Esercitazione 3

Esercizio 1

Implementare il buffer circolare descritto in exercism.

Attenzione: per definizione il buffer in cui memorizzare i valori T non va riallocato e modificato, quindi non si possono usare le funzioni avanzate di Vec che permettono di fare push e pop di valori. Infatti lo scopo del circular buffer è proprio quello di avere un buffer dove si possono leggere e scrivere valori garantendo I tempi, senza penalità di riallocazione del buffer: non è solo imporre una dimensione massima al buffer!
Questo in Rust presenta un "problema": quando si legge un valore dal buffer si deve liberare la cella, e il valore viene restituito con una move perché non è detto che T implementi il trait copy. Ma se si fa una move da un elemento del vettore che rimane nell'indice corrispondente? Rust non permette di lasciare una cella senza che ci sia un valore definito di cui qualcuno abbia l'ownership...
Il problema si può risolvere in due modi:

- sostituire un valore al posto del precedente, ma quale valore? C'è un valore di default di T? Vedere se esiste un tratto che può garantire tale condizione
- se le celle possono avere o non avere un valore di tipo T c'è una Enum apposta che permette di gestire la situazione: quale? *Option<T>*

Risolvere il problema con entrambe le strategie.

Esercizio 2

Realizzare la **`struct MyCycle<I: Clone+Iterator>`** che implementa un iteratore circolare a partire da un iteratore base generico la cui sequenza dovrà essere ripetuta n volte (n=0 ⇒ infinite volte). Quando l'iteratore base finisce, MyCycle deve essere in grado di usare un clone dell'iteratore ricevuto come parametro iniziale e ricominciare dal principio una sequenza, andando avanti così fino al numero di volte specificato all'atto della creazione.

Poiché MyCycle è un iteratore a sua volta, deve implementare il tratto Iterator. MyCycle può essere costruito così:

```
MyCycle::new(iter: I, repeat: usize);
```

Realizzare alcuni unit test, provando anche i seguenti casi:

- fornendo in ingresso un iteratore con zero elementi si deve ottenere un iteratore che restituisce zero elementi

- costruendo un MyCycle (con numero di ripetizioni finito pari a n1) a partire da un altro MyCycle (con numero di ripetizioni finito pari a n2), si deve ottenere una sequenza che contiene n1*n2 volte la sequenza originale
- concatenando (tramite `.chain(...)`) due MyCycle, il primo basato su una sequenza di l1 elementi con numero di ripetizioni pari a n1, il secondo basato su una sequenza di l2 elementi con numero di ripetizioni pari a n2, si deve ottenere una sequenza di `l1*n1+l2*n2` elementi
- facendo lo zip di due MyCycle si ottiene una sequenza formata dalle coppie ordinate ottenute dalle rispettive sequenze

Esercizio 3

Un file di testo contiene un calendario definito nel seguente modo:

```
start_day
end_day
sched1_start
sched1_end
sched2_start
sched2_end
...
```

La prima riga contiene l'ora di inizio e quella fine della giornata lavorativa, mentre ogni riga successiva contiene una serie di appuntamenti fissati come intervalli inizio-fine. Ogni ora è espressa nel formato `hh:mm`. Gli intervalli non si sovrappongono e sono ordinati per ora di inizio.

Scrivere un programma che ,dati due file di calendario, e invocato con una durata espressa in minuti, stampi tutti gli intervalli disponibili per fissare un appuntamento.

Contenuto del file cal1:

```
08:00
20:00
10:00
11:30
14:00
16:00
```

8-10

11:30-14

16-20

9-9:30

12-13:30

16:30-18

TUPLE COME ELEMENTI PER INDICARE LE ORE
(HOUR, MIN)

