Parcial 2 - Algoritmos I Taller: Tema C

Ejercicio 1

Considere las siguientes afirmaciones y seleccione la respuesta correcta:

a) Dada la siguiente función en C:

```
int g(void) {
    int a=0, res=0;
    scanf("%d", &a);
    if (a==0) {
      res = a+1;
    } else {
      res = 0;
    }
    return res;
}
```

   1) La función toma dos parámetros y devuelve **a** sumado **1**
   2) La función no tiene parámetros y un buen nombre en lugar de **g** podría ser **not**
   3) Está mal definida la función porque retorna un **int**
   4) La función devuelve un par ordenado cuyo primer componente es **res** y el segundo es **a**.
   5) Ninguna de las anteriores es cierta

b) ¿Cuál de las siguientes afirmaciones es cierta en C?
   1) La sentencia **while (1) {};** ocasiona que el programa nunca termine al ser ejecutado.
   2) La función **printf** toma <u>al menos</u> dos argumentos.
   3) Las siguientes declaraciones modifican el estado de manera <u>diferente</u>:
      1) **int x = 0, y = 0;**
      2) **int x = 0; int y = 0;**
   4) Ninguna de las anteriores es cierta.

c) Indicar cuál de las siguientes sentencias **assert NO** es equivalente a “skip”:
   1) **assert(0);**
   2) **assert(1);**
   3) **assert(100+1);**
   4) **assert(-1);**

d) Dado un arreglo de tamaño 10 declarado como **int a[10];** <u>inicializado completamente con 100 en todas sus posiciones</u>, cuál de los siguientes es válido:
   1) a.0
   2) a[10-3]
   3) a[10]
   4) a[a[9]]
   5) Ninguna de los anteriores.

Ejercicio 2

Considerar el 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 := x + 1, y + z + y, 2*x
□ (x mod 2 ≠ 0)→
     x, y, z := x - 1, y - z - y, 2*x
fi
{Pos: z=2*X ∧ ((x=X+1 ∧ y=Y+Z+Y)) ∨ (x=X-1 ∧ y=Y-Z-Y))}
```

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 stonks {
    int sube;
    int baja;
};
```

programar la función

```
struct stonks stonks_master(int tam, int a[]);
```

que dado un tamaño de arreglo `tam` y un arreglo `a[]` devuelve una estructura **`struct stonks`**, donde en el campo `sube` contará la cantidad de veces que *a[i] <= a[i+1] para i en el rango 0 <= i < tam-1*, de manera análoga `baja` contará la cantidad de veces que *a[i] > a[i+1] para el mismo rango.*

Por ejemplo:

tam	a[]	res=stonks_master(tam, a)	Comentario
4	[1,2,3,4]	res.sube == 3 res.baja == 0	En el arreglo se encuentran todos valores crecientes por lo tanto al comparar cada valor con su consecutivo solo tenemos subidas. Es decir, se da que: - **a[0] <= a[1]** - **a[1] <= a[2]** - **a[2] <= a[3]**
6	[6,3,7,4,1,0]	res.sube == 1 res.baja == 4	El arreglo comienza bajando, luego sube y finalmente baja hasta el final. Es decir, se da que: - **a[0] > a[1]** - **a[1] <= a[2]** - **a[2] > a[3]** - **a[3] > a[4]** - **a[4] > a[5]**
5	[-1,-2,-2,-3,-4]	res.sube == 1 res.baja == 3	En este caso todos los números son negativos, y se van haciendo progresivamente más pequeños. Salvo en la comparación entre **a[1]** y **a[2]** que vale la igualdad y por lo tanto contamos una única “subida”

Cabe aclarar que `stonks_master()` no debe mostrar ningún mensaje por pantalla ni pedir valores al usuario.

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 `stonks_master()` 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

La *famosa* blockchain Cardano vio la luz por el año 2017, iniciando con un protocolo bien simple muy similar al de otra (esta de verdad) famosa blockchain: Bitcoin. Eventualmente Cardano fue evolucionando, donde cada vez que se produce una evolución importante se dice que la blockchain *cambia de era*. Por ejemplo, tenemos las eras Byron, Shelley, Alonzo, Babbage.

Nuestra tarea será traducir determinado tipo de dato de la era Alonzo a la era Babbage. En particular queremos traducir una componente central de Cardano el tipo **output**.

- Un output en **Alonzo** es básicamente un arreglo de **`int`** que solo puede tener tamaño 2 o 3.
- Un output en **Babbage**, es una estructura más definida con 4 campos, uno de los cuales puede no estar presente.(esto lo indica present)

En lo que respecta a nosotros podemos asumir que ambas versiones contienen la misma información, solo que organizada de manera distinta. Puntualmente los tipos serán:

Para **Alonzo**, podremos simplemente declarar un arreglo de enteros de tamaño 2 o 3 (**cualquier otro tamaño no tendrá sentido**). Es decir, **no hace falta** un nuevo tipo y podemos hacer simplemente: **int a_output[2];** o **int a_output[3];**

Para **Babbage**, en cambio, tendremos los siguientes tipos nuevos

```
typedef struct {
  int addr;
  int value;
  int datum;
  bool present;
} babbage_output;
```

Por ejemplo, asumiendo tenemos declarada una variable **`babbage_output b_out;`** los siguientes outputs son el mismo pero en versiones distintas:

Alonzo	Babbage	Comentario

`[ 4242, 10 ]`	`b_out.addr == 4242`
	`b_out.value == 10`
	`b_out.datum == 0`
	`b_out.present == false`
		Output solo con dos enteros.
		
		Siempre pasará que el primer elemento del arreglo se corresponde con el campo `addr` y el segundo con el campo `value`.
		
		Notar el campo `present` debe ser **false** dado que no hay un tercer elemento en el arreglo.

`[ 4243, 100, 5555 ]`	`b_out.addr == 4243`
	`b_out.value == 100`
	`b_out.datum == 5555`
	`b_out.present == true;`
		Output con tres enteros.
		
		Como antes, **Siempre** pasará que el primer elemento del arreglo se corresponde con el campo `addr`, el segundo con el campo `value` y ahora además el tercer elemento del arreglo será el último campo `datum`.
		
		Notar el campo `present` debe ser **true** dado hay un tercer elemento en el arreglo.

La traducción de un Alonzo output a un Babbage output se deber realizar con la función `alonzo_to_babbage`, cuyo prototipo será:

```
babbage_output alonzo_to_babbage(int tam, int a_output[])
```

La función `alonzo_to_babbage()` tomará un tamaño y un arreglo de enteros, cuya precondición será que `tam` solo puede tener dos valores posibles: 2 o 3, para asegurar esto deberemos usar la sentencia `assert`. El resultado de llamar a la función devolverá un `babbage_output.`

En la función `main` se debe llamar a la función `alonzo_to_babbage` para al menos dos alonzo outputs distintos e imprimir por pantalla el resultado de la traducción. Por simplicidad esta permitido especificar en la misma función `main` ambos alonzo outputs cuando los declaramos, es decir: `int a_output[2] = {4242,10};`