Esercitazione 3

Esercizio 1

In questo esercizio affronteremo alcuni esempi di ricerca multipla dentro una struttura dati per familiarizzare con l'uso di lifetime in Rust, funzioni lambda e iteratori.
Non tutti gli esempi si compileranno e lo scopo è proprio capire perché il compilatore non permette alcune operazioni apparentemente lecite.

Nel file **es0301.rs** vengono fornite alcune funzioni senza annotazioni di lifetime e senza implementazione e occorre annotarle con i lifetime corretti e completare l'implementazione per ottenere i risultati attesi dagli esempi di utilizzo forniti.
Suggerimento: commentare tutto il file e scommentare una sola funzione per volta per concentrarsi su un solo problema, senza dover prima risolvere tutti i problemi di compilazione

Nella prima (funzioni subsequence1-3) si richiede di cercare delle sottosequenze di DNA all'interno di stringhe che rappresentano catene di DNA. Il DNA è una lunghissima catena di molecole di 4 tipi, che si indicano con i simboli A C G e T; quindi una sequenza di DNA può essere rappresentata da una stringa con i caratteri ACGT ripetuti a piacere.
Una particolare sottosequenza invece può essere rappresentata sinteticamente da una stringa del tipo "A1-2,T1-3,A2-2,G2-4,C2-2", dove il numero rappresenta il numero minimo massimo di ripetizioni della base corrispondente. Quindi questa stringa richiede di cercare una sequenza di lunghezza variabile con 1-2 A, 1-3 T, 2 A, 2-4 G, 2 C (esempio ATAAGGCC, ma anche AATTAAGGGCC).

Si richiede poi di implementare tre modi alternativi per ciclare sulle sottosequenze trovate:

- una funzione lambda passata alla funzione di ricerca e chiamata per ogni sottosequenza trovata (subsequence4)
- un iteratore naive realizzato tramite una struct, che permette di interrompere la ricerca in qualsiasi momento (SimpleDNAIter)
- un iteratore compliant con rust (DNAIter)
- un iteratore creato da un generatore, senza dover definire una struct di supporto (subsequence5_iter)

Vedere gli esempi in es0301.rs e i commenti che guidano alla soluzione

Esercizio 2

Realizzare un struct **FileSystem** che permetta di gestire la struttura (nomi e relazioni) di un file system in memoria, offrendo operazioni di: creazione, rimozione, ricerca, update di cartelle e file.

L’interfaccia da realizzare è fornita in file **es0302.rs**, comprese le indicazioni necessarie e un esempio d’uso.
Per compilare e far funzionare correttamente il codice sarà necessario annotare funzioni e metodi con gli opportuni lifetime.

Anche qui il suggerimento è commentare tutto e aggiungere un pezzo per volta in modo da non dover aggiustare subito tutti gli errori di compilazione.

Nota sui riferimenti mutabili
Attenzione in particolare all’uso della funzione find nell’esempio d’uso, quando si prova ad ottenere una get_mut() del path trovato. Non compilerà e non è possibile farla compilare senza un utilizzo diverso. Spiegare bene il motivo, tracciando i lifetime delle variabili coinvolte.
Successivamente commentare quel pezzo di codice e provare ad ottenere un riferimento mutabile dai path trovati usando il suggerimento nel codice.

Nota di teoria
Se fate attenzione la struttura filesystem è un albero, che avete realizzato senza usare un puntatore, ma solo una collezione standard Vec.
Questo potrebbe far pensare che un albero binario con un nodo definito così funzioni.

```
struct Node {
	val: i32,
	left: Node,
	right: Node
}
```

Se provate a compilare rust vi dà un errore di compilazione, qual è e come lo spiegate?

Se invece definire così la struttura Node (che è l’approccio usato in FileSystem), funziona

```
struct Node {
	val: i32,
	children: Vec<Node>
}
```

Qual è la differenza tra il primo e il secondo esempio? Dove vengono allocati i dati?