

---

ANNO ACCADEMICO 2024/2025

---

# Modelli Concorrenti e Algoritmi Distribuiti

---

Teoria

Altair's Notes



**UNIVERSITÀ  
DI TORINO**



---

DIPARTIMENTO DI INFORMATICA

---



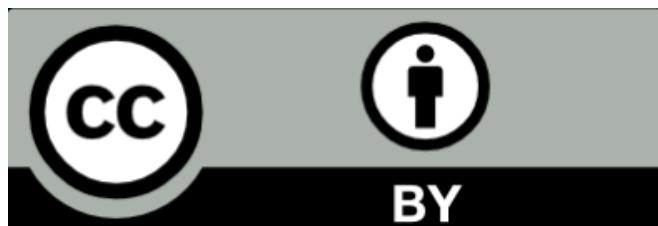
<b>CAPITOLO 1</b>	<b>INTRODUZIONE ALLA PROGRAMMAZIONE CONCORRENTE</b>	<b>PAGINA 5</b>
1.1	Il corso in breve... Cosa si intende per programmazione concorrente? — 5 • Cosa si intende per algoritmo distribuito? — 6	5
1.2	Parallelismo Interleaving di Istruzioni Atomiche — 6 • Sistemi Monoprocessoress e Sistemi Multiprocessoress — 10	6
1.3	Correttezza di Programmi Concorrenti Introduzione e Proprietà — 13 • La Correttezza di Programmi — 15 • Il Problema della Mutua Esclusione — 15 • Specifiche di Correttezza — 17 • Statement Atomici Particolari — 22 • Invarianti e Predicati — 23	13
<b>CAPITOLO 2</b>	<b>TEST2</b>	<b>PAGINA 26</b>



# Premessa

## Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/licenses/by/4.0/>).



## Formato utilizzato

Box di "Concetto sbagliato":

### Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

### Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

### Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

### Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

### Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

**Box di "Note":**

**Note:-**

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

**Box di "Osservazioni":**

**Osservazioni 0.0.1**

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.



# 1

## Introduzione alla Programmazione Concorrente

### 1.1 Il corso in breve...

#### 1.1.1 Cosa si intende per programmazione concorrente?

La programmazione concorrente nasce con i *sistemi concorrenti* nell'ambito dei *sistemi operativi* con il concetto di *processo* (o *thread*).

##### Definizione 1.1.1: Sistema concorrente

Un sistema concorrente è un sistema *software* implementato su una *piattaforma hardware* in grado di eseguire *contemporaneamente* più attività diverse che condividono *risorse comuni*<sup>a</sup>.

<sup>a</sup>Porzioni di memoria centrale, CPU, etc.

##### Note:-

Un esempio di sistemi concorrenti sono i *sistemi operativi multiprogrammati*.

##### Corollario 1.1.1 Programma concorrente

Un programma concorrente è un insieme di *moduli sequenziali* che possono essere eseguiti in parallelo.

#### Tipi di parallelismo:

- ⇒ *Parallelismo reale*: l'esecuzione dei moduli è realmente sovrapposta nel tempo;
- ⇒ *Parallelismo apparente*: si applica la tecnica di *interleaving delle istruzioni*.

##### Note:-

In entrambi i casi il termine *concorrenza* si utilizza come un'astrazione per studiare il parallelismo.

#### In questo corso si tratterà di:

- Introduzione ai *principi* della programmazione concorrente;
- Analisi dei principali *costrutti linguistici* per la programmazione concorrente;
- Applicazione di questi costrutti a vari problemi di *sincronizzazione* e *comunicazione* in programmi concorrenti;
- Studio delle *proprietà di correttezza* dei programmi concorrenti: *no deadlock, no starvation*, etc.

### 1.1.2 Cosa si intende per algoritmo distribuito?

#### Definizione 1.1.2: Sistema distribuito

Un sistema distribuito è un sistema composto da *più computer* che non condividono la memoria o altre risorse, ma sono connessi da canali di comunicazione<sup>a</sup>.

<sup>a</sup>Debolmente connessi.

#### Corollario 1.1.2 Algoritmo distribuito

Un algoritmo distribuito è un algoritmo progettato per essere eseguito da un sistema distribuito.

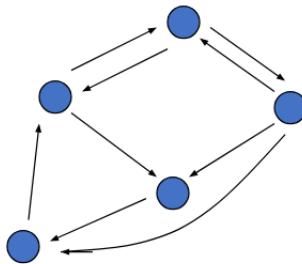


Figure 1.1: Schema di un sistema distribuito

#### Obiettivi per la parte di corso di algoritmi distribuiti:

- Introduzione di modelli formali che permettano l'analisi di *algoritmi distribuiti di base*;
- Studio della *correttezza* e delle *prestazioni* degli algoritmi distribuiti presentati;
- Analisi di *algoritmi distribuiti* in presenza di *malfunzionamenti* (algoritmi fault tolerant).

## 1.2 Parallelismo

### 1.2.1 Interleaving di Istruzioni Atomiche

#### Definizione 1.2.1: Processo

Modulo *sequenziale* di un programma concorrente (a volte si usa il termine thread).

#### Note:-

Per gli scopi di questo corso processo e thread vengono assunti come sinonimi.

#### Domanda 1.1

Che cos'è l'interleaving?

#### Definizione 1.2.2: Interleaving

Si suppone che ogni esecuzione di un programma concorrente sia ottenuta *interfogliando in maniera arbitraria* le istruzioni dei vari processi.

#### Note:-

Il risultato dell'interleaving è detto *computazione* o *scenario*.

```

begin
:
cobegin
    process P():
        ...
    process Q():
        ...
coend
:
end

```

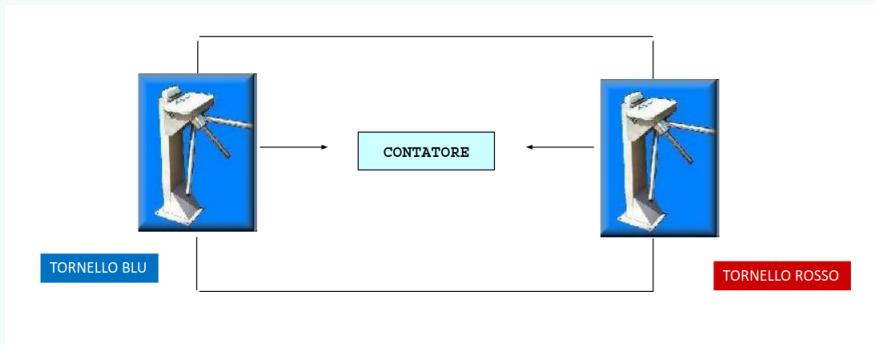
Figure 1.2: Pseudocodice

Nome del programma	
begin	:
P	Q
...	...
end	

Figure 1.3: Tabella

**Esempio 1.2.1** (Quante persone sono entrate in laboratorio?)

Si ha un laboratorio con due tornelli che hanno un contatore condiviso che viene incrementato di 1 quando entra una persona.



