UNIDAD 2

PROGRAMACIÓN IMPERATIVA

ÍNDICE

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Unidad 2

1. Búsqueda y ordenamiento

Los algoritmos de búsqueda y ordenamiento son técnicas fundamentales en programación y ciencias de la computación que se utilizan para manipular y gestionar datos. En esta unidad analizaremos los algoritmos más conocidos.

1.1. Algoritmos de Búsqueda

Refieren a métodos utilizados para encontrar un elemento específico dentro de una estructura de datos, como un arreglo, lista, árbol, o grafo. Y pueden clasificarse del siguiente modo:

- ✓ Búsqueda Lineal o Secuencial.
- ✓ Búsqueda Binaria
- ✓ Búsqueda por Hashing
- ✓ Búsqueda de Interpolación
- ✓ Búsqueda Exponencial
- ✓ Búsqueda en Árboles (e.g., Búsqueda en Árbol Binario de Búsqueda)
- ✓ Búsqueda en Grafos (e.g., BFS, DFS)

En Programación Imperativa veremos la Búsqueda Lineal o Secuencial Desordenada, Búsqueda Lineal o Secuencial Ordenada y la Búsqueda Binaria.

Búsqueda Lineal o Secuencial

En este caso se recorre la estructura elemento por elemento hasta encontrar el objetivo.

1.1.1. Búsqueda Lineal o Secuencial Desordenada

Dado un elemento a buscar, se realiza la búsqueda comparando con cada elemento de la estructura, hasta llegar al último.

Supongamos un arreglo con 10 números desordenados donde necesitamos saber en qué posición está el *33*.

Con este algoritmo empezamos a recorrer desde el índice 0 en adelante, comparando cada elemento con el número *33.* Si se encuentra, se detiene la búsqueda en el punto en que se encontró. Si no se encuentra, se habrá recorrido todo el arreglo sin éxito.

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Es decir que este algoritmo es bueno cuando el elemento está entre las primeras posiciones, pero es poco eficiente cuando está entre las últimas.

```
int busquedaSecuencialDesordenada(int arreglo[], int
dimension, int buscado)
{   int i = 0;
    while (i < dimension && arreglo[i] != buscado)
        i++;
    if (i < dimension)
        return i;
    else
        return -1; }
```

Link al simulador

Si encuentra el elemento buscado, se retorna la posición en donde se encontró y, si no lo encuentra, se retorna-1

Ilustración 1

En la *Ilustración 1* se muestra el ejemplo del mejor de los casos, el elemento buscado estará al principio y sólo se necesitará una comparación. Ahora en la siguiente ilustración, el caso es diferente:

Ilustración 2

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En el peor de los casos (*Ilustración 2*), el elemento está al final (o no está) y se necesitarán diez comparaciones para encontrarlo.

Para el procesador, la diferencia entre realizar 1 o 10 operaciones es mínima, entonces, la búsqueda secuencial es buena opción cuando hay pocos elementos *(cualquier procesamiento que tarde menos de 200 ms es percibido como instantáneo por el ser humano)*.

Pero imaginemos una estructura con miles de elementos. O vayamos más allá e imaginemos una base de datos de una red social con millones de datos. Es imprescindible que el procesamiento sea lo más rápido posible o habría muchos usuarios enojados…

Por lo tanto, para grandes volúmenes de datos, es preferible emplear otros algoritmos de búsqueda.

1.1.2. Búsqueda Lineal o Secuencial Ordenada

En este caso la búsqueda comienza desde el principio y se avanza por la estructura de manera secuencial hasta que se encuentra el elemento buscado o hasta que se encuentra un elemento mayor al buscado (suponiendo que están ordenados de menor a mayor).

```
int busquedaSecuencialOrdenada(int arreglo[], int dimension,
int buscado)
{   int i = 0;
    while (i < dimension && arreglo[i] < buscado)
        i++;
    if (i < dimension && arreglo[i] == buscado)
        return i;
    else
        return -1;}
```

Link al simulador

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

La **eficiencia** respecto a la búsqueda desordenada es mejor para el caso en que el elemento buscado no se encuentra en la estructura. Por ejemplo, si buscáramos el número 28 (en el ejemplo de la *Ilustración 3*), al llegar al número 31, y como sabemos que están en orden, ya podemos determinar que el 28 no se va a encontrar más adelante.

Ilustración 3

Esto mejora la eficiencia cuando el elemento buscado no está. Pero, si está, el mejor y peor caso tienen igual eficiencia que en la búsqueda desordenada.

1.1.3. Búsqueda Binaria

Para utilizar este algoritmo la estructura debe estar ordenada de manera creciente o decreciente. Entonces, se compara en primer lugar con el elemento central de la estructura y, si no es el valor buscado, se reduce el intervalo de búsqueda a la mitad derecha o izquierda (según donde pueda encontrarse el valor buscado).

El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda llega a cero.

En los casos en que la estructura esté ordenada y contenga varias ocurrencias del valor buscado, la búsqueda binaria puede devolver cualquiera de las posiciones donde se encuentre ese valor. No se garantiza cuál devolverá, ya que depende del recorrido realizado durante la búsqueda. Si los valores están repetidos, sus posiciones serán consecutivas debido al orden.

Supongamos que buscamos el número 31 en un arreglo de 10 elementos.

