1. Ordenamiento

Existen numerosos algoritmos para ordenar datos (*sorting*), usualmente dentro de un arreglo u otro tipo de *estructura de datos*. Es importante poder tener organizada la información, ya que la información ordenada es fundamental para la mayoría de las aplicaciones con las que se interactúan todos los días.

Existen muchos algoritmos para este propósito ya que algunos se ajustan mejor como solución en determinados escenarios. Con un fin pedagógico, aquí solo se mencionarán algunos de ellos y se desarrollará uno en particular denominado **ripple sort**.

1.1. Burbujeo (*bubble sort*)

Este método de ordenamiento se basa en recorrer el vector de datos comparando los elemento adyacentes entre sí, intercambiándolos si el elemento de la derecha es mayor o menor según el criterio que se desee emplear. Esto debe repetirse una y otra vez sobre el vector hasta dejarlo ordenado.

Puede encontrar una animación en el siguiente enlace:

`https://tute-avalos.com/images/bubblesort.gif`.

1.2. Por inserción (*insertion sort*)

A diferencia del burbujeo, en este algoritmo se recorre el vector y se toma el valor evaluado y se lo compara con cada uno de los elementos anteriores hasta *insertarlo en el lugar correcto*. Si bien, esta es una mecánica que sería la tendencia a aplicar para un ser humano, es bastante más compleja de programar.

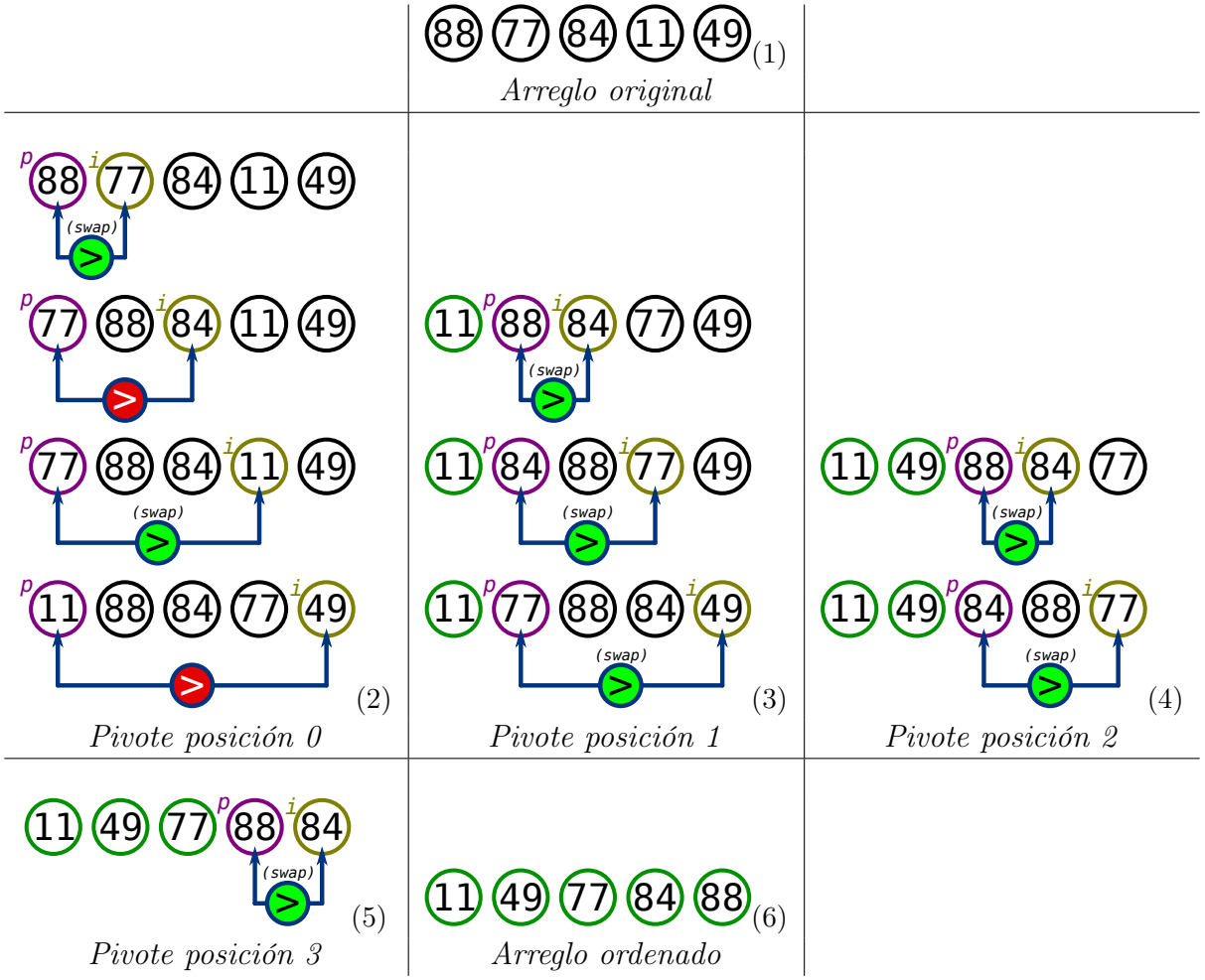
Puede encontrar una animación en el siguiente enlace:

