Parcial 2 - Algoritmos I Taller: Tema F

Ejercicio 1

Considerar las siguientes afirmaciones y seleccione la respuesta correcta:

a) Dada la siguiente función en C:

```
int f(int x) {
    int a=0, b=2;
    printf("Ingrese un valor entero:\n");
    scanf("%d", &a);
    printf("El resultado es: %d", x * a);
    return b;
}
```

   1) La función toma dos parámetros y devuelve el producto entre **`x`** y **`a`**
   2) Está mal definida la función porque falta **`\n`** en la segunda llamada de **`printf()`**
   3) La función toma un parámetro de tipo **`int`** y devuelve siempre **`2`**
   4) La función devuelve un par ordenado cuyo primer componente es **`x*a`** y el segundo es **`b`**.
   5) Ninguna de las anteriores es cierta

b) ¿Cuál de las siguientes afirmaciones son ciertas en C?
   1) La sentencia **`if`** de C siempre tiene que tener la cláusula **`else`**
   2) Si se quiere obtener un valor booleano del usuario y guardarlo en la variable **`bool b;`** se puede utilizar **`scanf("%d", &b);`**
   3) Si declaro una variable **`int x;`** es lo mismo hacer la asignación **`x = 3;`** que escribir la asignación **`X = 3;`** porque C es *Case Sensitive*
   4) Ninguna de las anteriores es cierta

c) Indicar cuál de las comparaciones entre lenguajes imperativos y funcionales es cierta:
   1) Los lenguajes imperativos tienen variables y los funcionales no
   2) En los lenguajes funcionales el resultado de un programa es una expresión a la cual no se le pueden aplicar más reglas de reducción mientras que en los lenguajes imperativos el resultado es un estado final al que se llega luego de ejecutar todas las sentencias.
   3) Los lenguajes imperativos y funcionales computan de la misma manera solo que los funcionales son fuertemente tipados
   4) Los lenguajes funcionales computan siempre a través de un intérprete mientras que los lenguajes imperativos siempre requieren un compilador.

d) Para determinar que un valor es mayor o igual a otro en C se usa la expresión:
   1) a => b
   2) a > b || a = b
   3) not (a < b)
   4) a >= b
   5) Ninguno de los anteriores.

Ejercicio 2

Considerar la siguiente código con asignaciones múltiples:

```
var x, y, z : Int;
{Pre: x = X, y = Y, z = Z, Z mod 2 ≠ Y mod 2}
if (x mod 2 = 0)→
     x, y, z := y * x, y + x + z + 1, 2*z
□ (x mod 2 ≠ 0)→
     x, y, z := 2*y + x + 1, y + x + z, 2*z
fi
{Pos: (x+y+z) mod 2=0 ∧ z=2*Z ∧ ((x=Y*X ∧ y=Y+X+Z+1) ∨ (x=2*Y+X+1 ∧ y=Y+X+Z))}
```

Escribir un programa en lenguaje C equivalente usando asignaciones simples teniendo en cuenta que:

- Se deben verificar las pre y post condiciones usando la función `assert()`.
- Los valores iniciales de `x, y, z` deben ser ingresados por el usuario
- Los valores finales de `x, y, z` deben mostrarse por pantalla usando la función `imprimir_entero` del proyecto 3.

NOTA: Poner como comentario al menos un ejemplo de ejecución, con los parámetros de entrada y la salida de tu programa (puedes hacer un copiar y pegar de la consola).

Ejercicio 3

Dada la siguiente estructura

```
struct paridad {
    int pares;
    int impares;
    bool paridad_alternada;
};
```

programar la función

```
struct paridad analizar_paridad(int tam, int a[]);
```

que dado un tamaño de arreglo `tam` y un arreglo `a[]` devuelve una estructura **`struct paridad`**, donde en el campo `pares` dejará la cantidad de valores pares encontrados en `a[]`, en el campo `impares` contará la cantidad de valores impares en `a[]` y en el campo `paridad_alternada` dejará el valor `true` si los valores son pares / impares alternadamente en `a[]`.