Si inizia dichiarando la variabile condivisa *counter* (inizializzata a 0). Dopo di ché si suppone che il tornello blu faccia entrare 100 persone e il tornello rosso altre 100.

```

begin
    int counter ← 0 //variabile condivisa tra i due processi concorrenti
    cobegin
        process tornello_blu():
            for (int j ← 0; j<100; j++):
                sleep(500 + (int)(math.random()*1000)) // prossimo arrivo
                counter++

        process tornello_rosso():
            for (int j ← 0; j<100; j++):
                sleep(500 + (int)(math.random()*1000)) // prossimo arrivo
                counter++

    coend
    System.println("Numero ingressi: " + counter)
end

```

```

program Tornelli
    int counter ← 0
    process tornello_blu | process tornello_rosso
        for (int j ← 0; j<100; j++) :
            sleep(500 + int (math.random)...)
            counter++
    System.println ("Numero ingressi:" + counter)

```

Alla fine della simulazione verrà stampato il valore 200? Dipende.

- Se l'istruzione **counter++** è atomica e indivisibile verrà stampato il valore 200;
- Ma se fosse realizzata mediante più load, add e store? In tal caso non è garantito che il risultato sarà 200.

### Definizione 1.2.3: Istruzione atomica

Un'istruzione viene detta atomica se viene sempre eseguita interamente senza possibilità di interruzioni<sup>a</sup>.

<sup>a</sup>No interleaving.

#### Note:-

Il risultato dell'esecuzione "simultanea" di due istruzioni atomiche è lo stesso che si otterrebbe dalla loro esecuzione sequenziale indipendentemente dall'ordine. In generale si assumerà che gli assegnamenti e le valutazioni di espressioni logiche siano operazioni atomiche.

### Esempio 1.2.2 (Possibile computazione)

Diamo delle etichette alle istruzioni (indivisibili) dei programmi P e Q

<b>Main</b>	
int a ← 0	
int b ← 0	
P	Q
p1: a ← a+1	q1: a ← a+2
p2: b ← b+1	q2: b ← b+2
end	

Le possibili computazioni sono:

p1, q1, p2, q2	q1, p1, q2, p2
p1, q1, q2, p2	q1, p1, p2, q2
p1, p2, q1, q2	q1, q2, p1, p2

È importante notare che p2 non può precedere p1 e q2 non può precedere q1.

Algoritmo: Istruzioni Atomiche di Assegnamento		
integer n ← 0		
<b>p</b>		<b>q</b>
p1: n ← n + 1		q1: n ← n + 1

Per l'algoritmo riportato sopra sono possibili due diversi **scenari**.

Process p	Process q	n	Process p	Process q	n
<b>p1: n←n+1</b>	q1: n←n+1	0	p1: n←n+1	<b>q1: n←n+1</b>	0
(end)	<b>q1: n←n+1</b>	1	<b>p1: n←n+1</b>	(end)	1
(end)	(end)	2	(end)	(end)	2

In entrambi i casi sia avrà sempre n = 2 come valore finale.

Algoritmo: Assegnamento con riferimento globale		
integer n ← 0		
<b>p</b>		<b>q</b>
integer temp		integer temp
p1: temp ← n		q1: temp ← n
p2: n ← temp + 1		q2: n ← temp + 1

Alcuni scenari di questo algoritmo restituiranno per la variabile n il valore atteso (2):

Process p	Process q	n	p.temp	q.temp
<b>p1: temp ← n</b>	q1: temp ← n	0	?	?
<b>p2: n ← temp + 1</b>	q1: temp ← n	0	0	?
(end)	<b>q1: temp ← n</b>	1	0	?
(end)	<b>q2: n ← temp + 1</b>	1	0	1
(end)	(end)	2	0	1

Altri scenari di questo algoritmo non restituiranno per la variabile n il valore atteso (2):

Process p	Process q	n	p.temp	q.temp
<b>p1: temp ← n</b>	q1: temp ← n	0	?	?
p2: n ← temp + 1	<b>q1: temp ← n</b>	0	0	?
<b>p2: n ← temp + 1</b>	q2: n ← temp + 1	0	0	0
(end)	<b>q2: n ← temp + 1</b>	1	0	0
(end)	(end)	1	0	0

In questo secondo algoritmo l'ordine di esecuzione influenza il risultato. Per cui avere operazioni atomiche non è sufficiente a garantire consistenza a un programma concorrente.

**Note:-**

Il comportamento osservato nel secondo algoritmo prende il nome di *race condition* (condizione di corsa).

### 1.2.2 Sistemi Monoprocesso e Sistemi Multiprocesso

#### Sistemi monoprocesso (parallelismo apparente):

- i processi sono *alternati nel tempo* per *simulare* un multiprocesso;
- in ogni istante un solo processo è in esecuzione;
- c'è interleaving nell'esecuzione delle singole istruzioni;
- il meccanismo degli *interrupt* su cui è basato l'avvicendamento dei processi garantisce che l'interrupt venga servito primo o dopo l'esecuzione di un'istruzione, mai durante;
- ogni istruzione macchina è *atomica*.

#### Sistemi multiprocesso con memoria comune (parallelismo reale):

- più processi vengono eseguiti *simultaneamente* su processori diversi;
- c'è *sovraposizione* (overlapping) nell'esecuzione delle istruzioni;
- i processi sono *sequenzializzati nell'accesso alla memoria*;
- le operazioni elementari (*microistruzioni*) che implementano istruzioni macchina sono realizzate da CPU diverse, ma le operazioni elementari che consistono in accessi alla memoria devono essere eseguite una alla volta;
- in questo caso è l'*arbitro del bus* che si preoccupa della sequenzializzazione e che garantisce l'atomicità delle operazioni di lettura/scrittura;
- l'accesso fisico al bus può rendere atomiche le istruzioni macchina.