Contenuto del file cal2:

```
09:00
18:00
09:30
12:00
13:30
16:30
```

9-9:30 x

12-13:30 x

16:30-18 x

un possibile approccio potrebbe essere prima trovare tutti gli intervalli disponibili per ogni calendario

poi, per il calendario con i bounds più piccoli, prendere ogni intervallo e controllare se è contenuto in almeno un intervallo del calendario più grande >> se lo è, è valido

cargo run -- cal1 cal2 30

```
09:00 09:30
12:00 13:30
16:30 18:00
```

I calendari una volta letti vanno memorizzati in una struttura:

```
struct Calendar<T> {
schedule: Vec<(T, T)>,
bounds: (T,T)
}
```

dove `T` è un placeholder per un tipo da definire (suggerimento: le stringhe non sono il tipo più comodo per gestire le ore). Come algoritmo ci sono due possibili approcci efficienti (complessità O(M+N) dove M e N sono le lunghezze degli appuntamenti):

1. creare una unica lista di intervalli e trovare "i buchi", ma richiede l'allocazione di memoria aggiuntiva (il vettore con entrambi gli appuntamenti)
2. usare due iteratori, uno per ciascuna lista di appuntamenti, che avanzano in modo indipendente e man mano che si trova un "buco" memorizzare l'intervallo; questo non richiede memoria aggiuntiva.

Provare a risolverlo con la strategia 2 senza allocare memoria aggiuntiva.

Inoltre scrivere alcuni unit test per provare la funzione ricerca intervalli liberi. Alcuni casi suggeriti:

- uno e entrambi i calendari senza appuntamenti
- un calendario pieno
- un calendario con orari liberi ma di lunghezza insufficiente
- tempo disponibile a inizio e fine giornata
- un intervallo disponibile lungo esattamente quanto richiesto

Esercizio 4

Risolvere il problema di exercism chiamato "React".

L'obiettivo generale è realizzare una versione semplificata di foglio elettronico, dove alcune celle contengono valori di ingresso e altre celle sono calcolate dinamicamente da una funzione, sulla base di valori presenti in altre celle. Il codice di partenza presente in exercism mostra l'interfaccia richiesta. In più l'esercizio richiede di poter impostare delle funzioni callback che dovranno essere invocate ogni volta che un valore derivato cambia. Per questo tipo di problema sono possibili due approcci:

1. "lazy", ovvero una volta impostati i valori di ingresso, non viene calcolato il valore di nessuna cella derivata, se non qualora si provi a leggere il contenuto di tali celle
2. "sincrono": quando si imposta un valore in ingresso, tutti i valori derivati da questo input vengono immediatamente aggiornati.

Entrambi i metodi hanno pro e contro che dipendono dal contesto di utilizzo (ad esempio in un foglio elettronico vero vorreste vedere tutte le celle derivate aggiornate, non solo quando vengono cliccate);

qui si chiede di risolverlo in modo sincrono. Il motivo si vedrà nella prossima esercitazione in cui verrà richiesto divparallelizzare il calcolo delle celle derivate per velocizzare l'approccio sincrono.

Suggerimenti (NB: leggerli dopo aver visto la base lib.rs di exercism o avranno poco senso; le classi non sono vincolanti e neppure complete). Nelle celle derivate vi serve memorizzare un valore per confrontare se è cambiato, ma all'inizio può non essere settato: si può usare `Option`.

Per memorizzare la funzione di calcolo dovete usare uno smart pointer in quanto il compilatore vi dice che la dimensione dell'oggetto con il trait Fn non può essere determinato a compile time. Esempio:

```
struct ComputeCell<T> {
val: Option<T>,
deps: Vec<CellId>,
fun: Box<dyn Fn(&[T]) -> T>,
callbacks: HashMap<CallbackId, Box<dyn FnMut(T)>>
}
```

Per evitare di ciclare su tutte le celle ad ogni aggiornamento, potete tenere una struttura (inv_deps) che mappa globalmente tutte le dipendenze tra celle; idem per quanto riguarda le callback

```
pub struct Reactor<T> {
inputs: Vec<T>,
cells: Vec<ComputeCell<T>>,
inv_deps: HashMap<CellId, HashSet<ComputeCellId>>,// cellid -> dep to compute only
dirty cells
cbcells: HashMap<CallbackId, ComputeCellId>,
cb_ids: usize
}
```