`https://tute-avalos.com/images/insertionsort.gif`.

1.3. Por "onda" (*ripple sort*)

Este es una variante de *bubble sort* para mejorar el tiempo de ejecución al evitar repetir constantemente y es relativamente sencillo de idear. Utiliza un sistema de *pivote*, donde se ordena una posición a la vez. Luego, ésta se va comparando con las posiciones siguientes hasta alcanzar el último elemento (`p+1, p+2, p+3, ..., p+(N-1)`), intercambiando los valores del vector (*swap*) si cumplen el criterio de ordenamiento (ascendente o descendente).

A continuación se expone un ejemplo gráfico de ordenamiento ascendente de un arreglo de 5 posiciones:



Como se observa, el pivote (indicado por el índice `p`) avanza comparando e intercambiando elementos, hasta ubicar el valor más bajo (o alto, según el criterio) en su posición final. En cambio, el elemento a comparar (indicado por el índice `i`) empieza siempre en la posición siguiente al pivote (`p+1`), es por ello que `p` solo va hasta el anteúltimo elemento del arreglo, ya que el último quedará ordenado por descarte.

En la siguiente implementación de código, se generará un arreglo de N elementos aleatorios, mostrando el orden original en que fueron creados y luego se aplicará *ripple sort* **ascendente**, por último se mostrará el arreglo ordenando al final del programa:

Programa 1: Ejemplo práctico de *ripple sort*

```
#include <iostream>          // cout
#include <random>            // random_device
#include <utility>           // swap()

using namespace std;
// Declaración de la función de ordenamiento ripple sort
void ripple_sort(int vector[], int tamanyo);

int main() {
  const int CANT_ELEM=10;//Cantidad de elementos del array
  random_device rand;

  int valores[CANT_ELEM]; // Array de CANT_ELEM
```

```
14    // Se asignan valores alaeatorios entre 0 y 99 al vector
15    //  y se muestran en pantalla.
16    cout << "Vector original: \n";
17    for(int &v : valores) {
18      v = rand() % 100; // se asigna un valor al elemento actual
19      cout << "[" << v << "]"; // se muestra el valor generado
20    }
21    cout << "\n";
22    // Se ordena el vector a través del algoritmo ripple sort
23    ripple_sort(valores, CANT_ELEM);
24    // Se muestra el arreglo ya ordenado:
25    cout << "\nVector ordenado: \n";
26    for(int v : valores) {
27      cout << "[" << v << "]";
28    }
29    cout << endl;
30  }
31
32  void ripple_sort(int vector[], int tamanyo) {
33    // El pívot se inicializa en 0 y se desplaza hasta el
34    //  anteúltimo elemento.
35    for(int pivot=0; pivot < tamanyo-1; pivot++) {
36      // Luego se compara sucesivamente con los demás elementos,
37      // intercambiándolos cuando es necesario.
38      for(int i=pivot+1; i < tamanyo; i++) {
39        if(vector[pivot] > vector[i]) { // swap:
40          swap(vector[pivot],vector[i]); // nuevo valor de pivote
41        }
42      }
43    }
44  }
```