Ilustración 4

El primer paso será buscar la mitad del arreglo y para eso se calcula

medio = (principio+fin)/2

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

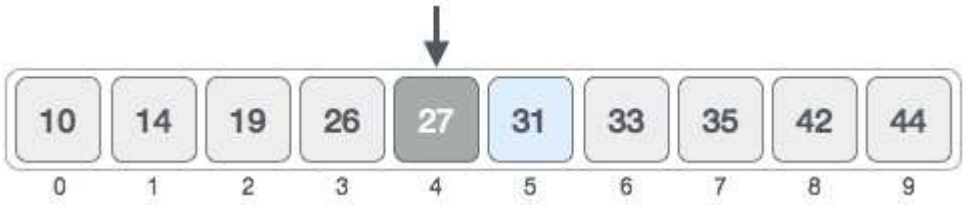


Ilustración 5

Ahora se compara el valor del medio (27) con el buscado (31). Como no coinciden, pero el valor buscado es mayor que el encontrado en el medio, se sabe que no se va a encontrar en la mitad inferior, entonces nos quedamos con la superior (*Ilustración 6*).

Ilustración 6

Ahora cambiamos nuestro principio, porque ya no nos interesa buscar desde el elemento en la posición 0. Por eso ahora principio = medio + 1 y se vuelve a calcular el medio: medio = (principio + fin) / 2 que esta vez será la posición 7.

Ilustración 7

A continuación, se comparará el número en el medio con el buscado. Como es mayor, entonces el valor tiene que estar en la mitad inferior, así que cambiamos nuestro final: fin = medio – 1

Ilustración 8

Volvemos a calcular el medio, que esta vez será la posición 5:

medio = (principio + fin) / 2

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

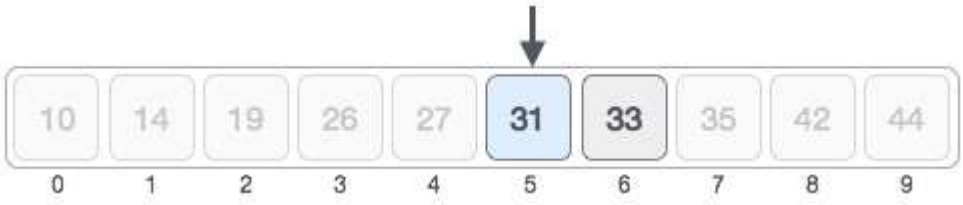


Ilustración 9

Comparamos el elemento en esa posición con el que buscamos y, efectivamente, coinciden. Entonces concluimos que el número 31 está en el arreglo y se encuentra en la posición 5.

Veámoslo en C++:

```
int busquedaBinaria(int arreglo[], int dimension, int buscado)
{
    int principio = 0;
    int fin = dimension-1;
    while (principio <= fin) {
        int medio = (principio + fin) / 2;
        if (arreglo[medio] == buscado) {
            return medio;
        }
        else {
            if (arreglo[medio] > buscado) {
                fin = medio - 1;
            }
            else {
                principio = medio + 1;
            }
        }
    }
    return -1;
}
```

Link al simulador

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Aunque su implementación es un poco más compleja, la búsqueda binaria puede ser mucho más eficiente que la secuencial, especialmente cuando hay muchos elementos en la estructura. Incluso si tomamos el peor de los casos (que haya que dividir el arreglo en dos la máxima cantidad de veces), la cantidad de operaciones es menor que el caso promedio de la búsqueda secuencial.

Cantidad de comparaciones hasta encontrar el valor buscado		
Elementos en el arreglo	Caso promedio de B. secuencial	Peor caso de B. Binaria
8	4	4
128	64	8
256	128	9
1000	500	11
100000	50000	18

Tabla 1

1.2. Algoritmos de Ordenamiento

Un algoritmo de ordenamiento es un algoritmo que pone elementos de un arreglo en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación o reordenamiento, de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico.

Los ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como el de búsqueda binaria, por ejemplo) que requieren arreglos ordenados para una ejecución rápida.

También es útil para generar resultados legibles por humanos.

Para poder ordenar una cantidad determinada de números almacenadas en un arreglo existen distintos métodos (algoritmos) con distintas características y complejidades.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Desde los métodos más simples, como el Método Burbuja, que consiste en simples iteraciones, hasta el Ordenamiento Rápido (o QuickSort) que, al estar optimizado usando recursión, su tiempo de ejecución es menor y es más efectivo.

Entonces podríamos identificar métodos de ordenamiento de dos clases:

1. los iterativos y
2. los recursivos (dejaremos estos últimos para más adelante).

Los iterativos son métodos fáciles de comprender y de programar ya que son repetitivos, utilizan ciclos simples, así como sentencias que hacen que la estructura resulte ordenada.

Dentro de los algoritmos iterativos podemos encontrar:

- Selección
- Burbuja
- Inserción

1.2.1. Ordenamiento por Selección

Es un algoritmo simple, en el que, si la estructura tiene N elementos, este método hace N-1 “pasadas” sobre la misma. Luego, el rango de recorrido, en cada pasada, se hace más chico.

La estructura se divide en dos partes, quedando el sector ordenado a la izquierda y el que resta ordenar a la derecha. En un principio el sector ordenado no tiene elementos y el sector desordenado es toda la estructura.

Se selecciona el elemento más chico del sector desordenado y se envía a la izquierda, al ordenado. Se llama "selección" porque "selecciona" sucesivamente el menor elemento que falta por ordenar.

Supongamos el siguiente arreglo a ordenar:

Ilustración 10

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Lo primero que se hace es buscar por todo el arreglo hasta encontrar el menor elemento. *En este caso es el 10* (ver *Ilustración 11*).

