Assumiamo un array di int A inizializzato con [30,10,20] e condiviso da due thread.
Un thread esegue l'istruzione A[0] = A[0] + A[1]
L'altro thread esegue l'istruzione A[0] = A[0] * A[2]
Si argomenti in modo formale se possono verificarsi race condition su A.

Funzione che formalizza l'operazione del primo thread f1: R -> R definita, per parti, come segue
(cioè A[0] = A[0] + A[1]):
f1 ([a, b, c]) = [a + b, b, c].
Nota: in particolare vale che:
f1 ([30, 10, 20]) = [30 + 10, 10, 20] = [40, 10, 20] e
f1 ([600, 10, 20]) = [600 + 10, 10, 20] = [610, 10, 20].
Funzione che formalizza l'operazione del secondo thread f2: R -> R definita, per parti, come segue
(cioè A[0] = A[0] * A[2]):
f2 ([a, b, c]) = [a * c, b, c].
Nota: in particolare vale che:
f2 ([30, 10, 20]) = [30 * 20, 10, 20] = [600, 10, 20] e
f2 ([40, 10, 20]) = [40 * 20, 10, 20] = [800, 10, 20].
Valori ammissibili su A, con A inizializzato con [x, y, z], cioè valori che otterremmo in caso di esecuzione sequenziale delle due operazioni:
f2 (f1 ([x, y, z])) = f2 ([x + y, y, z]) = [(x + y) * z, y, z]
e
f1 (f2 ([x, y, z])) = f1 ([x * y, y, z]) = [(x * y) + y, y, z]
Nel nostro esempio:
f2 (f1 ([30, 10, 20])) = [(30 + 10) * 20, 10, 20] = [40 * 20, 10, 20] = [800, 10, 20]
f1 (f2 ([30, 10, 20])) = [(30 * 20) + 10, 10, 20] = [600 + 10, 10, 20] = [610, 10, 20]
Possiamo pertanto dire che avremo race condition, perché l'array può assumere, per esempio, il valore [40, 10, 20] che è diverso dai due valori ammissibili.
Il primo thread esegue la sua operazione, perde il processore, il secondo thread procede con la sua operazione ed imposta in A[0] il nuovo valore. Il primo thread riacquisisce il processore ed imposta in A[0] il nuovo valore, che va a sovrascrivere quello precedente, erroneamente, poiché i suoi dati in input non erano ancora quelli modificati dal secondo thread.

Assumiamo un array di int A inizializzato con [10,20].
- Un thread esegue l'operazione A[0] = A[1]+5.
- L'altro thread esegue l'operazione A[1] = A[0]+55.
Si argomenti in modo formale se possono verificarsi race condition su A.

Funzione che formalizza l'operazione del primo thread f1: R -> R definita, per parti, come segue
(cioè A[0] = A[1] + 5):
f1 ([a, b]) = [b + 5, b].
Nota: in particolare vale che:
f1 ([10, 20]) = [20 + 5, 20] = [25, 20]
f1 ([10, 65]) = [65 + 5, 65] = [70, 65]
Funzione che formalizza l'operazione del secondo thread f2: R -> R definita, per parti, come segue
(cioè A[1] = A[0] + 55):
f1 ([a, b]) = [a, a + 55].
Nota: in particolare vale che:
f2 ([10, 20]) = [10, 10 + 55] = [10, 65]
f2 ([25, 20]) = [25, 80] = [25, 80]
Varoli ammissibili su A, con A inizializzato con [x, y], cioè valori che otterremmo in caso di esecuzione sequenziale delle due operazioni:
f2 (f1 ([x, y])) = f2 ([y + 5, y]) = [y + 5, (y + 5) + 55] = [y + 5, y + 60]
f1 (f2 ([x, y])) = f1 ([x, x + 55]) = [(x + 55) + 5, x + 55] = [x + 60, x + 55]
Nel nostro esempio:
f2 (f1 ([10, 20])) = f2 ([20 + 5, 20 + 60]) = [25, 80]
f1 (f2 ([10, 20])) = f1 ([10 + 60, 10 + 55]) = [70, 65]
Possiamo pertanto dire che avremo race condition, perché l'array può assumere, per esempio, il valore [25, 65] che è diverso dai due valori ammissibili.
Il primo thread esegue la sua operazione, perde il processore, il secondo thread procede con la sua operazione ed imposta in A[1] il valore 65. Il primo thread riacquisisce il processore ed imposta in A[0] il valore 25, che va a sovrascrivere quello precedente, erroneamente, poiché i suoi dati in input erano stati presi senza contare il cambio di valore eseguito dal secondo thread.

Assumiamo che due thread condividano una variabile X, con valore iniziale X = 100.
Il primo thread esegue l'istruzione X = 50;
Il secondo thread esegue l'istruzione if (x > 70) {X = 200;} else {X = 0;};
Discutere, formalmente, se è possibile che si verifichino race condition sulla variabile X.

