Algoritmos y Programación I (75.40)
Trabajo práctico n.º 2

Segundo cuatrimeste 2017

Consigna

El objetivo del trabajo práctico es implementar un **motor de búsqueda de archivos**, que ofrezca una línea de comandos interactiva permitiendo buscar archivos ubicados a partir del *directorio (carpeta) actual*[1] o cualquier subdirectorio.

Ejemplo

Dados los siguientes archivos ubicados en el directorio `/home/usuario/archivos`:

Ruta	Contenido
`a.txt`	`Hola! Este archivo contiene texto.`
`b.txt`	`Este archivo tambien contiene texto.`
`fotos/a.png`	(binario)
`fotos/foto-perfil.jpg`	(binario)

A partir de esta estructura de archivos, ejecutamos el programa habiéndonos parado en la misma carpeta `/home/usuario/archivos`:

[1] El directorio actual es el directorio a partir del cual se ejecuta el programa (que no necesariamente es el mismo directorio en el cual está el programa).

```
usuario@pc$ cd /home/usuario/archivos
usuario@pc$ python3 /home/usuario/tp2/tp2.py
Indexando archivos...
Listo! 5 archivos indexados.

>
```

A partir de este momento el programa presenta un *prompt* (`>`) en donde el usuario puede ingresar términos de búsqueda, y el programa lista la ruta completa de todos los archivos que coinciden (*match*) con el término de búsqueda.

Por ejemplo, si el usuario ingresa el término `archivo`:

```
> archivo
./a.txt
./b.txt

>
```

Los dos archivos que coinciden con `archivo` son `a.txt` y `b.txt` ya que incluyen la palabra "archivo" en el contenido. El programa muestra la ruta de ambos archivos, y luego vuelve a presentar el *prompt* para recibir un nuevo término de búsqueda.

Otro ejemplo: `a.txt` es el único archivo que contiene la palabra "Hola", y se considera como *match* con el término `hola`, ya que se ignoran mayúsculas y minúsculas:

```
> hola
./a.txt
```

Notar que para que se considere como *match* el archivo debe incluir la palabra completa:

```
> hol
No hay coincidencias.
```

También se considera como *match* cualquier parte de la ruta del archivo, incluyendo cualquier palabra que forme parte de los subdirectorios, el nombre y la extensión del archivo:

```
> fotos
```

```
./fotos/a.png
./fotos/foto-perfil.jpg

> foto
./fotos/foto-perfil.jpg

> a
./a.txt
./fotos/a.png

> jpg
./fotos/foto-perfil.jpg
```

Requerimientos

El programa debe cumplir con los siguientes requerimientos:

- Debe recorrer el directorio y subdirectorios y leer el contenido de los archivos **una sola vez**, cargando el *índice* (ver abajo) en memoria y manteniéndolo hasta que se termina la ejecución.
- Debe indexar **todos** los archivos que encuentra teniendo encuenta los términos que conforman la **ruta** del archivo.
- Además, todos los **archivos de texto** deben ser indexados teniendo en cuenta los términos que conforman el **contenido**. Un archivo se considera de texto si su extensión es `.txt`, `.md`, `.py` o `.c`. Para cualquier archivo que no tenga ninguna de esas extensiones, se ignorará su contenido.
- Tanto la **ruta** como el **contenido** debe ser indexado por **términos**. Se considera un término cualquier **secuencia consecutiva de caracteres alfanuméricos**. Además, se deben **ignorar mayúsculas y minúsculas**. Así, el texto "Abcd123/efg h! Xyz, 45 AS-DF" contiene los términos `abcd123`, `efg`, `h`, `xyz`, `45`, `as` y `df`.
- Debe ofrecer **3 modos de búsqueda**: *and*, *or* y *not*, detallados más abajo.

Proceso de indexado

El programa debe indexar todos los archivos que se encuentran en el *directorio actual* y todos sus subdirectorios. En Python, la función `walk` del módulo `os` permite esta funcionalidad. Notar que el directorio actual se representa con el nombre `.` (punto).

Nota: El separador de rutas de archivos es diferente según el sistema operativo (`\a\b\c` en Windows y `/a/b/c` en Linux y OSX). El programa desarrollado **debe funcionar correctamente independientemente del SO** en el que se ejecuta. Es decir, las rutas deben mostrarse utilizando el separador correcto. Para esto se recomienda utilizar la función `os.path.join` donde corresponda.