**Note:-**

In entrambi i sistemi non è possibile prevedere l'alternanza nell'esecuzione di due processi.

#### Esempio 1.2.3 (Programma concorrente)

<b>Main</b>	
int n ← 0	
P	Q
p1: load R1,n p2: add R1,1 p3: store R1,n	q1: load R1,n q2: add R1,1 q3: store R1,n
print n	

**Monoprocessore**

Il sistema operativo effettua i context-switch per alternare l'esecuzione dei due processi sulla stessa CPU.

<b>p1</b>	<b>load R1,n</b>	
<b>SO</b>	<b>interrupt</b>	<b>R1=0, n=0</b>
<b>SO</b>	<b>salvataggio P</b>	<b>n=0</b>
<b>SO</b>	<b>ripristino Q</b>	<b>n=0</b>
<b>q1</b>	<b>load R1,n</b>	<b>R1=? n=0</b>
<b>q2</b>	<b>add R1,1</b>	<b>R1=0, n=0</b>
<b>q3</b>	<b>store R1,n</b>	<b>R1=1, n=0</b>
<b>SO</b>	<b>interrupt</b>	<b>R1=1, n=1</b>
<b>SO</b>	<b>salvataggio Q</b>	<b>n=1</b>
<b>SO</b>	<b>ripristino P</b>	<b>n=1</b>
<b>p2</b>	<b>add R1,1</b>	<b>R1=0, n=1</b>
<b>p3</b>	<b>store R1,n</b>	<b>R1=1, n=1</b>
		<b>R1=1, n=1</b>

**Multiprocessore**

I due processi sono eseguiti da due diversi processori (che hanno ovviamente insiemi di registri distinti)

P	Q	n
<b>p1:load R<sub>p</sub>,n</b>		<b>n=0</b>
	<b>q1:load R<sub>q</sub>,n</b>	<b>n=0</b>
<b>P2:add R<sub>p</sub>,1</b>	<b>q2:add R<sub>q</sub>,1</b>	<b>n=0</b>
	<b>q3:store R<sub>q</sub>,n</b>	<b>n=0</b>
<b>p3:store R<sub>p</sub>,n</b>		<b>n=1</b>
		<b>n=1</b>

**Domanda 1.2**

Le istruzioni macchina sono atomiche?

**Esempio 1.2.4 (exc)**

Consideriamo una istruzione macchina del tipo: **exc a,b** che scambi i contenuti delle celle di memoria **a** e **b**, e che sia implementata in questo modo (quindi **non atomico**):

```
exc a,b
rd R1, a
rd R2, b
wr R2, a
wr R1, b
```

Supponiamo che in una macchina multiprocessore ci siano due processi **P** e **Q** che eseguono entrambi l'unica istruzione macchina **exc a,b**. Supponiamo che inizialmente le celle di memoria **a** e **b** contengano rispettivamente i valori 0 e 1.

<b>Main</b>	
<b>a = 0; b = 1</b>	
<b>P</b>	<b>Q</b>
<b>p1: exc a,b</b>	<b>q1: exc a,b</b>

Consideriamo una possibile esecuzione delle due istruzioni parallele tenendo presente che gli accessi alla memoria centrale devono essere sequenzializzati

P	Q	R <sup>P1</sup>	R <sup>P2</sup>	R <sup>Q1</sup>	R <sup>Q2</sup>	a	b
rd R1,a		?	?	?	?	0	1
rd R2,b		0	?	?	?	0	1
wr R2,a		0	1	?	?	0	1
	rd R1,a	0	1	?	?	1	1
	rd R2,b	0	1	1	?	1	1
wr R1,b		0	1	1	1	1	1
	wr R2,a	0	1	1	1	1	0
	wr R1,b	0	1	1	1	1	0
		0	1	1	1	1	1

Il valore finale assunto dalle variabili **a** e **b** risulta <1 1>. Ma non esiste nessuna esecuzione sequenziale delle due istruzioni **exc a,b** che dia questo risultato!!

#### Definizione 1.2.4: Stato di un programma concorrente

Lo stato di un programma concorrente è una tupla costituita dai valori dei *control pointer*<sup>a</sup> dei vari processi e dai valori delle variabili locali e globali.

<sup>a</sup>Indica lo statement del processo che sta per essere eseguito.

#### Corollario 1.2.1 Transizione

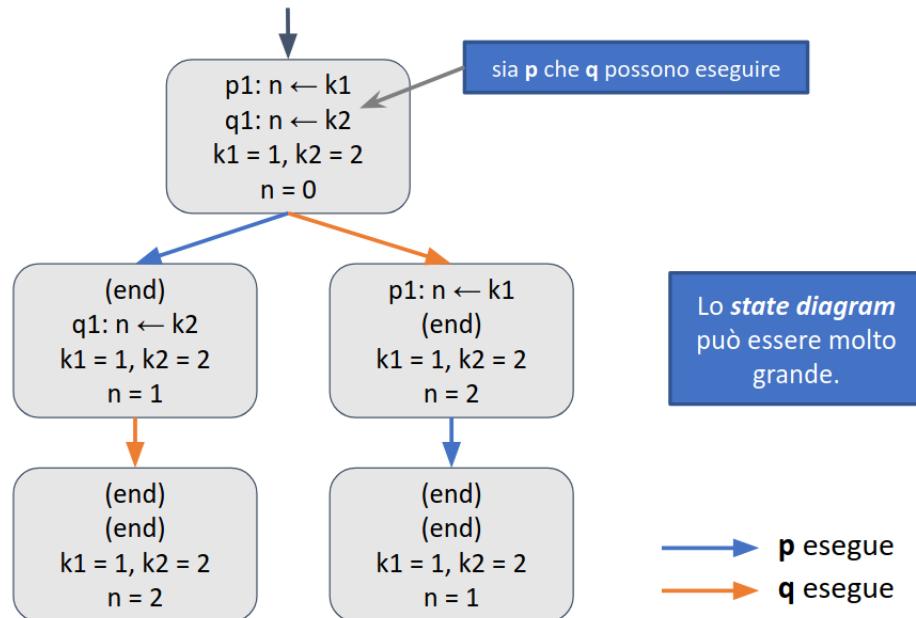
Siano  $s_1$  e  $s_2$  due stati di un programma concorrente. Esiste una *transizione* da  $s_1$  a  $s_2$ , indicata con  $s_1 \rightarrow s_2$ , se l'esecuzione di una istruzione nello stato  $s_1$  porta nello stato  $s_2$ .

