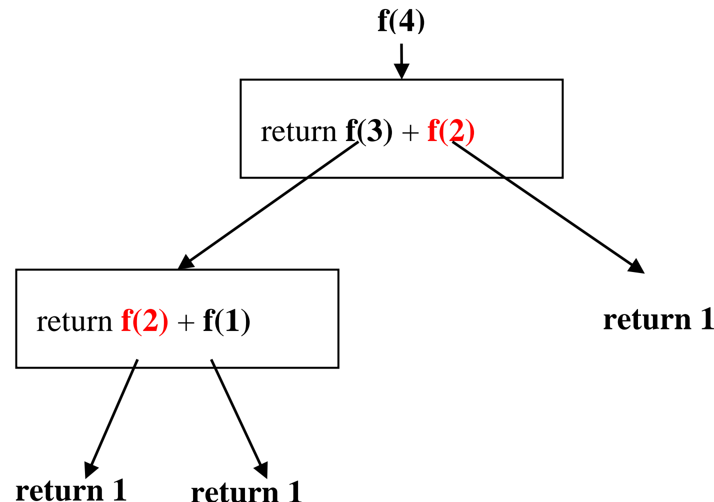
```
#include <stdio.h>

int fibonacci(int);
main()
{
   int n;
   printf("Entre el termino que desea obtener de la serie de Fibonacci: \n");
```

```
   scanf("%d", &n);
   if (n >= 0)
   {
      int f = fibonacci(n);
      printf("El termino %d de la serie de Fibonacci es: %d \n", n, f);
   }
   else
      printf("Lo sentimos, el numero debe ser mayor que cero");
}

int fibonacci(int n)
{
   if ((n == 1) || ( n == 2))
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Cada vez que se invoca a Fibonacci(), de inmediato prueba si se cumple el caso base (**n** = 1 o **n** = 2). Si esto es verdadero, se devuelve **n**. Si **n** es mayor que 1, el paso recursivo genera dos llamadas recursivas. En la siguiente figura se muestra como se evaluaría Fibonacci(4), sólo abreviamos Fibonacci con una **f**, a fin de hacer la figura más legible.



Esta figura plantea algunos temas interesantes en relación los programas recursivos como el que hemos analizado para generar números Fibonacci. Cada nivel e recursión en la función Fibonacci() tiene un efecto duplicador sobre el número de llamadas, es decir, el número de llamadas recursivas que se ejecutarán para calcular el número de Fibonacci de orden *n* será de 2^n. Esto se sale rápido de control. Simplemente calcular el veintavo número Fibonacci requeriría del orden de 2^{20} o aproximadamente de un millón de llamadas, y calcular el número Fibonacci del orden treintavo requeriría del orden de 2^{30} o aproximadamente de mil millones de llamadas, y así en lo sucesivo. Los científicos de la computación se refieren a lo anterior como complejidad exponencial. Además se realizan cálculos redundantes (ya que se vuelven a calcular innecesariamente términos de la serie ejemplo: el f(2), señalado

en rojo, se calcula dos veces). Problemas de esta naturaleza humillan incluso a las computadoras más poderosas del planeta. Los temas de complejidad, en general, y la complejidad exponencial en particular, se analizarán en detalle en asignaturas posteriores de la carrera.

Como la sucesión de Fibonacci se define recursivamente, parece natural que se escriba una función recursiva para calcular el término n-ésimo. Pero como se ha visto esta función recursiva trabaja pero tiene un serio problema de ineficiencia por los cálculos redundantes que realiza. Este ejemplo nos permite plantear la cuarta y última regla de la recursividad:

4. No redundancia. Nunca duplique el trabajo. No resuelva con una llamada recursiva por separado lo que ya ha sido resuelto en otra llamada.

7.3 Recursividad versus Iteración.

En las secciones anteriores hemos estudiados dos tipos de métodos que pueden con facilidad instaurarse, ya sea de forma recursiva o en forma iterativa. En esta sección compararemos los dos enfoques y analizaremos por qué el programador debe escoger un enfoque sobre el otro, en una situación en particular.

