Esercitazione 5

Esercizio 1: parallelizzazione del lavoro

Un rompicapo prevede di trovare la sequenza di operazioni elementari (somma, sottrazione, moltiplicazione e divisione) necessarie per ottenere 10, partendo da 5 numeri da 0 a 9 casuali. I vincoli sono:

- le cinque cifre sono comprese tra 0 e 9 e possono ripetersi
- le cifre devono essere utilizzate tutte, in qualsiasi ordine
- non ci sono limiti sulle operazioni (es. vanno bene anche quattro somme)
- non si considera la precedenza degli operatori, le operazioni vanno applicate da sinistra a destra secondo il loro ordine
- se una divisione è per zero o non dà un intero come risultato non è una operazione valida

Esempio: dato 2 7 2 2 1 una soluzione può essere 7 - 2 - 1 x 2 + 2 = 10

Scrivere un programma che, letta la sequenza di cifre da command line come argomento, trovi tutte le possibili soluzioni, se ve ne sono, le salvi in una collezione di stringhe (es: “7 - 2 - 1 x 2 + 2”) e la stampi. Nella collezione le stringhe devono essere uniche.

Per risolverlo utilizzare un approccio *brute force*, ovvero elencare in un vettore tutte le possibili *permutazioni* delle cinque cifre scelte, ciascuna unita a tutte le *combinazioni con ripetizione* delle quattro operazioni elementari. Calcoliamo poi il risultato per ognuna di esse e se è 10 la sequenza viene salvata, altrimenti viene scartata.
Essendo cifre e simboli delle operazioni di tipo differente utilizzare un vettore di tuple per memorizzare tutte le possibili combinazioni: il primo elemento sono le 5 cifre permutate, il secondo le operazioni.

Esempio:

```
[
        ([2,7,2,2,1], [‘+’, ‘+’,’+’, ‘+’]), // = 14
        ([2,7,2,2,1], [‘+’, ‘+’,’+’, ‘-’]), // = 12
        …
        ([1,7,2,2,2], [‘+’, ‘+’,’+’, ‘+’]), // = 14
]
```

Una volta verificato il corretto funzionamento sfruttiamo i thread per provare a velocizzare la ricerca delle soluzioni in parallelo. Si divide il vettore di tutte le possibili permutazioni in K blocchi uguali e per ciascuna si lancia un thread che calcola i risultati delle combinazioni. Provare con K=2,3,4… ecc, misurare i tempi e trovare il numero di thread oltre il quale non vi sono vantaggi (se ve ne sono).
Cambia qualcosa se la divisione del lavoro fra thread anziché essere a blocchi è

interleaved? Vale a dire con tre thread il primo prova le permutazioni con indice 0,3,6,... il secondo 1,4,7,... e il terzo 2,5,8,...

Se non vi sono vantaggi provare con n = 6 o 7 cifre invece di 5. Nota: le operazioni sono molto leggere, è probabile che siano necessari valori di n grandi per ottenere vantaggi misurabili

Suggerimento: per generare permutazioni e combinazioni usare il crate `itertools` e vedere i metodi `permutations()` e `repeat_n(...).multi_cartesian_product()`.

Esercizio 2: condivisione risorse fra thread

Adattare l'esercizio sul buffer circolare delle esercitazioni precedenti facendo sì che producer (chi scrive i valori) e consumer (chi legge i valori) vengono eseguiti in due thread differenti

Per testare la sincronizzazione scrivere un valore al secondo nel buffer e leggere ogni 2 e poi invertire i tempi. Così testiamo caso di buffer mediamente full e mediamente empty.

Attenzione che in questo caso l'accesso al buffer e agli indici di scrittura e lettura va sincronizzato, garantendo la mutua esclusione fra i due thread.
Il buffer quindi dovrà essere una struct RingBuf con due metodi, read() -> Option<T> e write(val) -> Result<(),()> che siano *thread safe*, quindi invocabili all'interno di thread senza che il chiamante debba preoccuparsi delle operazioni di sincronizzazione.

read restituisce None quando il buffer è vuoto. mentre la write restituisce Err(()) quando il buffer è pieno e Ok(()) quando la scrittura è avvenuta con successo.

Bonus: modificare producer e consumer in modo che scrivano e leggano sul buffer senza pause (se non quando è pieno o vuoto) e misurare il throughput del buffer con dimensioni differenti (10, 1000, 10000 valori). Misurare anche il throughput aumentando il numero di producer e consumer, come varia?

Esercizio 3: sincronizzazione

Una barriera è un pattern di sincronizzazione che permette a *n* thread di attendere che tutti arrivino a un punto comune prima di andare avanti.
La barriera è "chiusa" fino a quando tutti i thread non arrivano ad un certo punto (esempio un risultato pronto) e viene "aperta" quando l'ultimo arriva all'ingresso della barriera.

La barriera viene inizializzata con il numero di thread attesi (n) e quando un thread chiama `barrier.wait()` si ferma finché l'ultimo non chiama `barrier.wait()`.

Si dice *ciclica* una barriera che può essere riusata. In questo caso i thread hanno un loop che fa del lavoro e periodicamente chiamano `barrier.wait()`, che ha sempre lo stesso

comportamento.
La barriera quindi è inizialmente chiusa, poi quando tutti i thread sono arrivati viene aperta e quando esce l'ultimo viene nuovamente chiusa per bloccare i thread al prossimo loop.

Attenzione!!! Questo implica che occorre gestire eventuali thread troppo veloci, che, una volta usciti chiamano subito una seconda wait() mentre la barriera è ancora aperta perché dei thread non sono ancora usciti.
(questa è una situazione difficilmente debuggabile, ma va tenuta in conto durante la progettazione della barriera; per provarlo si possono provare introdurre ritardi casuali all'uscita dalla wait)

Un esempio di come può essere usata:

```
fn main() {
    let abarrrier = Arc::new(cb::CyclicBarrier::new(3));

    let mut vt = Vec::new();

    for i in 0..3 {
        let cbarrier = abarrrier.clone();

        vt.push(std::thread::spawn(move || {
            for j in 0..10 {
                cbarrier.wait();
                println!("after barrier {} {}", i, j);
            }
        }));
    }

    for t in vt {
        t.join().unwrap();
    }
}
```

In questo esempio i thread avanzano con i rispettivi indici "j" sincronizzati, nessun thread avanza più velocemente degli altri. Provate a vedere la differenza commentando wait().

La barriera ha due stati di funzionamento:

- chiusa: si aspetta che giungano tutti i thread
- aperta: lascia andare avanti i thread

Quando è aperta in uscita occorre evitare che i thread chiamino di nuovo wait rientrino.
Idealmente si può pensare come le porte doppie delle banche: finché non è aperta quella verso il fuori, non viene aperta quella verso il dentro e viceversa.
Questo implica che non basta un semplice contatore di quanti thread sono in attesa, ma occorre anche salvarsi uno stato e segnalare agli altri thread quando la barriera viene aperta

Per risolvere il problema è quindi necessaria anche una condition variable

Provare inoltre a passare un valore *T generico* alla wait e condividere fra tutti i thread il valore passato => la wait deve restituire un vettore con gli n valori raccolti