L'operazione eseguita dal primo thread può essere formalizzata con la funzione f1: R -> R definita come segue (x è un valore reale, non va confuso con la variabile X):
f1 (x) = 50
L'operazione eseguita dal secondo thread può essere formalizzata con la funzione f2: R -> R definita, per parti, come segue:
f2 (x) = 200, f2 (x) = 0, se x > 70
se x <= 70
Pertanto, componendo le funzioni come segue, spieghiamo il comportamento dei thread quando la loro attività viene eseguita sequenzialmente in ordine arbitrario:
f1 (f2 (x)) = f1 (200) = 50, se x > 70
f1 (f2 (x)) = f1 (0) = 50, se x <= 70
e
f2 (f1 (x)) = f2 (50) = 0
Possiamo pertanto dire che avremo race condition se e solo se X può acquisire valori diversi da 50 e 0.
Una possibile esecuzione è la seguente: il secondo thred testa la guardia (X > 70), che risulta vera, perde il processore, il primo thread esegue l'assegnamento, il secondo thread riacquisisce il processore e assegna il valore 200 alla variabile X, che non verrà modificato in seguito.
Essendo 200 diverso da 50 e da 0, concludiamo che questa esecuzione da luogo a race condition

Si assuma la variabile X condivisa da due thread che invocano il seguente metodo M:
public void M () {if (X >= 0) {X = -5;} else {X = 12;}}
Discutere formalmente se si possono verificare race condition su X.

L'effetto sulla variabile X dell'esecuzione di ognuno dei due thread può essere formalizzato con la funzione f: R -> R definita, per parti, come segue:
- f(y) = 12, se y < 0;
- f(y) = - 5, se y >= 0;
Pertanto, il comportamento dei due thread che eseguono sequenzialmente può esser formalizzato applicando due volte f:
- f(f(y)) = f(12) = - 5, - f(f(y)) = f(-5) = 12, se y < 0;
se y >= 0;
Pertanto, se il valore iniziale di X è non negativo, per non avere race condition la variabile X deve assumere il valore finale 12.
Altrimenti, se il valore iniziale di X è negativo, per non avere race condition la variabile X deve assumere il valore finale -5.
Assumiamo che inizialmente X valga 10. Un thread potrebbe testare la guardia e perdere il processo prima di eseguire il ramo "then".
L'altro thread potrebbe eseguire tutto il metodo.
Il primo thread riprenderebbe in seguito l'esecuzione ed eseguirebbe il ramo "then".
Alla fine, X varrebb e -5, ma l'unico valore ammissibile partendo da X = 10 è 12, pertanto in questo caso avremmo race condition.

Si assumano una variabile int X e una variabile int Y condivise da due thread.
Il valore iniziale di X è 0, il valore iniziale di Y è 100.
Un thread esegue l'operazione X=Y+50.
L'altro thread esegue l'operazione Y=Y+30.
Spiegare formalmente se si possono verificare race condition su X e/o Y.

Le operazioni svolte dai due thread possono essere formalizzate mediante le funzioni
f1,f2: Z x Z --> Z x Z,
definite come segue:
f1(a , b) = (b+50 , b)
f2(a , b) = (a, b+30)

Pertanto:
f1(f2(u , v)) = f1(u , v+30) = (v+80 , v+30)
f2(f1(u , v)) = f2(v+50 , v) = (v+50 , v+30)

In particolare,
f1(f2(0 , 100)) = (100+80 , 100+30) = (180,130)
f2(f1(0 , 100)) = (100+50 , 100+30) = (150,130)

Si ha race condition se e solo se eseguendo i due thread in concorrenza si ottengono valori di X e Y diversi da (180,130) e (150,130). Analizzando il codice dei thread emerge che ciò è impossibile, pertanto non si possono avere race condition.

Due thread condividono la variabile int x. I due thread chiamano il seguente metodo
public void m( ){
if (x%2 == 0) { x=5; } else { x=4; } }
Argomentare formalmente se possono verificarsi race condition sulla variabile x.

L'operazione svolta sulla variabile x dal metodo m può essere formalizzata dalla funzione f : Z —> Z definita (per parti) come segue:

f(u) = 5, se u è pari
f(u) = 4, se u è dispari.

Nota: entrambi i thread chiamano lo stesso metodo, pertanto la funzione che spiega il loro comportamento è la medesima.

Pertanto, f(f(u)) = 4, se u è pari, mentre f(f(u)) = 5, se u è dispari.

Assumendo il valore iniziale di x pari, si ha r.c. se x può assumere un valore diverso da 4.
Questo è possibile, perché entrambi i thread che eseguono m potrebbero eseguire il ramo "then", e il valore finale di x sarebbe pertanto 5.
In particolare, ciò è possibile se il processore viene perso dal primo thread che testa la guardia prima che il valore della variabile venga cambiato.
Assumendo il valore iniziale di x dispari, il ragionamento è analogo.