#### Esempio 1.2.5 (Programma concorrente banale)

Algoritmo: Programma Concorrente Banale	
integer n ← 0	
<b>p</b>	<b>q</b>
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

- Lo stato deve includere i valori dei control pointers (p1 e q1) dei 2 processi, il valore della variabile globale n e i valori delle variabili locali k1 e k2;
- Lo stato iniziale può transire da 2 diversi stati a seconda di quale processo esegua per primo l'assegnamento;
- Si può illustrare il comportamento di questo programma concorrente con un *diagramma degli stati*.

Diagramma degli stati di un programma concorrente:



## 1.3 Correttezza di Programmi Concorrenti

### 1.3.1 Introduzione e Proprietà

La definizione di *correttezza totale* di un programma sequenziale richiede che il programma *termini* e che, per ogni input, il *risultato* restituito dal programma sia il valore della funzione che il programma deve calcolare. Purtroppo questa definizione non è adeguata al caso dei programmi concorrenti:

- può essere desiderabile che un programma concorrente non termini (ad esempio i processi server degli OS);
- inoltre un programma concorrente non può più essere considerato una funzione perché si aggiungono richieste come *mutua esclusione*, l'assenza di *deadlock*, l'assenza di *starvation*.

**Note:-**

La correttezza di un programma concorrente viene definita in termini di *validità di proprietà*.

Lampton ha definito due categorie di proprietà di correttezza:

- *Safety*: la proprietà P deve sempre essere vera (P è vera in ogni stato della computazione);
- *Liveness*: la proprietà P prima o poi sarà vera (in ogni computazione esiste uno stato in cui P è vera).

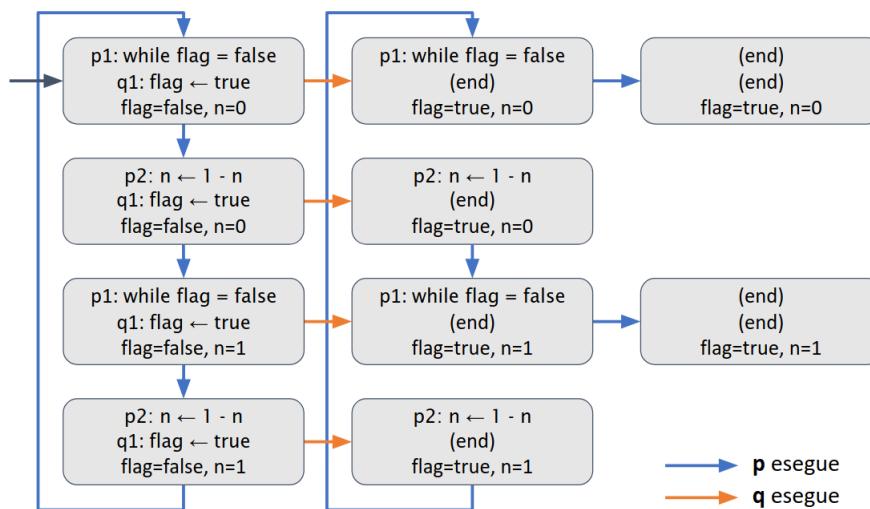
**Domanda 1.3**

Questo programma termina per tutte le computazioni?

Algoritmo StopTheLoopA	
boolean flag $\leftarrow$ false integer n $\leftarrow$ 0	
p	q
p1: while flag = false: p2:       n $\leftarrow$ 1 - n	q1: flag $\leftarrow$ true q2:

**Note:-**

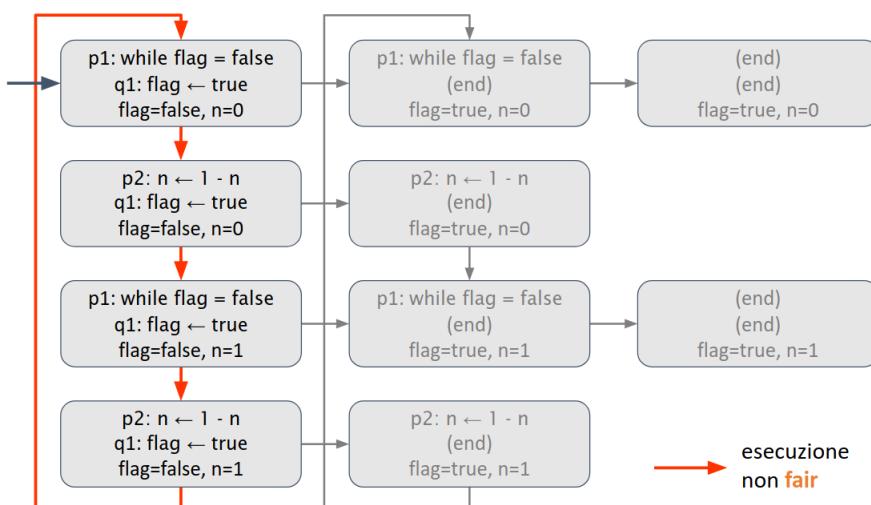
Per rispondere dobbiamo vedere il diagramma degli stati corrispondente.



**Risposta:** questo programma termina solo in due stati, quindi la risposta è no.

**Note:-**

Questo comportamento non è *fair*.



**Definizione 1.3.1: Fairness**

Una computazione è *fair* se per ogni istruzione "costantemente" abilitata esiste uno stato in cui essa viene eseguita, prima o poi.

**Note:-**

Imporre le *ipotesi di fairness* a un programma concorrente significa escludere dalle sue computazioni quelle che non soddisfano la proprietà di fairness.

**1.3.2 La Correttezza di Programmi**

La correttezza di un *programma sequenziale* richiede:

1. Terminazione;
2. Correttezza del risultato.

**Definizione 1.3.2: Program testing**

Per controllare la correttezza di un programma sequenziale si può:

- eseguire il programma n volte con dei *punti di break* in modo da individuare eventuali malfunzionamenti (debugging) o testare con input predefiniti (unit test);
- ricorrere a metodi di verifica (*program verification*) basati su sistemi formali, per esempio la logica dei predicati.

**Note:-**

La correttezza di un programma concorrente richiede la verifica di opportune proprietà per ogni possibile sua computazione, per cui le tecniche di debugging sono inefficienti.