El algoritmo de ordenamiento está comprendido entre las *líneas 32 y 44*, donde se observa que el pivote está inicializado en 0 y se desplaza consecutivamente hasta alcanzar el "último elemento -1", y que el índice con el cual se compararán los demás elementos con el pivote empieza siempre en **`pivot+1`** (*línea 38*) y se desplaza hasta el último elemento. El signo de la comparación (*línea 39*) es el que determinará si se ordena de forma ascendente o descendente. En este caso, si el pivote es mayor que el comparado en **`pivot+i`** se intercambian, de esta forma, *el menor valor es el nuevo pivote* (*línea 40*) y, al final, quedan ordenados **ascendentemente** (de menor a mayor).

Una posible salida al ejecutar este programa sería:

```
Vector original:
[44][39][2][94][62][14][0][75][13][16]

Vector ordenado:
[0][2][13][14][16][39][44][62][75][94]
```

Cuando se requiere ordenar vectores asociados, es decir, que ambos están relacionados por su índice, pero son datos en arreglos independientes, debe tener en cuenta que se aplica el criterio de ordenamiento en uno de ellos y luego se intercambian ambos datos.

Analicemos el siguiente ejemplo donde se requieren 5 nombres junto a su promedios y luego se pide mostrarlos ordenados de mayor a menor según su promedio.

Programa 2: Ordenamiento de más de un arreglo a la vez

```cpp
#include <iostream> // cout, getline()
#include <string>   // stof()
#include <cctype>   // isspace()
#include <utility>  // swap()

using namespace std;

/*               Declaración de funciones                */
string pedir_texto(string mensaje);
double pedir_numero(string mensaje);
double pedir_numero_entre(string mensaje,double min,double max);
void ordenar_por_promedio(double prom[],string nom[], int tam);

int main(){
  const int CANT_ESTUDIANTES = 5;
  string nombres[CANT_ESTUDIANTES];
  double promedios[CANT_ESTUDIANTES];
  // Se solicita el ingreso de datos por el usuario
  for(int i=0; i < CANT_ESTUDIANTES; i++) {
    nombres[i]=pedir_texto("Ingrese nombre del estudiante "
                           +to_string(i+1)+": ");
    promedios[i]=pedir_numero_entre("Ingrese promedio:",1,10);
  }
  // Ripple Sort descendente (mayor a menor) de promedios:
  ordenar_por_promedio(promedios, nombres, CANT_ESTUDIANTES);

  cout << "\nLos estudiantes ordenados por promedio:\n";
  for(int i=0; i < CANT_ESTUDIANTES; i++) {
    cout<< i+1 <<". "<< nombres[i] <<" ("<< promedios[i] <<")\n";
  }
  cout << endl;
}

/*              Implementación de funciones              */
void ordenar_por_promedio(double prom[], string nom[], int tam) {
  for(int p=0; p < tam-1; p++) {
    for(int i=p+1; i < tam; i++) {
      if(prom[p] < prom[i]) {
        // se ordenan los promedios
        swap(prom[p],prom[i]);
        // se ordenan los nombres con el mismo índice:
        swap(nom[p], nom[i]);
      }
    }
  }
}