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura de repetición, la recursión utiliza una estructura de selección. Tanto la iteración como la recursión implican repetición: la iteración utiliza la estructura de repetición de forma explícita, la recursividad consigue la repetición mediante llamadas repetidas a la misma función. La recursividad y la iteración ambas involucran una prueba de terminación: la iteración termina cuando falla la condición de continuación del ciclo, la recursividad termina cuando se reconoce el caso base. Tanto la iteración como la recursividad pueden ocurrir en forma indefinida: ocurre un ciclo infinito en la iteración si la prueba de continuación del ciclo nunca se hace falsa, la recursividad infinita ocurre si el paso de recursión no reduce el problema en cada ocasión, de tal forma que converja al caso base.

La recursividad tiene muchas negativas. Invoca de forma repetida al mecanismo y, por lo tanto, a la sobrecarga de las llamadas de métodos. Esto puede resultar costoso tanto en tiempo de procesador como en espacio en memoria. Cada llamada recursiva genera otra copia de la función (de hecho sólo de las variables de la función), esto puede consumir gran cantidad de memoria. La iteración por lo regular ocurre dentro de una función, por lo que no ocurre la sobrecarga de llamadas repetidas de funciones y asignación extra de memoria. Por lo tanto ¿por qué elegir la recursividad?

Cualquier problema que puede ser resuelto en forma recursiva, también puede ser resuelto en forma iterativa (no recursiva). Normalmente se escoge un enfoque recursivo en preferencia a uno iterativo cuando el enfoque recursivo es más natural al problema y resulta en un programa que sea más fácil de comprender y de depurar. Otra razón para seleccionar una solución recursiva, es que la solución iterativa pudiera no resultar aparente.

8. Algoritmos de búsqueda.

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos. La variante más simple del problema es la búsqueda de un número en un arreglo unidimensional.

8.1 Búsqueda secuencial en un arreglo.

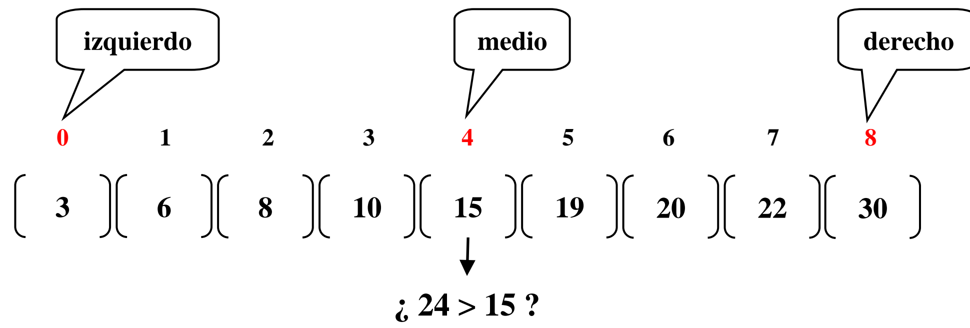
Este algoritmo se utiliza cuando el vector no está ordenado o no puede ser ordenado previamente. Consiste en buscar el elemento comparándolo secuencialmente (de ahí su nombre) con cada elemento del arreglo hasta encontrarlo, o hasta que se llegue al final. La existencia se puede asegurar cuando el elemento es localizado, pero no podemos asegurar la no existencia hasta no haber analizado todos los elementos del arreglo. A continuación se muestra una función *búsqueda* que implementa un algoritmo de búsqueda secuencial.