Per provare la correttezza di un programma concorrente si possono usare:

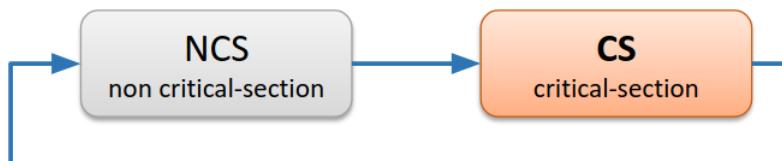
- *Diagramma degli stati*: però spesso si hanno troppi stati ed è difficile verificare le proprietà di Liveness;
- *Metodi formali di verifica*: prove di proprietà scritte in un adeguato linguaggio logico, basate su invarianti;
- *Model checker*: un programma che costituisce il diagramma degli stati di un programma concorrente e simultaneamente verifica le proprietà.

**1.3.3 Il Problema della Mutua Esclusione****Definizione 1.3.3: Problema della mutua esclusione**

Se un processo esegue la propria sezione critica, nessun altro processo esegue la propria.

**Note:-**

N processi concorrenti eseguono in un loop infinito una sequenza di statement che può essere divisa in due sotto-sequenze: non-sezione critica (NCS) e sezione critica (CS).



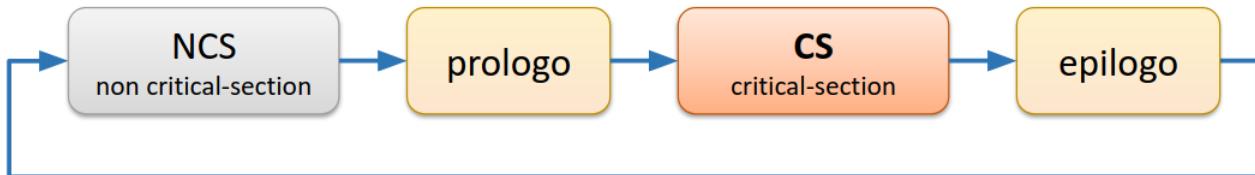
#### Definizione 1.3.4: Sincronizzazione

Per ottenere la mutua esclusione si deve introdurre un *meccanismo di sincronizzazione*, costituito da statement aggiuntivi, alcuni posti prima dell'accesso alla sezione critica: *prologo* (preprotocol) e alcuni all'uscita della sezione critica : *epilogo* (postprotocol).

Algoritmo Critical Section Problem	
global variables	
<b>p</b>	<b>q</b>
local variables loop forever: non-critical section preprotocol critical section postprotocol	local variables loop forever: non-critical section preprotocol critical section postprotocol

Per risolvere il problema della mutua esclusione si parte da alcune assunzioni:

1. quando un processo è in CS progredisce nell'esecuzione del suo codice e lo porta a termine (non ci sono loop infiniti, statement di terminazione o malfunzionamenti) (*proprietà di progresso all'interno della sezione critica*);
2. le computazioni sono fair;
3. le variabili usate nel prologo e nell'epilogo non sono utilizzate in CS o in NCS;
4. i processi utilizzano sempre correttamente il protocollo.



#### Note:-

- Se un utente si blocca in CS l'intero sistema si blocca o va in controllo a malfunzionamenti;
- Le assunzioni 1 e 2 garantiscono che nelle CS non si presentino malfunzionamenti, cicli infiniti, statement di terminazione, etc.);
- Non si fanno restrizioni sul codice in NCS;
- La quarta assunzione garantisce che i protocolli proposti siano sempre eseguiti;
- La terza assunzione garantisce che le variabili del protocollo siano disgiunte rispetto a quelle dei singoli processi.

### 1.3.4 Specifiche di Correttezza

Approccio di Dijkstra (storico):

- *Mutua esclusione*: solo un processo alla volta deve essere all'interno della CS;
- *Assenza di deadlock*: non può accadere che tutti i processi siano bloccati definitivamente durante l'esecuzione del prologo;
- *Assenza di delay non necessari*: un processo fuori dalla CS non può ritardare l'accesso alla CS da parte di un altro processo;
- *Assenza di starvation*: ogni processo che richiede l'accesso alla CS prima o poi l'ottiene.

**Note:-**

Le prime tre proprietà sono fondamentali, la quarta in alcuni casi può essere omessa.

Approccio di Silberschatz-Galvin (concetto dei sistemi operativi):

- *Mutua esclusione*;
- *Progresso*: se nessun processo sta eseguendo in CS e alcuni processi richiedono l'accesso alla propria CS, allora i processi che sono in NCS non possono partecipare alla decisione riguardante la scelta di chi può entrare per primo in CS (condensa i punti due e tre di Dijkstra);
- *Attesa limitata*: c'è un limite sul numero di accessi alla CS accordati a un processo, quando un altro processo ha fatto richiesta di accesso alla propria CS, ed è ancora in attesa di entrare (modo alternativo di definire l'assenza di starvation).

Approccio di Ben-Ari (principi di programmazione concorrente e distribuita):

- *Mutua esclusione*: non ci può essere interleaving tra gli statement della CS dei processi;
- *Assenza di deadlock*:
- *Assenza di starvation*;

Approccio di Lynch (algoritmi distribuiti):

- *Mutua esclusione*: non esiste uno stato del programma concorrente in cui più di un processo è in CS;
- *Progresso*: in ogni computazione fair deve essere verificato che:
  - se uno o più processi stanno eseguendo il prologo e nessuno è in CS, prima o poi uno di questi processi accede alla CS (progresso per l'accesso);
  - se un processo sta eseguendo l'epilogo prima o poi accede alla NCS (progresso per l'uscita);
- *Assenza di starvation* (lockout freedom): in ogni computazione fair ogni processo che sta eseguendo il prologo (epilogo) prima o poi accede alla CS (esce dalla CS).

**Esempio 1.3.1 (Mutua esclusione di due processi)**

Si vuole dimostrare la correttezza della seguente soluzione mediante il diagramma degli stati. Ogni stato è definito da terne del tipo ( $\pi_i, q_j, \text{turn}$ ).