Ilustración 11

Entonces el 10 se coloca en la primera posición, intercambiándose con el 14, y así empieza a formarse el sector ordenado del arreglo (ver *Ilustración 12*).

Ilustración 12

Ahora continuamos buscando en el sector desordenado del arreglo, hasta encontrar el menor elemento. *En esta oportunidad es el 14* (ver *Ilustración 13*).

Ilustración 13

Se intercambia el 14 con el 33, que estaba más a la izquierda en el sector desordenado. El 14 pasa a estar en el sector ordenado.

Ilustración 14

De la misma manera, se continúa buscando siempre al menor dentro del sector desordenado y moviéndolo hacia el sector ordenado, hasta terminar de ordenarlos a todos.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

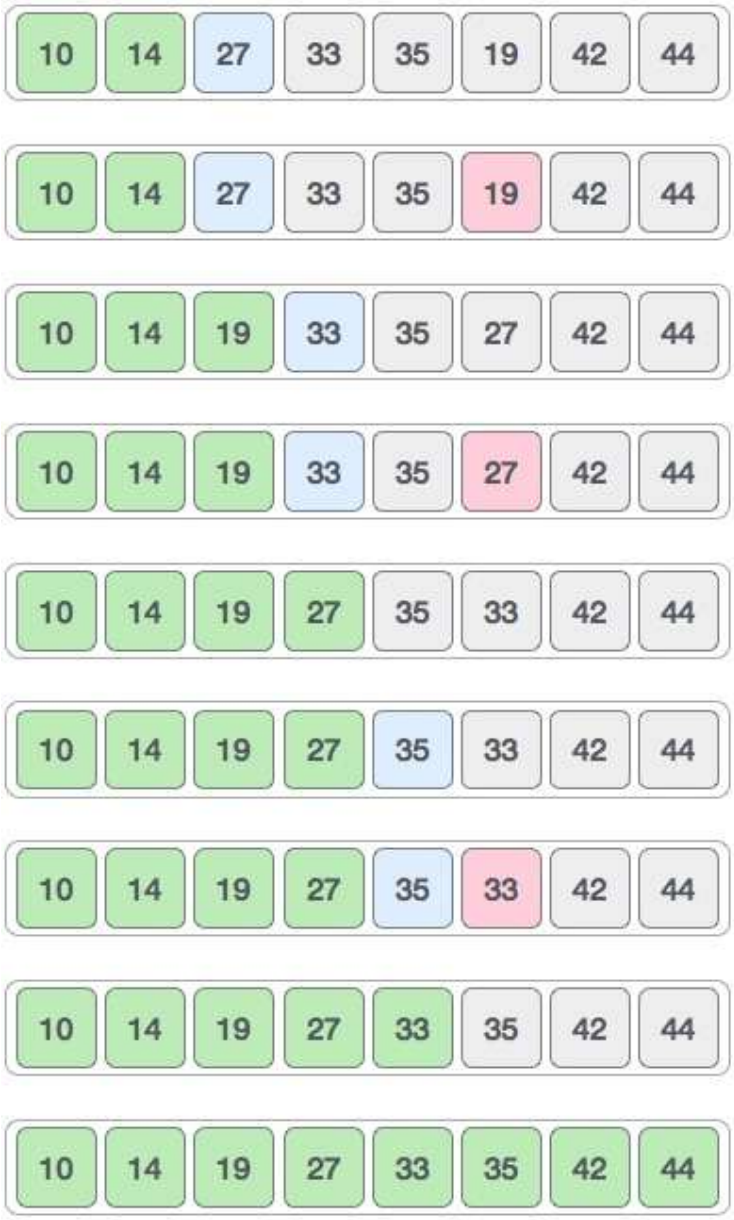


Ilustración 15

En resumen, lo que el algoritmo hace es buscar el menor elemento en el arreglo. Reducir el espacio dentro del cual se busca y volver a buscar el menor elemento. Repetir hasta terminar de reducir el espacio de búsqueda a 0.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En C++, el algoritmo sería:

```
void seleccion(int arreglo[], int dimension)
{
    for (int i = 0; i < dimension-1; i++) {
        int menor = i;
        for (int j = i+1; j<dimension; j++) {
            if (arreglo[j] < arreglo[menor]) {
                menor = j;
            }
        }
        if (i != menor) {
            int temp = arreglo[i];
            arreglo[i] = arreglo[menor];
            arreglo[menor] = temp;
        }
    }
}
```

Link al simulador

Este algoritmo no es eficiente para estructuras con gran cantidad de elementos. Para arreglos con muy pocos elementos (10, 20), podría usarse tanto selección como inserción.

Pero, como constantemente se está recorriendo todo el sector desordenado sin importar en qué posición esté el menor elemento, el mejor y el peor caso tienen la misma complejidad.

Si la cantidad de elementos de la estructura es N, este algoritmo realiza N-1 pasadas.

1.2.2. Burbuja

El algoritmo de ordenamiento conocido como "burbuja" recibe su nombre en alusión a la forma en que las burbujas de una bebida ascienden hacia la superficie.

Este método funciona comparando pares de elementos adyacentes y, si no están en el orden deseado, se intercambian. Este proceso se repite, permitiendo ordenar los elementos de manera ascendente o descendente según se necesite.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Una de sus características distintivas es que, si la estructura ya está ordenada, solo se requiere una pasada, lo que lo diferencia del algoritmo de selección, que siempre realiza N-1 pasadas independientemente del estado de orden de los elementos.