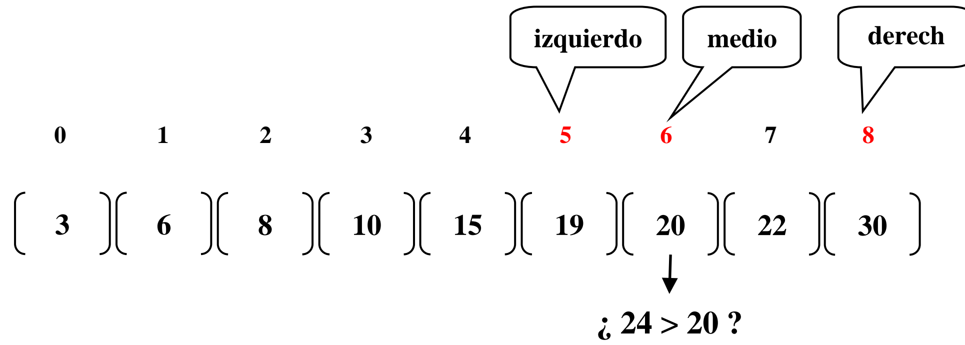
```
FUNCIÓN  Busqueda(numero, arreglo, tamaño)
        index ← 0
        DESDE index ← 0 HASTA tamaño - 1
                SI arreglo(index) = numero ENTONCES
                        Busqueda ← true
                FIN SI
        FIN DESDE
        Busqueda← false
FIN FUNCIÓN
```

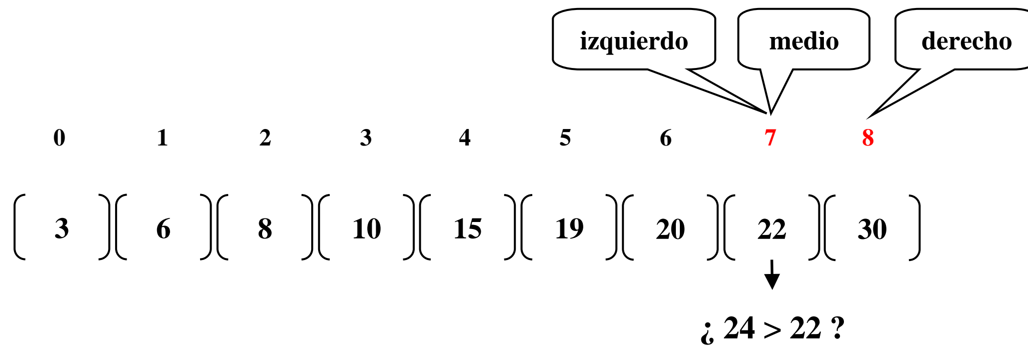
8.2 Búsqueda binaria en un arreglo.

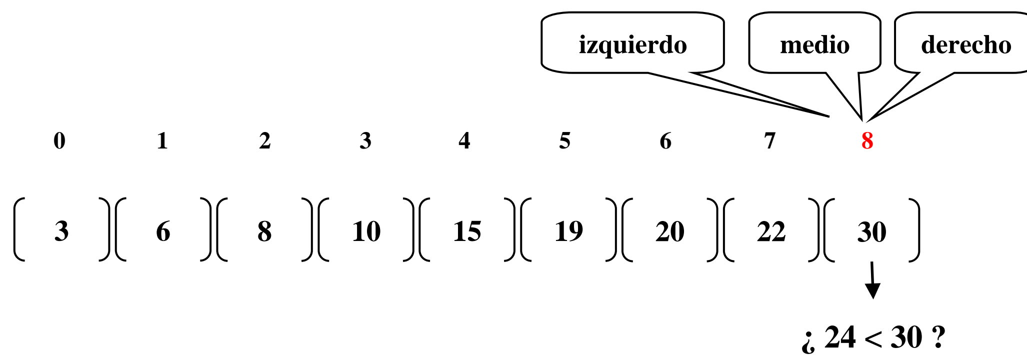
Este algoritmo se basa en comparar el elemento que se desea buscar con un elemento cualquiera del arreglo ordenado (normalmente el elemento del medio): si el valor de éste es mayor que el del elemento buscado se repite el procedimiento en la parte del arreglo que va desde el inicio de éste hasta el elemento tomado, en caso contrario se toma la parte del arreglo que va desde el elemento tomado hasta el final. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible. Si el elemento no se encuentra dentro de este último entonces se deduce que el elemento buscado no se encuentra en todo el arreglo. Sea por ejemplo un arreglo con los elementos {3, 6, 8, 10, 15, 19, 20, 22, 30} se quiere buscar si el elemento 24 está en el arreglo. Tomemos el elemento que está en la posición intermedia del arreglo (15 en este ejemplo), si el elemento a buscar es igual a éste la búsqueda habrá terminado, en caso contrario, como los elementos en el arreglo están ordenados, si el elemento a buscar es menor que el del medio entonces

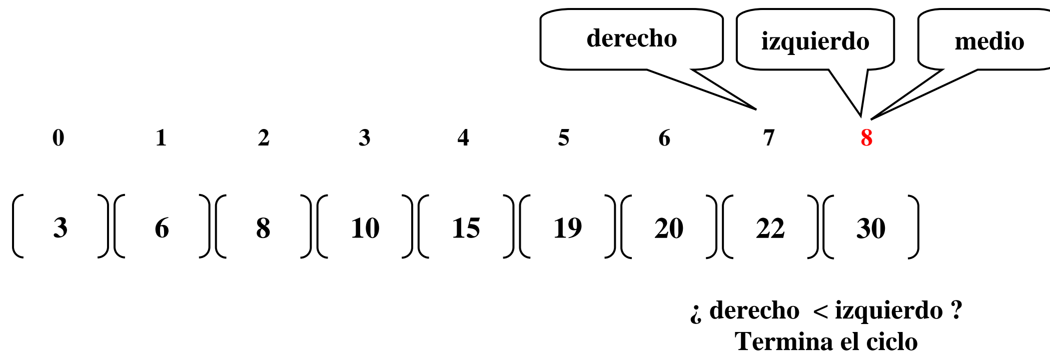
hay que buscar a la izquierda de éste y si es mayor a la derecha. En este caso 24 es mayor que 15 luego hay que buscar en {19, 20, 22, 30}. Con esta sección del arreglo se repite de nuevo el mismo procedimiento. Cuando la cantidad de elementos es par se considerará, de los dos posibles, que el del medio es el más a la izquierda (un razonamiento similar podría hacerse tomando el más a la derecha) por lo que ahora el elemento del medio será 20, como 24 es mayor que 20 se deberá buscar ahora en la sección {22, 30}. Ahora el del medio es 22 y como 24 es mayor que 22 entonces la próxima sección en la que buscar es {30}. El elemento medio de ésta es el propio 30 que no es igual 24, pero ya no se pueden seguir haciendo subdivisiones luego se puede concluir que el ¡24 no está! en el arreglo.