```

```
48 string pedir_texto(string mensaje) {
49   string texto;
50   do {
51     cout << mensaje;
52     getline(cin, texto);
53   } while(texto == "" or isspace(texto[0]));
54   return texto;
55 }
56
57 double pedir_numero(string mensaje) {
58   while(true) {
59     try {
60       return stof(pedir_texto(mensaje));
61     } catch (exception &e) {
62       cout << "¡Error! Escriba un número válido\n";
63     }
64   }
65 }
66
67 double pedir_numero_entre(string mensaje,double min,double max) {
68   if(min > max) {
69     swap(min, max);
70   }
71   double f;
72   do {
73     f = pedir_numero(mensaje);
74   } while(f < min or f > max);
75   return f;
76 }
```

Si bien este tipo de ordenamientos *cruzados* ocurren, lo ideal es tener un solo arreglo con los **datos estructurados** (registros), que es un tema que se abordará en la próxima unidad.

1.4. Algoritmos avanzados

1.4.1. Ordenamiento rápido (*quick sort*)

Este algoritmo de ordenamiento es uno de los más populares y estuvo impuesto por muchos años como la mejor solución genérica. Éste, al igual que muchos otros, está basado en la técnica de *divide y vencerás*. Se toma el último elemento como *pívot*, y se recorren los elementos comparando el primero y ante-último e intercambiándolos si el de la izquierda es mayor que el *pívot* y el de la derecha menor. Luego se traslada el *pívot* a la mitad del arreglo. Para este momento todos los mayores están a la derecha del pivot y los menores a la izquierda, pero desordenados, entonces se hace lo mismo con ambas mitades, y luego con la mitad de la mitad, etc.

A continuación se deja un link a una animación que clarifica la operación:

`https://tute-avalos.com/images/quicksort.gif`

1.4.2. Ordenamientos "mixtos"

Hoy en día los algoritmos que se utilizan para ordenar información de grandes o bajos volúmenes son, en realidad, una mezcla de varios algoritmos. En determinados casos, dependiendo de la longitud de los datos o la tecnología utilizada para almacenarlos se opta por uno u otro como parte del algoritmo en sí. La mayoría de los lenguajes ya los incluyen como herramientas de biblioteca, pero en este curso nos centramos en el pensamiento computacional y entender este tipo de algoritmos y no ser meros usuarios o *codificadores* de software. Aunque, cabe destacar, que en general es conveniente utilizar las herramientas provistas por los lenguajes, ya que han sido arduamente *testeadas* y evolucionan con las distribuciones nuevas de los mismos.

2. Búsqueda

Otro factor importante en un arreglo es la búsqueda, es decir, encontrar uno o varios datos que cumplan con un criterio dado. Para ello, existen varios algoritmos. En este curso expondremos únicamente 2 de ellos. Por un lado, la más fácil de implementar es la búsqueda lineal.

2.1. Lineal

Las búsquedas lineales son las que utilizaríamos naturalmente, recorriendo uno por uno los elementos hasta encontrar el o los que cumplan con el criterio deseado. Esto usualmente se logra con un bucle que recorre desde el primero (o segundo) elemento comparándolo con otro elemento que cumple con el criterio dado.

Supóngase que se quiere encontrar el índice del mejor elemento en un arreglo de números, una posible implementación de esta función sería::

```
/** Busqueda lineal del indice con el mayor valor **/
int obtener_indice_del_mayor(double vector[],int cantidad) {
  int indiceDelMayor = 0;
  for(int i=1; i < cantidad; i++) {
    // Si el valor actual (en la posición i) tiene
    // un valor mayor al anteriormente detectado se
    // actualiza el indiceDelMayor con el actual.
    if(vector[i] > vector[indiceDelMayor]) {
      indiceDelMayor = i;
    }
  }
  return indiceDelMayor;
}
```

Como se observa el algoritmo anterior este es recorrido con un primer pívot en la posición 0 y comparándolo secuencialmente con los próximos elementos. Si se encontrara el que cumpliera con el criterio, este es reemplazado.

