# Ejercitación – Algoritmos genéricos sobre iteradores

Se recomienda resolver los ejercicios en orden. En CLion se encuentran disponibles los siguientes targets:

- ejN, si ($1 \leq N \leq 8$): hasta al ejercicio N inclusive.

Los targets también pueden compilarse y ejecutarse sin usar CLion. Para ello:

1. En una consola pararse en el directorio raíz del projecto. En este debería haber un archivo CMakeLists.txt.
2. Ejecutar el comando $ cmake . (incluyendo el punto). Esto generará el archivo Makefile.
3. Ejecutar el comando $ make TARGET donde TARGET es uno de los targets mencionados anteriormente. Esto creará un ejecutable con el nombre del target en el directorio actual.
4. Ejecutar el comando $ ./TARGET siendo TARGET el nombre del target utilizado anteriormente. Esto correrá el ejecutable.

## Ejercicio 1

Implementar la función:

```
template<class Contenedor>
typename Contenedor::value_type minimo(const Contenedor& c);
```

Que devuelva el elemento mínimo del contenedor, suponiendo que la colección no es vacía. Se asume que existen:

- `bool operator<(const Contenedor::value_type&, const Contenedor::value_type&)`
- `Contenedor::const_iterator Contenedor::begin() const`
- `Contenedor::const_iterator Contenedor::end() const`

## Ejercicio 2

Implementar la función:

```
template<class Contenedor>
typename Contenedor::value_type promedio(const Contenedor& c);
```

Que devuelva el promedio de los elementos del contenedor, suponiendo que la colección no es vacía. Se asume que existen

- `Contenedor::const_iterator Contenedor::begin() const`
- `Contenedor::const_iterator Contenedor::end() const`

y que `Contenedor::value_type` es un tipo numérico, cuyos valores se pueden sumar y dividir por un entero no nulo.

## Ejercicio 3

Implementar las funciones análogas a los ejercicios 1 y 2 pero esta vez recibiendo iteradores:

```
template<class Iterator>
typename Iterator::value_type minimoIter(const Iterator& desde, const Iterator& hasta);

template<class Iterator>
typename Iterator::value_type promedioIter(const Iterator& desde, const Iterator& hasta);
```

## Ejercicio 4

Implementar una función que dado un contenedor y un elemento del tipo guardado en el mismo, modifique el contenedor recibido por parámetro eliminando todas las apariciones del elemento parámetro. La función tendrá la siguiente aridad:

```
template<class Contenedor>
void filtrar(Contenedor &c, const typename Contenedor::value_type& elem);
```

Se aumen las siguientes funciones del contenedor y su iterador:

- `Contenedor::iterator Contenedor::begin()`
- `Contenedor::iterator Contenedor::end()`
- `Contenedor::iterator Contenedor::erase(Contenedor::iterator)`
- `bool operator==(const Contenedor::value_type&, const Contenedor::value_type&)`

El iterador resultado de `erase` se encuentra ubicado en la posición siguiente al elemento eliminado.

## Ejercicio 5

Implementar una función que decida si los elementos —dispuestos en el orden en el que los devuelve el iterador— están ordenados de manera estrictamente creciente.

```
template<class Contenedor>
bool ordenado(Contenedor &c);
```

Se asumen las mismas condiciones que el ejercicio 1.

## Ejercicio 6

Implementar una operación que dado un contenedor y un elemento devuelva una tupla con dos nuevos contenedores del mismo tipo que el recibido por parámetro. El primer contenedor deberá tener todos los elementos menores al elemento. El segundo deberá tener los elementos mayores o iguales. La aridad será la siguiente:

```
template<class Contenedor>
std::pair<Contenedor, Contenedor>  split(const Contenedor & c,
                                         const typename Contenedor::value_type& elem)
```

Se asumen las siguientes funciones:

- `bool operator<(const Contenedor::value_type&, const Contenedor::value_type&)`
- `Contenedor::const_iterator Contenedor::begin() const`
- `Contenedor::const_iterator Contenedor::end() const`
- `Contenedor::iterator Contenedor::insert(const Contenedor::const_iterator&, const Contenedor::value_type&) const`

## Ejercicio 7

Implementar una función que dados tres contenedores del mismo tipo, agregue los elementos de los primeros dos al final del tercero. Se asume como precondición que los elementos de `c1` y `c2` se encuentran ordenados de manera creciente (no necesariamente estricta). Los elementos se deben agregar a `res` de manera creciente (no estricta).

Por ejemplo, si `c1` es la lista `[10, 11, 15]` y `c2` es la lista `[5, 7, 13, 20]`, se deben agregar a `res` los elementos `[5, 7, 10, 11, 13, 15, 20]` en ese orden.

```
template <class Contenedor>
void merge(const Contenedor& c1, const Contenedor & c2, Contenedor & res);
```

Se asume sobre `Contenedor` lo mismo que en el ejercicio 6.

## Ejercicio 8

En el ejercicio 1, se asumió que el tipo contenido (`Contenedor::value_type`) poseeía el operador asignación. Esto puede no cumplirse siempre.

Modificar la implementación del ejercicio 1 para que no requiera asignar el tipo contenido.