.UBAfiuba
FACULTAD DE INGENIERÍA

Guía de Ejercicios 1: Introducción a Rust

75.42 / 95.08 Taller de Programación I - 1er C 2021
Ing. Pablo Deymonnaz

Ejercicio 1 - Ahorcado

El objetivo del ejercicio es implementar un programa de consola para jugar al *ahorcado*.

```
Bienvenido al ahorcado de FIUBA!

La palabra hasta el momento es: _ _ _ _ _ _
Adivinaste las siguientes letras:
Te quedan 5 intentos.
Ingresa una letra: r

La palabra hasta el momento es: _ _ _ _ _ r
Adivinaste las siguientes letras: r
Te quedan 5 intentos.
Ingresa una letra: c
```

Si se ingresa una letra que no forma parte de la palabra, se pierde un intento.

La lista de palabras se debe leer de un archivo de texto, donde cada línea del archivo contendrá una palabra. De esa lista, se deberá elegir una palabra (puede ser una selección secuencial de palabras).

El programa termina cuando se adivina correctamente la palabra pensada, o cuando se acabaron los intentos.

Tips:

- Recuerda que las variables son inmutables por default. Para cambiar el estado de una variable, se la debe declarar como *mut*.
- Se puede comparar Strings usando: $==$
- Usa `println!(...)` para imprimir elementos en la salida estándar. Para imprimir una variable, puedes escribir algo como esto:

  ```
  println!("Contenido: {}", s);
  ```

- Para leer de la entrada estándar, se puede usar algo como esto:

  ```
  io::stdin()
  .read_line(&mut v)
  .expect("Error leyendo la linea.");
  ```

Parte B

Mostrar las letras que se ingresaron y que no forman parte de la palabra (las que hacen que se pierda un intento).

Verificar si se ingresó nuevamente una letra que ya estaba.

Parte C

Sobre la implementación de las funciones, modelizar una estructura de datos que represente al tipo de error de retorno. Por ejemplo: se agotaron los intentos. Basarse en el *enum* **Result**.

Ejercicio 2 - Contar palabras

Escribir un programa para contar las frecuencias de palabras únicas leídas de un archivo de entrada. Luego imprimirlas con sus frecuencias, ordenadas primero por las más frecuentes. Por ejemplo, dado este archivo de entrada:

```
La casa tiene una ventana
La ventana fue defenestrada
```

El programa debe imprimir:

```
ventana -> 2
La -> 2
casa -> 1
tiene -> 1
una -> 1
fue -> 1
defenestrada -> 1
```

Una solución básica consiste en leer el archivo línea por línea, convertirlo a minúsculas, dividir cada línea en palabras y contar las frecuencias en un *HashMap*. Una vez hecho esto, convertir el *HashMap* en una lista de pares de palabras y cantidad y ordenarlas por cantidad (el más grande primero) y por último imprimirlos.

Se debe seguir las siguientes recomendaciones:

- Para separar en palabras, se debe considerar los espacios en blanco, ignorando los signos de puntuación.
- Si la frecuencia de dos palabras es la misma, no importa el orden en el que aparecen las dos palabras en la salida impresa.
- No leer el archivo completo en memoria, se puede ir procesando línea por línea, o en conjuntos de líneas. Sí se puede mantener en memoria el hashmap completo.
- Usar solamente las herramientas de la biblioteca std del lenguaje.

Para leer un archivo línea por línea, se puede utilizar el método *read_line*[1].

Ejercicio 3 - Buscador Full-text

La búsqueda de texto está en todos lados. Desde encontrar un mensaje en redes sociales, productos en portales de comercio electrónico, o cualquier otra cosa en la web a través de Google.

En este ejercicio, construiremos un motor de búsqueda sencillo que pueda buscar en millones de documentos y clasificarlos según su relevancia.

El primer paso consiste en la **preparación de los datos**. Necesitamos construir el conjunto de datos sobre el que realizaremos las búsquedas, denominado **corpus**. Este conjunto será un grupo de archivos de texto plano (txt) que puede generarse a partir de artículos de Internet. Cada archivo será un *documento* que estará identificado por un *id*.

Luego se debe realizar la **indexación:** Se debe implementar una estructura conocida como de *índice invertido*. Que será una estructura de datos de tipo *HashMap* que contendrá como clave cada una de

[1] https://doc.rust-lang.org/std/io/trait.BufRead.html#method.read_line

las palabras y como valor, el o los ids de documentos en los que aparece la palabra. Para este paso, se debe realizar el proceso de *tokenización*, es decir, obtener cada una de los tokens que conforman al documento, considerando las separaciones de los mismos por espacios en blanco o saltos de línea, y quitando los signos de puntuación. De estos tokens, se debe ignorar las palabras más usadas del lenguaje español (conocidas como *stop words*), por ejemplo, los artículos: la, el, las, los. Se debe considerar la frecuencia de cada token, es decir, la cantidad de veces que el mismo aparece en el documento. Ese valor debe ser almacenado para el ordenamiento de los resultados.

El último paso es implementar la **búsqueda**. Para ello, se debe solicitar al usuario una frase a buscar y aplicar la tokenización de la misma y la eliminación de las stop words. Se debe buscar los documentos que contengan los términos de búsqueda ingresados.

Luego se debe determinar la relevancia de cada documento resultado de la búsqueda. Para esto, se debe determinar el *puntaje* del documento. Esto se puede computar a partir de sumar las frecuencias de cada uno de los términos encontrados.

Para mejorar el cálculo de puntaje del documento, calcularemos la frecuencia inversa de documentos para un término (denominado *tf-idf*) dividiendo la cantidad de documentos (N) en el índice por la cantidad de documentos que contienen el término, y tomaremos el logaritmo.

$$tf(t, D) = \log\left(\frac{|D|}{|d \in D : t \in d|}\right)$$

donde:

- $|D|$ es la cantidad de documentos del corpus.
- $|d \in D : t \in d|$ es el número de documentos donde aparece el término t. Si el término no está en la colección se producirá una división-por-cero. Por lo tanto, se suele ajustar esta fórmula a $1 + |d \in D : t \in d|$

Luego, multiplicaremos la frecuencia del término con la frecuencia inversa del documento durante nuestra clasificación, por lo que las coincidencias en términos que son raros en el corpus contribuirán más a la puntuación de relevancia.

Requerimientos no funcionales

Los siguientes son los requerimientos no funcionales para la resolución de los ejercicios:

- El proyecto deberá ser desarrollado en lenguaje Rust, usando las herramientas de la biblioteca estándar.
- No se permite utilizar *crates* externos. El único crate autorizado a ser utilizado es *rand*[2] en caso de que se quiera generar valores aleatorios.
- El código fuente debe compilarse en la versión stable del compilador y no se permite utilizar bloques unsafe.
- El código deberá funcionar en ambiente Unix / Linux.
- El programa deberá ejecutarse en la línea de comandos.
- La compilación no debe arrojar *warnings* del compilador, ni del linter *clippy*.
- Las funciones y los tipos de datos (*struct*) deben estar documentadas siguiendo el estándar de *cargo doc*.
- El código debe formatearse utilizando *cargo fmt*.
- Las funciones no deben tener una extensión mayor a 30 líneas. Si se requiriera una extensión mayor, se deberá particionarla en varias funciones.

[2] https://crates.io/crates/rand

- Cada tipo de dato implementado debe ser colocado en una unidad de compilación (archivo fuente) independiente.