Sin embargo, este método no es eficiente para grandes volúmenes de datos debido a su complejidad temporal elevada, que puede resultar en tiempos de ejecución prolongados en dichos escenarios.

Supongamos un arreglo a ordenar de manera ascendente:

Ilustración 16

Inicialmente se toman los elementos en las dos primeras posiciones (14 y 33) y se comparan entre sí. Como ya están ordenados (14 es menor que 33) no se modifica nada.

Ilustración 17

Se pasa al siguiente elemento, comparando ahora 33 y 27. Como 27 es menor que 33, entonces se deben intercambiar.

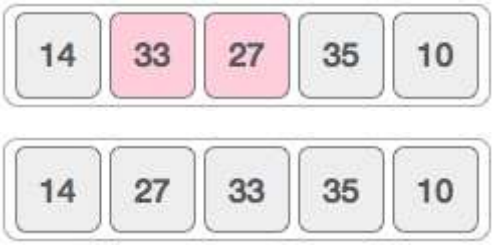


Ilustración 18

A continuación, se toma el siguiente elemento y se comparan 33 y 35, que están ya ordenados, por lo que no cambia nada.

Ilustración 19

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Luego, se comparan el 35 y el 10, y se intercambian por estar desordenados:

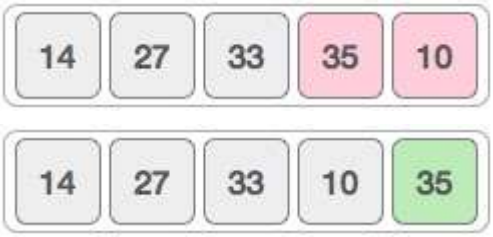


Ilustración 20

Se finalizó la primera pasada y llegamos al final del arreglo. Ahora se realiza una segunda pasada y obtenemos:

Ilustración 21

Tras una tercera iteración por todo el arreglo, se obtiene:

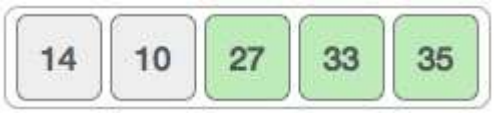


Ilustración 22

Finalmente, el arreglo queda ordenado y se finalizan las iteraciones realizadas sobre él:

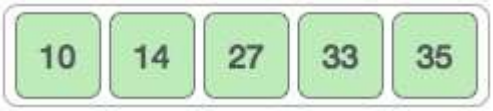


Ilustración 23

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Este algoritmo podría resumirse como:

Tomar dos elementos. Si el primero es mayor que el segundo, intercambiarlos. Pasar al siguiente par de elementos y repetir hasta llegar al sector ordenado.

En C++, el algoritmo sería:

```
void burbujeo(int arreglo[], int dimension)
{
    for (int i = 0; i < dimension-1; i++) {
        for (int j = 0; j < dimension-i-1; j++) {
            if (arreglo[j] > arreglo[j+1]) {
                int temp = arreglo[j];
                arreglo[j] = arreglo[j+1];
                arreglo[j+1] = temp;
            }
        }
    }
}
```

Link al simulador

En el peor caso, la estructura estará ordenada de forma inversa a la que se desea y se deben realizar N^2 comparaciones (siendo N la cantidad de elementos en la estructura).

En el mejor de los casos la estructura estará ya ordenada de la manera en que se desea, pero igualmente este algoritmo estará realizando N comparaciones.

Es por esto que se lo considera un algoritmo ineficiente. Sólo es útil cuando hay muy pocos elementos.

Se podría optimizar cortando el recorrido cuando se descubre que el arreglo ya ha quedado ordenado.

1.2.3. Ordenamiento por Inserción

Este método de ordenamiento ascendente funciona separando una sección de la estructura, la cual se mantiene ordenada a medida que se procesa. Este proceso generalmente comienza con la parte inferior del arreglo. A medida que se recorre el arreglo en busca de elementos desordenados, estos se insertan en la sección ordenada,

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

asegurando que esta última se expanda progresivamente hasta abarcar toda la estructura original, resultando en un arreglo completamente ordenado de menor a mayor.

Supongamos el siguiente arreglo a ordenar:

Ilustración 24

Se comparan los dos primeros elementos

Ilustración 25

Como están ya en orden, entonces el 14 pasa a estar en el sector ordenado. Ahora se comparan el 33 y el 27.

Ilustración 26

Como no están ordenados, entonces se intercambian y el 27 pasa a estar en el sector ordenado.

Ilustración 27

Se continúa tomando el siguiente par (33 y 10) y se intercambian sus valores, por no estar ordenados.

Ilustración 28

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

El 10 ahora pasa al sector ordenado.

Ilustración 29

Pero este intercambio hace que el 27 y el 10 estén desordenados.

Ilustración 30

Entonces, se intercambian 27 y 10.

Ilustración 31

Nuevamente, el 14 y el 10 están desordenados, entonces se intercambian.

Ilustración 32

Al final de esta iteración, tenemos un sub-arreglo (el sector ordenado) que está en el orden deseado.

Ilustración 33

El proceso continúa hasta que todos los valores estén ordenados.