Algoritmo 1	
integer turn ← 1	
p	q
loop forever: p1: non-critical section p2: <b>await</b> turn = 1 p3: critical section p4: turn ← 2	loop forever: q1: non-critical section q2: <b>await</b> turn = 2 q3: critical section q4: turn ← 1

★ Il costrutto: **await** condizione: istruzioni  
è definito come: while not condizione: <esegui istruzioni>

Per determinarne la correttezza si costruisce il diagramma degli stati e si sceglie di seguire l'approccio di Ben-Ari:

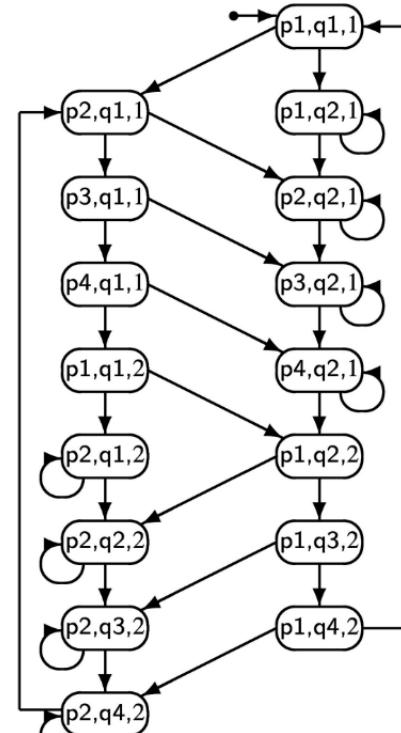
- Mutua esclusione;
- Assenza di deadlock;
- Assenza di starvation.

Costruiamo quindi il Diagramma degli stati.

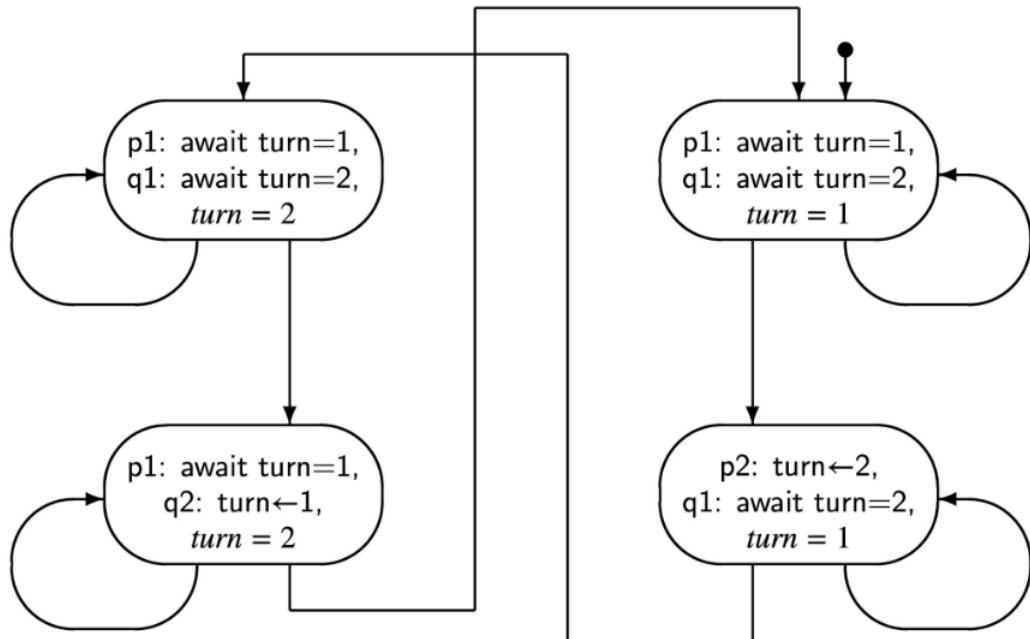
p
loop forever:
p1: non-critical section
p2: <b>await</b> turn = 1
p3: critical section
p4: turn ← 2

q
loop forever:
q1: non-critical section
q2: <b>await</b> turn = 2
q3: critical section
q4: turn ← 1



**Mutua esclusione:** è sufficiente verificare che nel diagramma non esistono stati in cui p e q siano entrambi in sezione critica, ossia stati del tipo (p3, q3, \*).



**No deadlock:** si utilizza una versione semplificata del diagramma degli stati non considerando la CS. Si vede che ogni stato ha sempre un'uscita.

**No starvation:** questo non è vero perché non si ha un progresso garantito al di fuori della CS.

### Esempio 1.3.2 (Mutua esclusione)

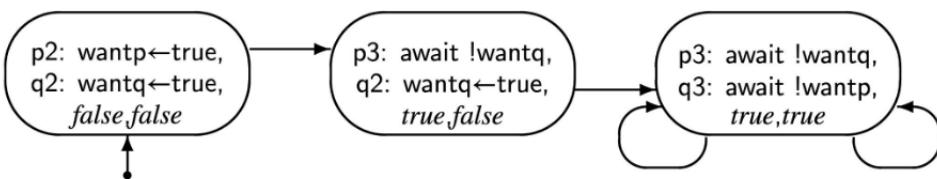
Consideriamo un altro algoritmo che non utilizza turn, ma wantp e wantq per indicare se un processo si trova in sezione critica.

<b>Algoritmo 2</b>	
boolean wantp $\leftarrow$ false, wantq $\leftarrow$ false	
p	q
loop forever: p1: non-critical section p2: wantp $\leftarrow$ true p3: <b>await</b> wantq = false p4: critical section p5: wantp $\leftarrow$ false	loop forever: q1: non-critical section q2: wantq $\leftarrow$ true q3: <b>await</b> wantp = false q4: critical section q5: wantq $\leftarrow$ false

Questo algoritmo garantisce la mutua esclusione però può dare origine a deadlock.

Process p	Process q	wantp	wantq
<b>p1: non-critical section</b>	q1: non-critical section	false	false
p2: wantp $\leftarrow$ true	<b>q1: non-critical section</b>	false	false
<b>p2: wantp<math>\leftarrow</math>true</b>	q2: wantq $\leftarrow$ true	false	false
p3: await wantq=false	<b>q2: wantq<math>\leftarrow</math>true</b>	true	false
p3: await wantq=false	q3: await wantp=false	true	true

Se entrambi i processi impostano wantp e wantq a true, poi si aspettano a vicenda, e nessuno dei due entra in CS.



**Note:-**

entrambi questi algoritmi sono scorretti. Si deve usare l'algoritmo di Dekker (proposto da Dijkstra nel 1965) che usa sia wantp/wantq che turn per stabilire quale dei due processi abbia il diritto di insistere nel tentativo di accesso.