Por ejemplo, si tuviera arreglos desordenados de estudiantes y promedios y quisiera mostrar el estudiante de mejor promedio podría hacer:

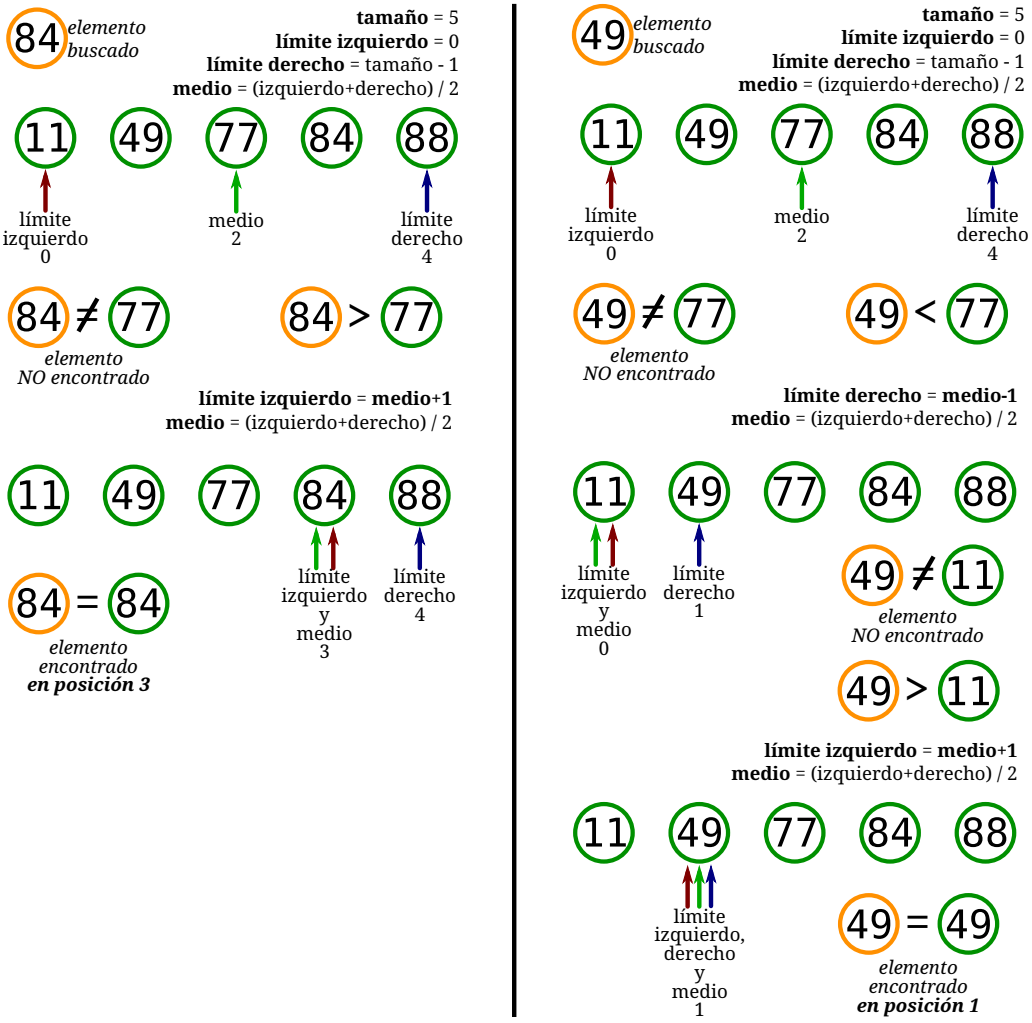
```
int mejor=obterner_indice_del_mayor(promedios,cant_estud);
cout << "Mejor promedio: "<< estudiantes[mejor] << "\n";
```

También podría *ahorrarse* la variable si se hiciera:

```
estudiantes[obterner_indice_del_mayor(promedios, cant_estud)]
```

2.2. Binaria

Este es un algoritmo muy sencillo, se basa en comparar el elemento del medio entre dos extremos. Primero se evalúa el elemento en el medio del arreglo, es decir entre el primer y último elemento, si este elemento es mayor al buscado entonces se busca entre el primero y el anterior al consultado, sino entre el siguiente y el último y así sucesivamente hasta encontrar el elemento deseado.



