Examen final 2021-04-17

95.14/75.40 - Algoritmos y Programación I - Curso Essaya

Objetivo

Se dispone de los archivos `ej1.py`, `ej2.py`, `ej3.py`, `ej4.py` y `ej5.c` correspondientes a los 5 ejercicios del examen.

Cada uno tiene un lugar para escribir la implementación del ejercicio, y una función de pruebas para verificar que la solución es correcta.

El examen se aprueba con al menos 3 ejercicios correctamente resueltos. Un ejercicio se considera correctamente resuelto si:

- El programa `ej<n>` no tiene errores de sintaxis y puede ser ejecutado
- La implementación cumple con lo pedido en el enunciado

En algunos ejercicios se incluye un ejemplo de uno o dos casos de prueba y queda a cargo del alumno agregar más casos de prueba, para los que se provee sugerencias. En otros ejercicios se provee únicamente sugerencias. La implementación de las pruebas adicionales es **opcional**, pero se recomienda hacerlo ya que permite asegurar que la resolución del ejercicio es correcta.

Ejercicios en lenguaje Python

Al ejecutar cada uno de los ejercicios (`python3 ej<n>.py`), se ejecutan todas las pruebas presentes en la función `pruebas`.

Si alguna de las verificaciones falla, se imprime un mensaje de error y el programa termina su ejecución. Por ejemplo:

```
$ python3 ej1.py
Traceback (most recent call last):
File "ej1.py", line 148, in pruebas
    assert p != None
AssertionError
```

Cuando todas las pruebas pasan correctamente, se imprime `OK`:

```
$ python3 ej1.py
ej1.py: OK
```

Pruebas

Se recomienda usar la instrucción `assert` de la biblioteca estándar para verificar condiciones en las pruebas. Ejemplo de uso:

```python
# función a probar
def sumar(a, b):
    return a + b

# pruebas
```

```python
def pruebas():
    assert sumar(0, 0) == 0
    assert sumar(2, 3) == 5
    assert sumar(2, -2) == 0

    from os import path
    print(f"{path.basename(__file__)}: OK")

pruebas()
```

Nota: A veces para depurar un error en las pruebas es útil imprimir valores; se permite el uso de `print()` para ello.

Nota: A veces para implementar las pruebas es útil utilizar números aleatorios. Se permite el uso de la biblioteca `random` para ello. En ese caso, se recomienda ejecutar `random.seed(0)` al inicio del programa para asegurar que la secuencia de números aleatorios sea siempre la misma, y así facilitar la depuración.

Ejercicios en lenguaje C

Para compilar y ejecutar el ejercicio `ej5.c`:

```
$ gcc -Wall -pedantic -std=c99 ej5.c -o ej5
$ ./ej5
ej5.c: OK
```

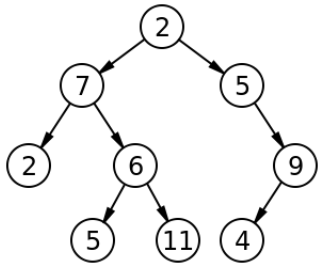
Pruebas

Se recomienda usar la función `assert` de la biblioteca estándar para verificar condiciones en las pruebas. Ejemplo de uso:

```c
#include <stdio.h>
#include <assert.h>

// funcion a probar
int sumar(int a, int b) {
    return a + b;
}

// pruebas
int main(void) {
    assert(sumar(0, 0) == 0);
    assert(sumar(2, 3) == 5);
    assert(sumar(2, -2) == 0);

    printf("%s: OK\n", __FILE__);
    return 0;
}
```

Ejercicios

Ejercicio 1 Un *árbol binario* es una estructura enlazada en la que cada nodo contiene referencias a otros dos nodos, llamados *hijo izquierdo* y *derecho* (pudiendo cualquiera de ellos ser una referencia nula).



Dado un *camino* formado por una secuencia de unos y ceros, se puede recorrer un árbol binario de la siguente manera:

- Se comienza en el nodo raíz (el primer nodo del árbol).
- Por cada elemento del camino:
  - Si es un 0, continuar hacia el hijo izquierdo
  - Si es un 1, continuar hacia el hijo derecho

Ejemplo: para el árbol mostrado arriba, el camino `"010"` conduce al valor 5.