### Esempio 1.3.3 (Algoritmo di Dekker)

<b>Algoritmo di Dekker</b>	
<pre> boolean wantp <math>\leftarrow</math> false, wantq <math>\leftarrow</math> false integer turn <math>\leftarrow</math> 1   </pre>	
<b>p</b>	<b>q</b>
loop forever: p1: non-critical section p2: wantp $\leftarrow$ true p3: <b>while</b> wantq p4: <b>if</b> turn = 2 p5:         wantp $\leftarrow$ false p6: <b>await</b> turn = 1 p7:         wantp $\leftarrow$ true p8:     critical section p9:     turn $\leftarrow$ 2 p10:    wantp $\leftarrow$ false	loop forever: q1: non-critical section q2: wantq $\leftarrow$ true q3: <b>while</b> wantp q4: <b>if</b> turn = 1 q5:         wantq $\leftarrow$ false q6: <b>await</b> turn = 2 q7:         wantq $\leftarrow$ true q8:     critical section q9:     turn $\leftarrow$ 1 q10:    wantq $\leftarrow$ false

La sua correttezza può essere provata dal diagramma degli stati.

Per provarla in maniera informale si può ragionare per assurdo sulla mutua esclusione: supponiamo che entrambi i processi siano in CS. Uno dei due sarà entrato per primo, supponiamo q (quindi wantq = true). Ora se p entra in CS prima che q ne esca dovrebbe esistere un istante in cui wantq = false mentre q è in CS, ma ciò è assurdo.

Per l'assenza di deadlock si suppone che entrambi i processi vogliano entrare nel ciclo while. Le variabili wantp e wantq sono entrambe true e si suppone turn = 2. Il valore di turn sarà modificato solo quando q uscirà dalla CS. Quindi il processo p dovrà eseguire p5 ed entrare nel ciclo await. Non c'è nulla che ostacoli

q a uscire dal while ed entrare in CS.

Per l'assenza di starvation si suppone che il processo p voglia entrare in CS e il processo q sta eseguendo in NCS, p entra. Se invece il processo q esegue in CS o nel prologo/epilogo, per ipotesi prima o poi termina e pone turn = 1 e wantq = false, quindi p può entrare.

**Note:-**

Nel 1981 Peterson propose una variante dell'algoritmo di Dekker più semplice e più generale. Invece di usare turn utilizza una variabile last per indicare chi è stato l'ultimo a provare ad accedere: questo determina la priorità di accesso.

**Esempio 1.3.4 (Algoritmo di Peterson)**

<b>Algoritmo di Peterson</b>	
<b>p</b>	<b>q</b>
loop forever:	loop forever:
p1: non-critical section	q1: non-critical section
p2: wantp $\leftarrow$ true	q2: wantq $\leftarrow$ true
p3: last $\leftarrow$ 1	q3: last $\leftarrow$ 2
p4: <b>await</b> wantq=false <b>or</b> last=2	q4: <b>await</b> wantp=false <b>or</b> last=1
p5: critical section	q5: critical section
p6: wantp $\leftarrow$ false	wantq $\leftarrow$ false

**Diagramma stati  
algoritmo di Peterson  
(abbreviato)**

**M.E.:**

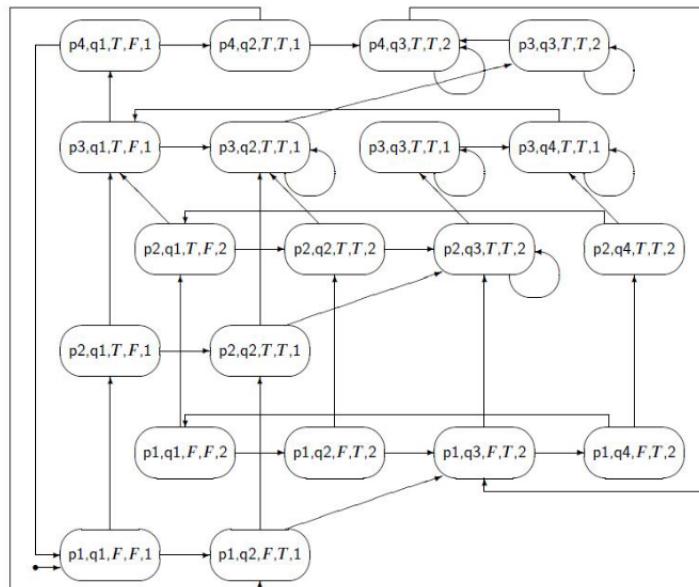
non esistono stati della forma  
(p4,q4...)

**No deadlock:**

è possibile uscire dallo stato  
(p3,q3,...)

**No starvation:**

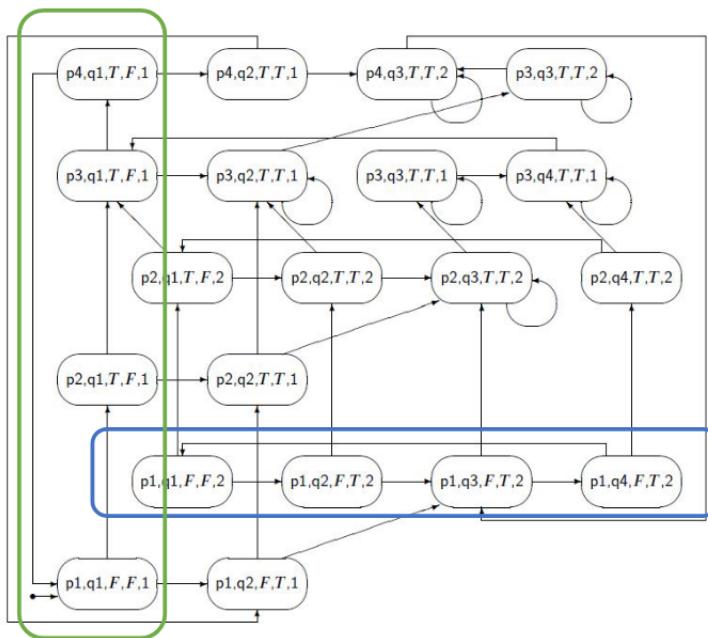
non esistono loop che  
coinvolgono stati che non  
contengano p4 (o q4)



**Diagramma stati  
algoritmo di Peterson  
(abbreviato)**

**NOTA:**

Esistono loop dove **p1** o **q1** non avanzano. Questi però sono esclusi, in quanto sono le computazioni **unfair**.