Este algoritmo podría resumirse como: *comparar cada elemento con el anterior y colocarlo en orden. Repetir para el resto de los elementos.*

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En C++ podría ser del siguiente modo:

```
void insercion(int arreglo[], int dimension)
{
    for (int i = 1; i < dimension; i++) {
        int temp = arreglo[i];
        int j = i-1;
        while (j >= 0 && temp < arreglo[j]) {
           arreglo[j+1] = arreglo[j]; j--;
        }
        arreglo[j+1] = temp;
    }
}
```

Link al simulador

Este algoritmo es apropiado cuando los datos están casi ordenados o cuando la cantidad de elementos es pequeña.

El mejor de los casos se da cuando el arreglo está ya ordenado (realiza N comparaciones). El peor de los casos es cuando los elementos están ordenados de forma descendente y se deben invertir (se realizan N^2 comparaciones).

Se suele relacionar el ordenamiento por inserción con el reparto de naipes en un juego: a medida que se reciben los naipes de una mano, se van insertando en la posición correspondiente para mantenerlos siempre en orden.

Estos son sólo algunos de los algoritmos de búsqueda y de ordenamiento que normalmente se estudian. Existen optimizaciones para algunos de ellos y también otros algoritmos más eficientes, pero más complejos.

Por ejemplo, puede accederse a información y visualizaciones de varios algoritmos de ordenamiento en el siguiente sitio web: www.sorting-algorithms.com

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

2. Struct

En el lenguaje de programación C++, el programador tiene la capacidad de definir un nuevo

de dato mediante la utilización de estructuras, conocidas como “Structs”, en las cuales se asocian diversos datos que están lógicamente relacionados bajo un único nombre. Este enfoque es particularmente útil cuando es necesario agrupar múltiples datos que tienen sentido al estar juntos, como, por ejemplo, la información personal de un empleado de una empresa.

Una vez definido este nuevo tipo de dato, es posible declarar variables de dicho tipo e incluso almacenar datos de este nuevo tipo dentro de contenedores, como arreglos.

Las struct son llamadas también registros o record, y tienen muchos aspectos en común con los registros usados en bases de datos.

Permiten representar permiten representar datos complejos del mundo real, que, en general, son más complejos que un número entero.

2.1. Características de los Struct

Esta estructura es una colección de valores, pudiendo estos ser de diferentes tipos, por lo que se trata de una estructura de datos **heterogénea.**

En cuanto al **espacio de memoria**, un registro es fijo, por lo tanto, se la clasifica como una **estructura estática**.

Otra característica fundamental es que el **acceso** a sus campos es **directo**.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Analicemos un ejemplo concreto:

```
#include <iostream>

using namespace std;
// Definimos una estructura heterogénea, con tres campos de distintos tipos
struct Persona {
   string nombre;
   int edad;
   float altura;
};

int main() {
   // Creamos una instancia de la estructura
   Persona persona1;

   // Accedemos en forma directa a los campos de la estructura y asignamos valores
   persona1.nombre = "Juan Pérez";
   persona1.edad = 30;
   persona1.altura = 1.75f;

   // Imprimimos los valores de los campos de la estructura
   cout << "Nombre: " << persona1.nombre << endl;
   cout << "Edad: " << persona1.edad << endl;
   cout << "Altura: " << persona1.altura << " metros" << endl;

   return 0;
}
```

En el código anterior se puede observar que se trata de una ***Estructura de datos heterogénea***, dado que contiene un string (que almacenará el nombre), un int (para la edad) y un float (para almacenar la altura), demostrando que puede contener valores de diferentes tipos.

Además, se trata de una ***Estructura estática***, porque la memoria para Persona es asignada en el momento de la declaración de persona1, y la cantidad de memoria asignada no cambia durante la ejecución del programa. Esto demuestra la naturaleza estática de la struct.

Por último, el acceso a sus campos es directo. Los campos de la estructura Persona (nombre, edad y altura) son accedidos directamente utilizando el operador punto (.) y los valores son asignados e impresos directamente.

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

2.2. Definición de los Struct

Esta estructura se define indicando la palabra reservada struct, el nombre de la estructura, y el tipo y nombre de los campos que componen la estructura.

```
struct identificador
{
   tipo campo1;            no existe restricción para el tipo de
   tipo campo2;            los campos
   ...
}
```

Veamos un ejemplo en C++:

```
struct Persona
{
   string nombre;
   int dni;
   string direccion;
};
```

Cada campo puede tener un tipo diferente, que puede ser cualquier tipo de dato existente (int, string, bool...), incluso otras structs, contenedores, etc.

Una vez definida una struct, hemos creado un nuevo tipo de dato, que nos permitirá declarar variables de ese tipo.

En una variable de un tipo definido mediante struct, podemos acceder a cualquiera de sus campos sin necesidad de recorrer el resto de ellos, dado que el acceso es directo.