izquierdo
medio
derecho
0 1 2 3 4 5 6 7 8
3 6 8 10 15 19 20 22 30
¿ 24 > 15 ?


izquierdo
medio
derech
0 1 2 3 4 5 6 7 8
3 6 8 10 15 19 20 22 30
¿ 24 > 20 ?


izquierdo
medio
derecho
0 1 2 3 4 5 6 7 8
3 6 8 10 15 19 20 22 30
¿ 24 > 22 ?


izquierdo
medio
derecho
0 1 2 3 4 5 6 7 8
3 6 8 10 15 19 20 22 30
¿ 24 < 30 ?




```
FUNCIÓN BusquedaBinaria(numero, arreglo, tamaño)
    izquierdo ← 0
    derecho = tamaño - 1
    MIENTRAS izquierdo <= derecho
        medio ← (izquierdo + derecho) / 2
        SI arreglo(medio) = numero ENTONCES
            BusquedaBinaria ← true
        SI NO SI arreglo(medio) < numero
            izquierdo ← medio + 1
        SI NO
            derecho ← medio - 1
        FIN SI
    FIN MIENTRAS
    BusquedaBinaria ← false
FIN FUNCIÓN
```

9. Algoritmos de ordenamiento.

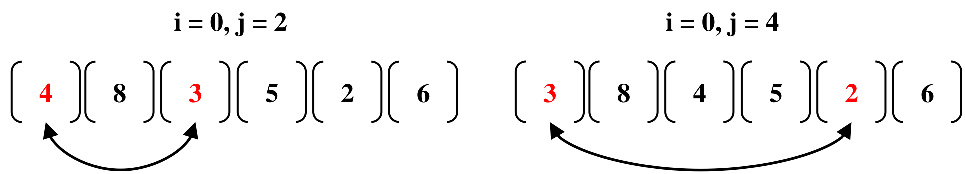
Un **algoritmo de ordenamiento** es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.
Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, El método de ordenamiento por burbuja fue analizado desde 1956. Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy.

9.1 Ordenamiento por mínimos sucesivos.

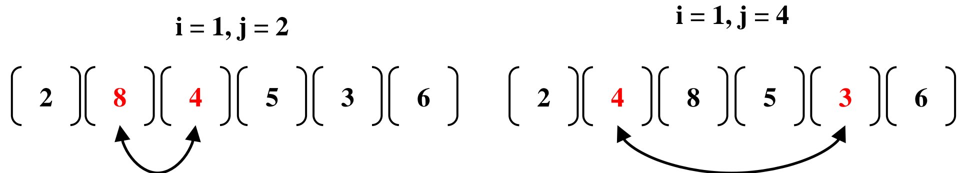
Tal vez el método de ordenamiento más simple es el que se conoce como mínimos sucesivos. Se hace un recorrido del arreglo y cada vez que se encuentre un elemento menor que el que está en la primera posición entonces se intercambia con éste. Esto garantiza que al finalizar el recorrido habrá quedado en la primera posición el menor de todos los elementos del arreglo. Se vuelve a repetir un proceso similar de recorrido pero ahora empezando en la segunda posición, al finalizar este segundo recorrido queda en la segunda posición el siguiente menor. Se realiza un recorrido similar empezando en cada una de las posiciones hasta un último recorrido que será el que empiece en el penúltimo elemento. Después de hacer todos los recorridos el arreglo habrá quedado ordenado de menor a mayor. La siguiente figura ilustra los diferentes cambios para ordenar un arreglo con los elementos {4, 8, 3, 5, 2, 6}.