Es un pre-requisito para utilizar este algoritmo que el arreglo esté previamente ordenado. Sino, será imposible poder determinar dónde se encuentra el elemento.

Ejemplo de búsqueda binaria

```
int busqueda_binaria(int vector[], int buscado, int tam) {
  int izq = 0;
  int der = tam - 1;
  // Mientras sean válidos los límites:
  while (izq <= der) {
    // Se calcula el índice de la mitad
    int med = (izq + der) / 2;
    // Si se encontró el elemento buscado se devuelve el índice:
    if (vector[med] == buscado) {
      return med;
    }
    // Se calcula el nuevo límite:
    if (buscado < vector[med]) {
      der = med - 1;
    } else {
      izq = med + 1;
    }
  }
  // Devolver un valor de índice no válido
  return -1;
}
```

Tener el vector ordenado es una ventaja a la hora de hacer búsquedas ya que para encontrar máximos y mínimos simplemente se obtiene el último o primer elemento respectivamente (sin necesidad de buscar).

3. Mezclar un arreglo

Desordenar un arreglo correctamente requiere lógica no trivial, ya que no basta con intercambios aleatorios simples. Pero afortunadamente existe una función estándar que proporciona un algoritmo de mezcla llamada `shuffle()` presente en la biblioteca `algorithm`.

Programa 3: Desordenar un arreglo con `shuffle()`

```
#include <iostream>  // cout
#include <algorithm> // shuffle()
#include <random>    // random_device

using namespace std;

int main() {
  const int CANT_ELEM = 10;
  // Generador rand, utilizado por shuffle()
  random_device rand;

  // vector ordenado
  int vec[CANT_ELEM] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

```

```
15    // Desordenar el vector:
16    shuffle(vec, vec+CANT_ELEM, rand);
17
18    // Se muestra el vector desordenado:
19    for(int v : vec) {
20      cout << "[" << v << "]";
21    }
22    cout << '\n';
23  }
```

Como se observa en la *línea 10*, es necesario proveer un generador de número aleatorios `rand`, ya que la función `shuffle()` la utiliza internamente. Por otro lado, en la *línea 16* se utiliza este algoritmo de mezcla, pasando el arreglo como primer argumento y como segundo el mismo arreglo sumándole la cantidad de elementos, por último el generador de números aleatorios `rand`.

Una posible salida de este programa se vería de la siguiente manera:

```
[6][4][7][1][0][5][8][3][9][2]
```

4. Ejercitación

Codifique los siguientes programas utilizando arreglos y funciones en C++.

1. Pida ingresar 10 nombres y muéstrelos ordenados alfabéticamente.
2. Pida ingresar 5 estudiantes por apellido y sus notas trimestrales (3 notas). Muestre los apellidos junto a su 3 notas ordenados por el promedio de las notas (de mayor a menor).
3. Pida ingresar 10 fechas (día, mes y año) en formato numérico. Muestre las fechas ordenadas de la más actual a la más antigua en el formato `día/mes/año`.
4. Pida ingresar los nombres completos (un nombre de pila y primer apellido), guardados en vectores diferentes. Muéstrelos ordenados en el formato "Apellido, Nombre" sabiendo que si hay apellidos repetidos, debe utilizar el nombre como criterio de orden. P.e. `Ferrari, Alfredo` está antes que `Ferrari, Bruno`.
5. Desarrolle un software que pida ingresar 10 "nombres de usuario" cualquiera y asignarles un código de 4 dígitos diferentes entre sí. Luego se ingresa a un menú que permite:
   a) `Ingresar código` que pida ingresar un código y muestre su nombre o el texto `"Usuario no encontrado"`.
   b) `Listar` que muestra los usuarios ordenados por código.
   c) `Salir` que finaliza la ejecución del programa.