Práctica 1 - Arreglos

1. La *mediana* de un arreglo ordenado de n números se define como el elemento del medio cuando n es impar y como el promedio de los dos elementos del medio cuando n es par. Por ejemplo,

```
mediana([-1.0, 2.2, 2.9, 3.1, 3.5]) = 2.9
mediana([-1.0, 2.2, 2.9, 3.1]) = 2.55
```

Escriba una función que reciba un arreglo, no necesariamente ordenado, de números y calcule su mediana:

```
float mediana(float *arreglo, int longitud);
```

Puede utilizar la siguiente rutina para ordenar un arreglo de menor a mayor

```
void bubble_sort(float arreglo[], int longitud) {
  for (int iter = 0 ; iter < longitud - 1 ; iter++) {
    for (int i = 0 ; i < longitud - iter - 1; i++) {
      if (arreglo[i] > arreglo[i + 1]) {
        float aux = arreglo[i];
        arreglo[i] = arreglo[i + 1];
        arreglo[i + 1] = aux;
      }
    }
  }
}
```

Nota: La función `mediana` no debe modificar el arreglo original.

2. Implemente las siguientes funciones para cadenas de caracteres:

a) `int string_len(char* str)`, que retorne la longitud de la cadena `str`, excluyendo el carácter nulo ('\0').

b) `void string_reverse(char* str)`, que invierta la cadena `str`.

c) `int string_concat(char* str1, char* str2, int max)`, que concatene no más de `max` caracteres de la cadena `str2` al final de la cadena `str1`. El carácter inicial de `str2` debe sobrescribir el carácter nulo de `str1`. La cadena resultante debe terminar con un carácter nulo. Retorna el número de caracteres copiados.

Nota: Si `str1` no tiene espacio suficiente para almacenar el resultado, el comportamiento queda indefinido.

d) `int string_compare(char* str1, char* str2)`, que compare en orden lexicográfico las cadenas `str1` y `str2`, y retorne retorne -1 si la primera es menor que la segunda, 0 si son iguales, y 1 si es mayor.

e) `int string_subcadena(char* str1, char* str2)`, que retorne el índice de la primera ocurrencia de la cadena `str2` en la cadena `str1`. En caso de no ocurrir nunca, retorna -1.

f) `void string_unir(char* arregloStrings[], int n, char* sep, char* res)`, que concatene las `n` cadenas del arreglo `arregloStrings`, separándolas por la cadena `sep` y almacenando el resultado en `res`.

Nota: Si `res` no tiene espacio suficiente para almacenar el resultado, el comportamiento queda indefinido.

g) Con el objetivo de medir el rendimiento de las funciones recién definidas, agregue un *contador de operaciones*. Es decir, una variable que se incremente cada vez que realizamos una operación elemental. En cada caso, ¿depende este número de la longitud del arreglo (o de los arreglos) que pasamos como argumento?

3. Considere arreglos de enteros definidos a través de una estructura que lleve registro de la capacidad:

```
typedef struct {
  int* direccion;
  int capacidad;
} ArregloEnteros;
```

Implemente las operaciones básicas:

a) `ArregloEnteros* arreglo_enteros_crear(int capacidad);`

b) `void arreglo_enteros_destruir(ArregloEnteros* arreglo);`

c) `int arreglo_enteros_leer(ArregloEnteros* arreglo, int pos);`

d) `void arreglo_enteros_escribir(ArregloEnteros* arreglo, int pos, int dato);`

e) `int arreglo_enteros_capacidad(ArregloEnteros* arreglo);`

f) `void arreglo_enteros_imprimir(ArregloEnteros* arreglo);`

g) Si midiéramos el tiempo contando la cantidad de operaciones, como hicimos en el ejercicio **2.g)**, ¿cuál de ellas *tardaría* más? ¿Para cuál de ellas su tiempo depende de la longitud del arreglo?

4. Extienda la implementación del ejercicio anterior con las siguientes funciones:

a) `void arreglo_enteros_ajustar(ArregloEnteros* arreglo, int capacidad)`, que ajuste el tamaño del arreglo. Si la nueva capacidad es menor, el contenido debe ser truncado.

b) `void arreglo_enteros_insertar(ArregloEnteros* arreglo, int pos, int dato)`, que inserte el dato en la posición dada, moviendo todos los elementos desde esa posición un lugar a la derecha (tendrá que incrementar el tamaño del arreglo).

c) `void arreglo_enteros_eliminar(ArregloEnteros* arreglo, int pos)`, que elimine el dato en la posición dada, moviendo todos los elementos posteriores un lugar a la izquierda (tendrá que reducir el tamaño del arreglo).

5. Una matriz M de tamaño $m \times n$ se puede definir como un arreglo bidimensional, donde el elemento de la fila i y columna j es $M[i][j]$, o también como un arreglo unidimensional, donde el elemento de la fila i y columna j es $M[i * n + j]$, para todos $0 \leq i \leq m - 1$ y $0 \leq j \leq n - 1$.

a) Implemente el tipo de datos `Matriz` usando arreglos unidimensionales y bidimensionales de números flotantes, junto a las operaciones básicas asociadas (crear, destruir, escribir una posición, leer de una posición).

A pesar de tener dos implementaciones distintas, las operaciones básicas sobre una matriz son las mismas. Por dicha razón queremos tener un único archivo de cabecera `matriz.h`, y un archivo `.c` por cada implementación: `matriz_uni.c` y `matriz_bi.c`.

De esta forma, si tuviéramos un programa `main.c` que utiliza matrices (incluye `matriz.h`), podremos optar por la implementación preferida en el momento de compilarlo.

```
$ gcc main.c matriz_uni.c
```

b) ¿Cuáles son las ventajas y desventajas de cada implementación?

c) Defina una función `matriz_intercambiar_filas`, que intercambie dos filas dadas. ¿En cuál de las dos implementaciones anteriores resulta más simple de definir? ¿Y para la operación `matriz_insertar_fila`, que agrega una nueva fila en una posición dada?