```
FUNCIÓN OrdenarMinimosSucesivos(arreglo, tamaño)
        DESDE i ← 0 HASTA tamaño – 1
                DESDE j ← i + 1 HASTA tamaño – 1
                        SI arreglo(j) < arreglo(i) ENTONCES
                                temp ← arreglo(i)
                                arreglo(i) ← arreglo(j)
                                arreglo(j) ←  temp
                        FIN SI
                FIN DESDE
        FIN DESDE
FIN FUNCIÓN
```

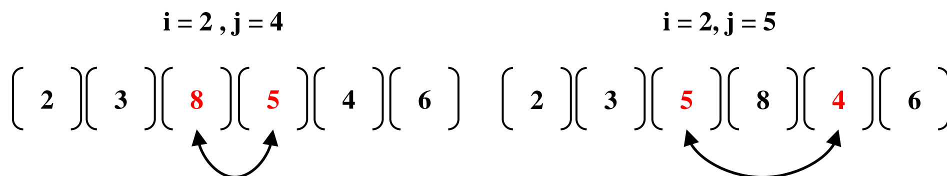
Intercambios para i = 0, a = {4, 8, 3, 5, 2, 6}



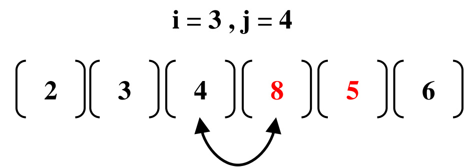
Intercambios para i = 1, a = {2, 8, 4, 5, 3, 6}



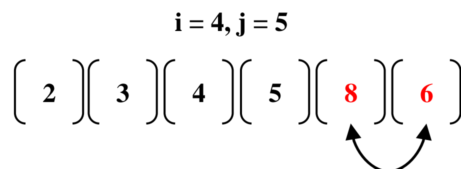
Intercambios para i = 2, a = {2, 3, 8, 5, 4, 6}



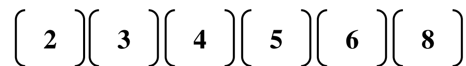
Intercambios para i = 3, a = {2, 3, 4, 8, 5, 6}



Intercambios para i = 4, a = {2, 3, 4, 5, 8, 6}



Arreglo ordenado.



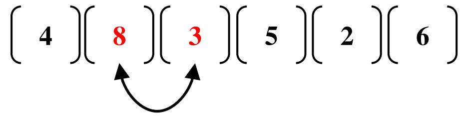
9.2 Ordenamiento por burbuja o por comparación de pares consecutivos.

El método de ordenamiento conocido como ordenamiento burbuja (bubble sort) consiste en recorrer todos los elementos del arreglo comparando cada elemento con el siguiente e intercambiarlos si el siguiente es menor, de ahí también el nombre de ordenamiento por comparación de pares consecutivos. Este proceso de recorrido se repite hasta que en un recorrido no se haya hecho intercambio alguno, lo que significa que ya habrán quedado ordenados. El ordenamiento del arreglo {4, 8, 3, 5, 2, 6} se ilustra en la siguiente figura.