Índice invertido

Para ofrecer una búsqueda rápida, el programa debe leer los archivos una sola vez al inicio, y armar un *índice invertido*, que es una estructura que asocia cada uno de los términos existentes con la lista de rutas que coinciden con ese término.

Para el ejemplo presentado en la introducción, el índice invertido sería:

Término	Rutas
a	a.txt fotos/a.png
b	b.txt
txt	a.txt b.txt
fotos	fotos/a.png fotos/foto-perfil.jpg
foto	fotos/foto-perfil.jpg
perfil	fotos/foto-perfil.jpg
jpg	fotos/foto-perfil.jpg
png	fotos/a.png
hola	a.txt
este	a.txt b.txt
archivo	a.txt b.txt
contiene	a.txt b.txt
texto	a.txt b.txt
tambien	b.txt

Separación de términos

Se considera un término cualquier **secuencia consecutiva de caracteres alfanuméricos**. Una manera de obtener la lista de términos a partir de un texto en Python es utilizando la función `split` del módulo `re`[2]:

```
>>> import re
>>> re.split('\W+', "Abcd123/efg h! Xyz, 45 AS-DF")
['Abcd123', 'efg', 'h', 'Xyz', '45', 'AS', 'DF']
```

(Atención: la lista producida incluye mayúsculas y minúsculas.)

Modos de búsqueda

El programa debe soportar los siguientes tipos de consulta de búsqueda:

Búsqueda OR

En caso de introducir un término, se listarán los archivos que coinciden con el mismo:

```
> archivo
./a.txt
./b.txt
```

En caso de introducir más de un término, se listarán los archivos que coinciden con **al menos uno** de los términos:

```
> hola tambien
./a.txt
./b.txt
```

Por consistencia, también se puede introducir como consulta la palabra clave `OR:` y luego la lista de términos para obtener el mismo resultado:

```
> OR: hola tambien
./a.txt
```

[2] (Opcional) Para saber más acerca de cómo funciona el módulo `re`, leer acerca de **expresiones regulares**: https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

```
./b.txt
```

Búsqueda AND

En caso de introducir la palabra clave `AND:` más una lista de términos, se listarán los archivos que coinciden con **todos** los términos:

```
> AND: hola tambien
No hay coincidencias.

> AND: archivo tambien
./b.txt
```

Búsqueda NOT

En caso de introducir la palabra clave `NOT:` seguido de un término, se listarán los archivos que **no** coinciden con el término:

```
> NOT: hola
./b.txt
./fotos/a.png
./fotos/foto-perfil.jpg
```

Criterios de aprobación

A continuación se describen los criterios y lineamientos que deben respetarse en el desarrollo del trabajo.

Informe

El informe debe consistir en una descripción del **diseño** del programa.

Debe recordarse que la etapa de diseño es *anterior a la implementación*, por lo tanto debe describirse, utilizando texto y/o diagramas, cómo se va a estructurar el código para cumplir con las especificaciones de la consigna.

Algunas preguntas que deberían responderse:

- A grandes rasgos, ¿cómo será el flujo del programa? ¿Cuáles archivos se leen o escriben, y en qué momentos?
- ¿Qué estructura(s) de datos se usan para representar los datos en memoria?
- ¿Cómo se obtienen los resultados de búsqueda? ¿Cómo se implementan los distintos modos de búsqueda?

Código

Además de satisfacer las especificaciones de la consigna, el código entregado debe cumplir los siguientes requerimientos:

- El código debe ser claro y legible.
- El código debe estructurarse en funciones y, cuando corresponda, módulos. Las funciones deben definirse de la manera más genérica posible.
- Todas las funciones deben estar adecuadamente documentadas, y donde sea necesario el código debe estar acompañado de comentarios.

Entrega

La entrega del trabajo consiste en:

- El informe y código fuente impresos. Para el código fuente utilizar una tipografía `monoespacio`.
- El informe en formato *PDF*.
- Una versión digital de todos archivos `.py` de código, separados del informe. En el caso de ser más de un archivo, comprimidos en un `.zip`.

El informe impreso debe entregarse en clase. Los dos últimos (PDF y código fuente) deben enviarse utilizando el formulario de entregas ubicado en https://algoritmos1rw.appspot.com.

Este trabajo práctico se desarrolla en forma **individual**. El plazo de entrega vence el **lunes 23 de octubre de 2017**.