Dada la clase `Nodo` que representa un nodo del árbol, se pide implementar el método `recorrer(camino)`, que devuelve el dato almacenado en el nodo resultante de recorrer el camino, o `None` si el camino no conduce a un nodo.

Sugerencia: pensar la función en forma recursiva.

Ejercicio 2 Bitspeak es un algoritmo simple que permite codificar secuencias de bytes (números entre 0 y 255) en palabras pronunciables. A cada byte se le asigna una sílaba, formada por una ”consonante” y una ”vocal” (entre comillas porque pueden estar formadas por mas de una letra), de la siguiente manera:

1. El byte se expresa en notación hexadecimal, que resulta en dos dígitos hexadexcimales (el prefijo `0x` indica notación hexadecimal). Ejemplos:

```
dec   hex
---  ----
  0  0x00
  9  0x09
 10  0x0a
 75  0x4b
255  0xff
```

2. Al grupo de 2 dígitos hexadecimales se le asigna una sílaba de la siguiente manera: el dígito más significativo determina la consonante y el dígito menos significativo la vocal, según la siguiente tabla:

```
Consonantes:
0x0: p, 0x1: b, 0x2: t, 0x3: d, 0x4: k, 0x5: g, 0x6: sh, 0x7: j,
0x8: f, 0x9: v, 0xa: l, 0xb: r, 0xc: m, 0xd: y, 0xe: s,  0xf: z

Vocales:
0x0: a,  0x1: e,  0x2: i,  0x3: o,  0x4: u,  0x5: an, 0x6: en, 0x7: in,
0x8: un, 0x9: on, 0xa: ai, 0xb: ei, 0xc: oi, 0xd: ui, 0xe: aw, 0xf: ow
```

Por ejemplo, la codificación de la secuencia de 2 bytes `[165, 8]` es `"lanpun"`, ya que expresando los números en notación hexadecimal, la secuencia es `[0xa5, 0x08]`.

Escribir la función `bitspeak(b)` que recibe una secuencia de bytes y devuelve la codificación Bitspeak correspondiente.

Ayuda: la función `hex(n)` convierte un número en su representación hexadecimal (incluyendo el prefijo `"0x"`, sin incluir ceros a la izquierda):

```
hex(0) -> "0x0"
hex(9) -> "0x9"
hex(10) -> "0xa"
hex(75) -> "0x4b"
hex(255) -> "0xff"
```

Ejercicio 3 Implementar la función `interseccion(a, b)`, que recibe dos listas ordenadas `a` y `b`, y devuelve la intersección de ambas, **en tiempo lineal**.

Las listas pueden contener elementos duplicados; la intersección debe incluir tantas copias del elemento como veces que aparece en ambas listas.

Ejemplos:

```
interseccion([2],          [3])          -> []
interseccion([2],          [2])          -> [2]
interseccion([2, 2],       [2])          -> [2]
interseccion([2, 2],       [2, 2])       -> [2, 2]
interseccion([1, 2, 2, 4], [1, 2, 2, 3]) -> [1, 2, 2]
```

Ejercicio 4 Escribir la función `tail(entrada, salida, n)`, donde `entrada` es la ruta a un archivo de texto existente, `salida` es la ruta a un archivo para escribir, y `n` es un número entero no negativo. La función debe escribir en `salida` las últimas `n` líneas del archivo `entrada`.

El archivo puede ser de un tamaño arbitrariamente grande. Solo se puede recorrer una vez, y en la memoria del programa se puede almacenar a lo sumo `n` líneas.

Ayuda: utilizar una cola para guardar las `n` líneas. Se puede utilizar la clase `Queue` del módulo `queue`:

```
>>> import queue
>>> q = queue.Queue()
>>> q.empty()  # cola vacía?
True
>>> q.put('a') # encolar
>>> q.put('b') # encolar
>>> q.get()    # desencolar
'a'
>>> q.get()    # desencolar
'b'
```

Ejercicio 5 Escribir en C la función `int buscar(const char *aguja, const char *pajar)` que devuelve la primera posición en la que la cadena `aguja` aparece en `pajar`, o `-1` si no aparece.

Ejemplo: `buscar("def", "abcdefhijk abcdefhijk") → 3`

Solo se permite utilizar la función `strlen` de la biblioteca estándar de C.