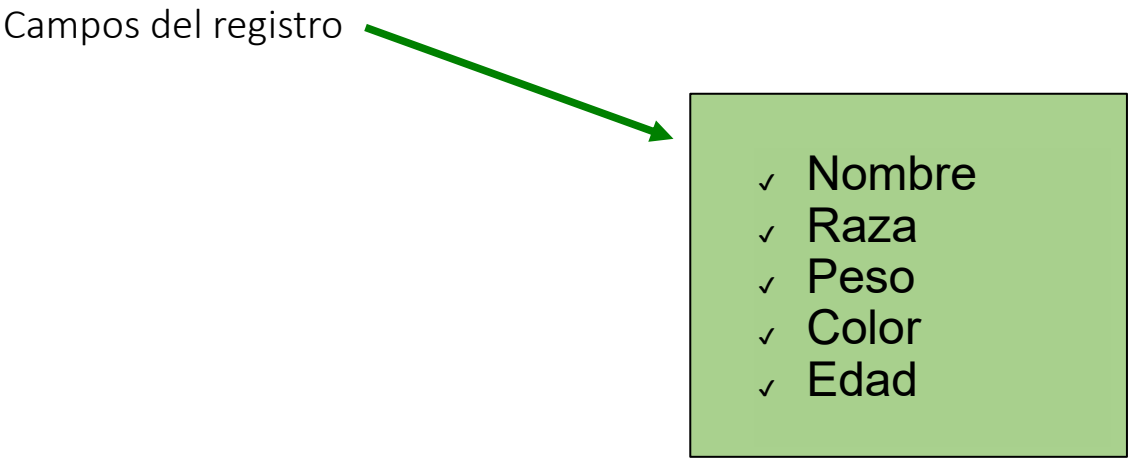
Cada campo puede tratarse individualmente como un dato del tipo que le corresponda (por ejemplo, un campo de tipo int podría utilizarse dentro de una operación matemática).

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Supongamos que queremos representar los datos de un perro, entonces deberíamos almacenar de algún modo el nombre, la raza, el peso, el color, la edad, etc.

Una manera natural y lógica de agrupar la información de cada perro en una sola estructura es declarar un tipo struct asociado a todos los datos que nos interesa guardar de cada perro.

Cada dato que compone la estructura es un campo de la struct, pudiendo definir al tipo struct del perro del siguiente modo:



Los datos que almacenaremos dependerán del problema a resolver.

Veamos otro ejemplo en el que se desea almacenar datos de personas.

```
struct Persona
{   string nombre;
   int dni;
   string direccion;   };
```

Ahora, definida la struct Persona, hemos creado un nuevo tipo de dato, que nos permitirá declarar variables de ese tipo. A continuación, declaramos dos variables de tipo Persona:

```
Persona una_persona;
Persona otra_persona;
```


Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Para referenciar a cada campo de la estructura se utiliza el operador de selección (.) que se coloca entre el nombre de la variable y el nombre del campo:

```
una_persona.dni = 29568425;
```

2.2.1. Declaración de la struct y variables, en diferentes ámbitos.

Una struct puede declararse en diferentes ámbitos.

Si necesitamos que el tipo de dato que estamos definiendo exista en todo el programa, lo correcto será definirlo en el ámbito global.

```
struct Persona {
    string nombre;
    int dni;
    string direccion;};
void funcion(Persona dato) {
    //cuerpo de la función}
int main() {
    Persona una_persona;
    //resto de la función main}
```


Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Al igual que con cualquier otro tipo de variables, se considera una mala práctica declarar variables de un tipo de struct en el ámbito global.

```
struct Persona {
    string nombre;
    int dni;
    string direccion;};
Persona una_persona;
void funcion() {
    //cuerpo de la función}

int main() {
    //cuerpo de la función main}
```

Ámbito global

2.3. Asignación

La asignación de los campos en un struct en C++ se refiere al proceso de inicializar o modificar los valores almacenados en los miembros de la estructura. Los campos de un struct pueden ser accedidos directamente utilizando el operador punto (.). Esto permite tanto la lectura como la escritura de los valores de esos campos.

A continuación, se asignan valores a cada uno de los campos de la variable "una_persona" (campo a campo):

```
una_persona.direccion = "Gral. Paz 179";
una_persona.dni = 29568425;
una_persona.nombre = "Maria";
```

En el siguiente ejemplo, se lo hace en una misma línea:

```
una_persona = {"Claudio", 53453455, "Alsina 43"};
```


Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En el siguiente ejemplo, se asignan los mismos valores de cada uno de los campos de la variable “una_persona” a la variable “otra_persona”:

```
otra_persona = una_persona;
```

Es importante recordar que el nombre de la struct creada es ahora el nombre de un tipo de dato, no de una variable. Entonces, no es posible seleccionar un campo utilizando el nombre de la struct como se observa en el siguiente código:

```
Persona.dni = 29568425;
```

El error radica en que Persona es el nombre de un tipo de dato, no el nombre de una variable.

2.4. Comparación

Las operaciones de comparación no están definidas para la estructura completa, pero sí se pueden comparar los campos de una struct:

```
if (una_persona.dni < otra_persona.dni)
    //bloque if
```

En el código anterior se comparan los campos dni de las variables una_persona y otra_persona.

Pero en el siguiente ejemplo arroja un error indicando que la operación == no puede aplicarse entre dos variables de tipo Persona:

```
if (una_persona == otra_persona)
    //bloque if
```


Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

2.5. Impresión

De igual manera, no es posible imprimir los datos de una struct completa con "cout", sino que deben individualizarse los campos a mostrar:

```
cout << una_persona.nombre;
cout << una_persona.dni;
cout << una_persona.direccion;
```