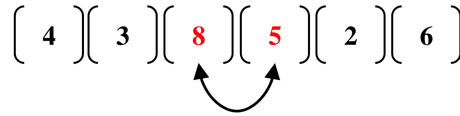
```
FUNCIÓN OrdenarPorBurbuja(arreglo, tamaño)
        HACER
                huboCambio ← false
                DESDE  i ← 0 HASTA tamaño - 2
                        SI arreglo(i+1) < arreglo(i) ENTONCES
                                temp ← a(i)
                                arreglo (i) ←  arreglo (i + 1)
                                arreglo (i + 1) ←  temp
                                huboCambio ←  true
                        FIN SI
                FIN DESDE
        MIENTRAS (huboCambio)
FIN FUNCIÓN
```

Intercambios para huboCambio = false, i = 0, a = {4, 8, 3, 5, 2, 6}

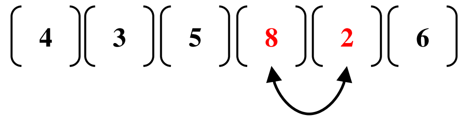
i = 1, huboCambio = false



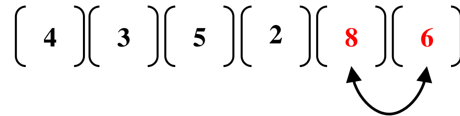
i = 2, huboCambio = true



i = 3, huboCambio = true

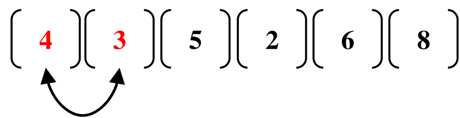


i = 4, huboCambio = true

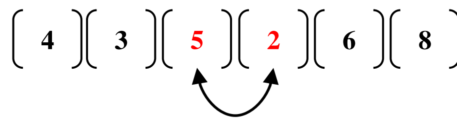


Intercambios para huboCambio = false, i = 0, a = {4, 3, 5, 2, 6, 8}

i = 0, huboCambio = true

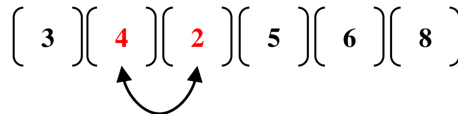


i = 2, huboCambio = true



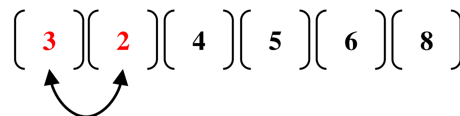
Intercambios para huboCambio = false, i = 0, a = {3, 4, 2, 5, 6, 8}

i = 1, huboCambio = false

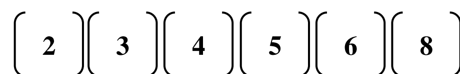


Intercambios para huboCambio = false, i = 0, a = {3, 2, 4, 5, 6, 8}

i = 0, huboCambio = false



Arreglo ordenado.



Conclusiones

En esta conferencia se estudiaron los arreglos en C y los aspectos más importantes de su manipulación, viéndose como un caso particular de éstos las cadenas de caracteres. Después de este estudio sabemos que:

- Los arreglos son los tipos de datos estructurados que permiten guardar colecciones de valores.
- Cada uno de los elementos de un arreglo tiene una posición específica, a través de la cual se le puede acceder. Esta posición se expresa mediante un subíndice que es un valor ordinal.
- La dimensión de un arreglo es la cantidad de subíndices necesarios para referenciar un elemento dentro del arreglo.
- Los algoritmos que recorren arreglos, por lo general utilizan variables que se incrementan/decrementan y se usan como subíndice.

Hasta ahora hemos visto problemas que se pueden resolver tanto con recursividad como con estructuras repetitivas. Es muy importante tener en cuenta que siempre que podamos emplear un algoritmo no recursivo será mejor (ocupa menos memoria RAM y se ejecuta más rápidamente). Pero hay casos donde el empleo de recursividad hace mucho más sencillo el algoritmo.

Con relación a los métodos de ordenamiento, cuando se trata de de arreglos ordenados, a partir de una determinada longitud es recomendable usar la búsqueda binaria en lugar de la búsqueda secuencial, debido a que se reduce el tiempo de búsqueda considerablemente, al disminuir exponencialmente el número de iteraciones necesarias.

En el método de ordenación por mínimos sucesivos, la repetición de los dos ciclos no depende de que se hagan o no intercambios, en cambio, en el ordenamiento por burbuja, el proceso de recorrido se repite hasta que en un recorrido no se haya hecho intercambio alguno, lo que significa que ya habrán quedado ordenados. En el peor caso, para arreglos originalmente desordenados la eficacia es la misma en ambos algoritmos, pero en la medida en que los elementos del arreglo original aparezcan bastante ordenados el algoritmo de ordenamiento por burbuja hará menos recorridos.