**Note:-**

Nelle moderne CPU multi-core gli algoritmi di Dekker e Peterson non funzionano più (almeno non nella forma vista precedentemente). Perché:

- Coerenza della memoria per codice sequenziale per core;
- Utilizzo della cache intermedia;
- Servono le memory fences (per sincronizzare le cache intermedie).

### 1.3.5 Statement Atomici Particolari

Fino a ora abbiamo supposto l'utilizzo di istruzioni atomiche di assegnamento (write e read). Esistono istruzioni atomiche particolari che permettono di realizzare facili ed efficienti meccanismi di sincronizzazione:

- test-and-set(common, local);
- exchange(a,b);
- fetch-and-add(common, local, x).

**Note:-**

Questi statement sono tali che durante la loro esecuzione accedono direttamente in memoria centrale (no cache) e non rilasciano il controllo del bus. Così evitano interleaving di operazioni di write e read da parte di altri processi.

**Esempio 1.3.5** (test-and-set)

**atomic instruction** test-and-set (**common**, **local**):  
**local**  $\leftarrow$  **common**  
**common**  $\leftarrow$  1

Algoritmo: Sezione critica con test-and-set	
integer common $\leftarrow$ 0	
<b>p</b>	<b>q</b>
integer local1 loop forever: p1: non-critical section p2: <b>repeat</b> test-and-set(common, local1) p3: <b>until</b> local1 = 0 p4: critical section p5: common $\leftarrow$ 0	integer local2 loop forever: q1: non-critical section q2: <b>repeat</b> test-and-set(common, local2) q3: <b>until</b> local2 = 0 q4: critical section q5: common $\leftarrow$ 0

**1.3.6 Invarianti e Predicati**

Per provare la correttezza di un programma concorrente si può usare l'induzione su *proprietà invarianti*.

**Definizione 1.3.5: Invariante**

Un *invariante* è una formula A in un qualche sistema formale che ha la proprietà di *essere sempre vera in ogni punto di qualsiasi computazione*.

**Corollario 1.3.1** Predicato

Un *predicato*, chiamato anche *proposizione atomica*, è un'espressione booleana che può essere valutata avendo a disposizione la tupla delle variabili di stato dei processi, e i loro control pointers.

**Note:-**

Le formule A che verranno utilizzate sono anche proposizioni atomiche.

**Osservazioni 1.3.1** Predicati e invarianti**Domanda 1.4**

Come si dimostra che una proprietà A è un'invariante?

**Risposta:** la prova che A è un'invariante si fa *per induzione* sugli stati di tutte le computazioni. Si prova che A vale nello stato iniziale (passo base) e si suppone che A valga in uno stato s e in tutti gli stati precedenti a s (ipotesi induttiva) e si dimostra che vale anche in ciascuno dei possibili stati successivi (passo induttivo).

**Note:-**

In uno programma concorrente gli stati successori possono essere più di uno (per via dell'interleaving) quindi il passo induttivo deve essere verificato per tutti i possibili successori.

### Esempio 1.3.6 (Invarianti)

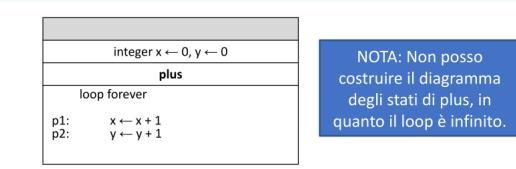
Consideriamo il programma costituito da un unico processo plus; siano:

- **val(x)** il valore della variabile x in uno stato s della computazione;
- **val(y)** il valore della variabile y in uno stato s della computazione;

Si vuole provare che la formula:

$$0 \leq val(x) - val(y) \leq 1$$

è un *invariante* del programma.



NOTA: Non posso costruire il diagramma degli stati di plus, in quanto il loop è infinito.

Si introducono le notazioni:

- $Np_i$  ( $i:1..2$ ): variabili intere che in ogni stato indicano il numero N di esecuzioni di  $p_i$  fino a quel punto completate;
- $p_i$  ( $i:1..2$ ): proposizione atomica vera se il control pointer del processo plus vale  $p_i$ .

**Lemma 1.** La seguente formula **A** è un invariante:

$$A: (p1 \rightarrow (Np1 = Np2)) \wedge (p2 \rightarrow (Np1 = Np2 + 1))$$

La dimostrazione è per induzione sugli stati. L'assunto è banalmente vero nello stato iniziale (passo base). Supposto che **A** sia vera in tutti gli stati precedenti lo stato **s** dimostriamo che è vera anche per lo stato **s**. Se nello stato che prendiamo in esame il control pointer vale  $p_2$ , nello stato precedente il control pointer valeva  $p_1$  e l'ipotesi induttiva garantisce che  $Np1 = Np2$ , ora l'esecuzione di  $p_1$  ha comportato l'incremento di  $Np1$  e quindi possiamo affermare che  $p2 \rightarrow (Np1 = Np2 + 1)$

Se il control pointer vale  $p_1$  la prova è analoga.

Dal Lemma 1 si può dedurre che in ogni stato vale l'Invariante

$$0 \leq Np1 - Np2 \leq 1$$

Un'invariante che si basa sull'ordine delle istruzioni di un processo viene indicato come *invariante topologico*.

**Lemma 2.** La seguente formula **B** è un invariante:

$$B: (val(x) = Np1) \wedge (val(y) = Np2)$$

La dimostrazione è per induzione sugli stati. L'assunto è banalmente vero nello stato iniziale (passo base).

Supponendo **B** vera in tutti gli stati precedenti lo stato **s**, dimostriamo che è vera anche in **s**. Supponiamo che in **s** il control pointer valga  $p_2$ : quindi nello stato precedente il control pointer valeva  $p_1$  e l'ipotesi induttiva garantiva che  $val(x) = Np1$ , per transire ad **s** è stata eseguita  $p_1$  che ha comportato l'incremento di  $Np1$  ma anche l'incremento di  $x$  quindi la prima parte della formula è nuovamente vera, la seconda parte della formula non è stata modificata.

Se il control pointer vale  $p_1$  la prova è analoga.

**Teorema.** La formula è un invariante:

$$0 \leq val(x) - val(y) \leq 1$$

La prova segue dai lemmi 1 e 2.



2

Test2