Por lo tanto, lo siguiente no está permitido:

```
cout << una_persona;
```

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

2.6. Ejemplo completo de Struct

```
#include <iostream>
#include <string>
using namespace std;
// Definición de un struct llamado Perro, en el ámbito global
struct Perro {
   string nombre;
   string raza;
   float peso;
   string color;
   int edad;};
int main() {
   // Declaración de dos variables del tipo struct Perro, en el ámbito local
   Perro perro1;
   Perro perro2;
   // Asignación de valores a los campos de la variable perro1
   perro1.nombre = "Rex";
   perro1.raza = "Pastor Alemán";
   perro1.peso = 30.5;
   perro1.color = "Negro y marrón";
   perro1.edad = 5;
   // Asignación de valores a los campos de la variable perro2
   perro2.nombre = "Luna";
   perro2.raza = "Labrador";
   perro2.peso = 25.0;
   perro2.color = "Amarillo";
   perro2.edad = 3;
   // Mostrar la información de la variable perro1. Debe hacerse campo a campo
   cout << "Información del primer perro:" << endl;
   cout << "Nombre: " << perro1.nombre << endl;
   cout << "Raza: " << perro1.raza << endl;
   cout << "Peso: " << perro1.peso << " kg" << endl;
   cout << "Color: " << perro1.color << endl;
   cout << "Edad: " << perro1.edad << " años" << endl;
   cout << endl; // Salto de línea para separar la información de los dos perros
   // Mostrar la información de la variable perro2. Debe hacerse campo a campo
   cout << "Información del segundo perro:" << endl;
   cout << "Nombre: " << perro2.nombre << endl;
   cout << "Raza: " << perro2.raza << endl;
   cout << "Peso: " << perro2.peso << " kg" << endl;
   cout << "Color: " << perro2.color << endl;
   cout << "Edad: " << perro2.edad << " años" << endl;
   return 0;
}
```

Link al simulador

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Ahora:

¿Cómo haríamos si quisiéramos registrar que el perro1 tiene el doble de la edad del perro2?

¿Y si quisiéramos registrar que el perro2 tiene el mismo color que perro1?

¿Cuál o cuáles serían las líneas de código necesarias para registrar que el perro2 aumentó unos 2 kilos?

3. Arreglos de Structs

Mientras todos los elementos de un **arreglo estático** sean del mismo tipo (homogéneo), ese tipo puede ser cualquiera, incluidos los definidos por el programador. Así, se puede declarar un arreglo cuyos elementos sean **structs**.

Para referenciar a un campo determinado de una struct almacenada en un arreglo, se debe acceder al elemento correspondiente (mediante su índice) y utilizar el punto para individualizar el campo:

```
Persona estudiantes[5000];
int dimL = 0;
estudiantes [0].direccion = "Gral Paz 179";
estudiantes [0].dni = 29568425;
estudiantes [0].nombre = "María";
```

estudiantes[5000] es un arreglo estático con 5000 lugares para almacenar "algo". En este caso ese "algo" es de tipo **Persona**, siendo este último un **struct**.

Para el ejemplo anterior, en el primer lugar del arreglo (con índice 0) se almacenan los datos de María, por lo que faltaría incrementar la variable que representa la dimensión lógica del arreglo (dimL).

Es fundamental recordar que en un arreglo de structs, cada elemento es una struct. Por ende, si se accede a un elemento del arreglo, se estará obteniendo una struct entera o completa. Entonces, sólo se podrá acceder campo a campo.

Mediante la línea `estudiantes [0]` se obtiene una struct de tipo Persona (la que esté almacenada en la posición 0 del arreglo).

Y a través de la línea `estudiantes [0].dni` se obtiene un número de tipo int (el que esté almacenado en el campo dni de la struct en la posición 0 del arreglo).

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En cada caso sólo pueden realizarse las operaciones que el tipo permita. Por ejemplo, **no podría asignarse un dato de tipo int a una struct de tipo Persona**, del mismo modo en el que no puede almacenarse un dato de tipo float en una variable de tipo string.

Por lo tanto, `estudiantes [0] = 25` retornaría un error.

3.1. Arreglos de structs: Elementos y campos

Lo importante es, una vez obtenido un dato, saber con qué tipo de dato estamos trabajando y qué cosas es posible hacer con ese tipo. Ya no importa si el dato está dentro de un arreglo, si es parte de una struct o dónde está almacenado.

Si un elemento de un arreglo es de tipo Persona, podremos hacer con ese dato todo lo que se pueda hacer con un dato de tipo Persona. Si un campo de una struct es de tipo string, podremos hacer con ese dato todo lo que se pueda hacer con un dato de tipo string.

Podría pasarse una struct contenida en el arreglo como **argumento** en una llamada a una función que reciba una struct del mismo tipo (en este ejemplo, de tipo Persona):

```
int digitosDNI(Persona estudiante) {
    int digitos = 0;
    while (estudiante.dni != 0) {
        digitos++;
        estudiante.dni = estudiante.dni/10; }
    return digitos;
}
int main() {
    Persona estudiantes[5000];
    int dimL = 0;
    dimL = cargarEstudiantes(estudiantes, dimL);
    cout << "Cantidad de dígitos del DNI: " << digitosDNI(estudiantes[3]);
}
```

estudiantes[3] y **estudiante** son datos de tipo **Persona**

Como el **parámetro** de tipo Persona está pasado **por valor**, el dato original dentro del arreglo **no se modifica**.

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

Debido a que cada campo puede accederse de forma individual, también podría pasarse únicamente uno de los campos de la struct contenida en el arreglo (y en este caso sería lo óptimo, ya que la función sólo trabaja con un número):

```
int cantDigitos(int n) {
    int digitos = 0;
    while (n != 0) {
        n = n/10;
        digitos++;
    }
    return digitos;
}

int main() {
    Persona estudiantes[100];
    int dimL = 0;
    dimL = cargarEstudiantes(estudiantes, dimL);
    cout << "Cantidad de dígitos del DNI: " <<
cantDigitos(estudiantes[3].dni);
}
```

Esto ya lo vimos, cuando aprendimos sobre "operaciones con los campos" de una struct.