Por ejemplo:

tam	**a[]**	**res**=analizar_paridad(tam, a)	**Comentario**

4	[8,3,12,5]	`res.pares == 2`
		`res.impares == 2`
		`res.paridad_alterada == true`
			En el arreglo se encuentran los valores `8` y `12` que son pares (dos en total) mientras que los valores `3` y `5` son impares (también dos). La paridad **es alternada** ya que: **`a[0]`** es par y **`a[1]`** es impar; **`a[1]`** es impar y **`a[2]`** es par; **`a[2]`** es par y **`a[3]`** no lo es.

6	[6,3,7,4,1,0]	`res.pares == 3`
		`res.impares == 3`
		`res.paridad_alternada == false`
			En el arreglo se encuentran los valores pares `6`, `4` y `0` sumando tres en total, por otro lado los valores 3, 7 y 1 son impares (también son tres). La paridad **no es alternada** ya que **`a[1]`** es impar y **`a[2]`** también.

5	[1,6,5,8,7]	`res.pares == 2`
		`res.impares == 3`
		`res.paridad_alternada == true`
			En el arreglo se encuentran los valores `6` y `8` que son pares (dos en total) y los valores impares `1`, `5` y `7` (tres en total). La paridad **es alternada** ya que: **`a[0]`** no es par y **`a[1]`** sí lo es; **`a[1]`** es par y **`a[2]`** no; **`a[2]`** no es par y **`a[3]`** sí; **`a[3]`** es par y **`a[4]`** no.

4	[1,9,9,8]	`res.pares == 1`
		`res.impares == 3`
		`res.paridad_alternada == false`
			En el arreglo se encuentra el valor par 8 (un solo par) y los valores impares 1, 9 y 9 (tres en total). La paridad **no es alternada** ya que **`a[0]`** no es par y **`a[1]`** tampoco.

De los ejemplos se puede ver que:

- Si contamos con un predicado `es_par()`, verificar que hay alternancia de paridad entre una posición y la siguiente es lo mismo que verificar que el resultado de `es_par()` del valor en esa posición es distinto al resultado de `es_par()` del siguiente elemento.

NOTA: No es obligatorio definir una función `es_par()`

Cabe aclarar que `analizar_paridad()` no debe mostrar ningún mensaje por pantalla ni pedir valores al usuario. Además debe programarse usando **un solo ciclo**.

En la función `main` se debe solicitar al usuario ingresar un arreglo de longitud `N`. Definir a `N` como una constante, **el usuario no debe elegir el tamaño del arreglo**.

Finalmente desde la función `main` se debe mostrar el resultado de la función `analizar_paridad()` por pantalla.

NOTA: Poner como comentario al menos un ejemplo de ejecución, con los parámetros de entrada y la salida de tu programa (puedes hacer un copiar y pegar de la consola).

Ejercicio 4

Programar la siguiente función

```
struct rango_info verificar_rango(float x, struct rango r);
```

donde las estructuras `struct rango` y `struct rango_info` se definen de la siguiente manera:

```
struct rango {
    float cota_inf;
    float cota_sup;
};
```

```
struct rango_info {
    bool es_anterior;
    bool es_posterior;
    bool esta_dentro;
};
```

La función `verificar_rango` toma un valor decimal flotante `x` y una **`struct`** `rango`, y devuelve una **`struct`** `rango_info` con tres booleanos que respectivamente indican:

- `es_anterior` es **`true`** si y sólo si `x` es menor que `cota_inf`. Caso contrario es **`false`**.
- `es_posterior` es **`true`** si y sólo si `x` es mayor que `cota_sup`. Caso contrario es **`false`**.
- `esta_dentro` es **`true`** si y sólo si `x` es mayor o igual que `cota_inf` y es menor o igual que `cota_sup`. Caso contrario es **`false`**.

En la función `main` se debe solicitar al usuario ingresar los valores de la **`struct`** `rango` y un valor flotante `x` y luego de llamar a la función `verificar_rango` mostrar el resultado por pantalla (los tres booleanos de **`struct`** `rango_info`).

NOTA: Poner como comentario al menos un ejemplo de ejecución, con los parámetros de entrada y la salida de tu programa (puedes hacer un copiar y pegar de la consola).