3.2. Arreglos de structs: Impresión

Como ya hemos visto, no es posible imprimir una struct completa. Entonces, si queremos imprimir las structs contenidas en el arreglo, deberemos iterar por él, imprimiendo campo a campo:

```
void imprimir(Persona arreglo[], int dl) {
    for (int i = 0; i != dl; i++) {
        cout << arreglo[i].nombre << endl;
        cout << arreglo[i].dni << endl;
        cout << arreglo[i].direccion << endl;
    }}
```


Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

3.3. Arreglos de structs: Carga de datos

Los algoritmos de carga son los mismos que usamos para cargar cualquier arreglo, sólo que ahora indicamos cada campo de la struct donde queremos almacenar algo. Por ejemplo, para insertar al final:

```
int cargar(Persona arreglo[], int dl) {
    Persona p;
    cout << "Nombre: (x para cortar) ";
    getline(cin>>ws, p.nombre);
    while (p.nombre != "x" and dl < DIMENSION_FISICA) {
        cout << "DNI: ";
        cin >> p.dni;
        cout << "Dirección: ";
        getline(cin>>ws, p.direccion);
        arreglo[dl] = p;
        dl++;
        cout << "Nombre: (x para cortar)
        getline(cin>>ws, p.nombre);
    }
    return dl;
}
```

No cargamos los datos **directamente** en el arreglo, sino que usamos una variable auxiliar, para evitar escribir más allá del final del arreglo si es que está lleno.

Como **p** es de **tipo Persona** y el arreglo contiene elementos de tipo Persona, podemos asignar p a una posición del arreglo.

4. Resumen de la unidad

Los algoritmos de búsqueda son técnicas esenciales en programación y ciencias de la computación para localizar un elemento específico dentro de una estructura de datos como un arreglo, lista, árbol o grafo. En esta unidad se abordan tres tipos principales de búsqueda: búsqueda lineal o secuencial desordenada, búsqueda lineal o secuencial ordenada y búsqueda binaria. La búsqueda lineal o secuencial desordenada recorre la estructura elemento por elemento hasta encontrar el objetivo. Este método es eficiente cuando el elemento buscado se encuentra al inicio de la estructura, pero su eficiencia disminuye si el elemento está al final o no está presente. La implementación en C++

Documento para la asignatura "Programación Imperativa" by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

consiste en un ciclo que compara cada elemento del arreglo con el valor buscado hasta encontrarlo o llegar al final del arreglo. La búsqueda lineal o secuencial ordenada, aunque similar a la desordenada, se realiza en una estructura ordenada. Mejora la eficiencia en el caso en que el elemento no está presente, ya que la búsqueda puede detenerse tan pronto se encuentra un elemento mayor al buscado. Por otro lado, la búsqueda binaria requiere que la estructura esté ordenada. Este método divide repetidamente el intervalo de búsqueda a la mitad, lo que permite localizar el elemento buscado de manera más eficiente que la búsqueda secuencial. Aunque más compleja de implementar, la búsqueda binaria es significativamente más rápida para grandes estructuras de datos.

Los algoritmos de ordenamiento son fundamentales para optimizar el uso de otros algoritmos que requieren estructuras de datos ordenadas. Estos algoritmos se dividen en métodos iterativos y recursivos, y en esta unidad se analizan los métodos iterativos más comunes: el ordenamiento por selección, el ordenamiento por burbuja y el ordenamiento por inserción. El ordenamiento por selección selecciona repetidamente el menor elemento del sector desordenado y lo mueve al sector ordenado. Este método es simple de entender y programar, pero no es eficiente para grandes volúmenes de datos debido a su complejidad O(N^2). Por su parte, el ordenamiento por burbuja compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este algoritmo es eficiente si la estructura ya está casi ordenada, pero en el peor de los casos también tiene una complejidad O(N^2). Además, es fácil de optimizar deteniendo el proceso si no se realizan intercambios en una pasada completa. El ordenamiento por inserción construye gradualmente una sección ordenada del arreglo, insertando cada nuevo elemento en su lugar correspondiente. Este método es eficiente para arreglos pequeños o casi ordenados, con una complejidad que varía desde O(N) en el mejor de los casos hasta O(N^2) en el peor.

Los structs permiten agrupar datos de diferentes tipos bajo un mismo nombre, facilitando la manipulación de estructuras de datos complejas. En esta unidad se cubren varios aspectos de los structs, como sus características y definición, así como el manejo de arreglos de structs. Se explica cómo definir structs y declarar variables en diferentes ámbitos, incluyendo ejemplos prácticos de asignación, comparación e impresión de structs. Además, se abordan las técnicas para manejar arreglos de structs, incluyendo la impresión y la carga de datos. Ejemplos completos ilustran cómo trabajar con estas estructuras de datos en C++.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License

En resumen, esta unidad brinda una comprensión sólida sobre algoritmos de búsqueda, ordenamiento y el uso de estructuras como struct. Estas herramientas son fundamentales para desarrollar programas eficientes y manejables en diversos contextos de la informática.

Documento para la asignatura “Programación Imperativa” by Equipo docente del área de ALGORITMOS Y ESTRUCTURAS DE DATOS de la UNNOBA es licensed under a Creative Commons Reconocimiento-NoComercial 4.0 Internacional License