Esercitazione 11
Gruppo LZ

Monitor Avanzato

Agenda

Esempio – Gestione di un ponte

Esercizio 1 – isola con ponte pedonale

Esercizio 2 – isola con ponte pedonale e gruppi di numerosità variabile.

Esempio

Gestione di un ponte

Esempio

Si consideri un piccolo **ponte** che collega le due rive (Nord e Sud) di un fiume.

Al ponte possono accedere due tipi di veicoli: **auto** e **moto**.

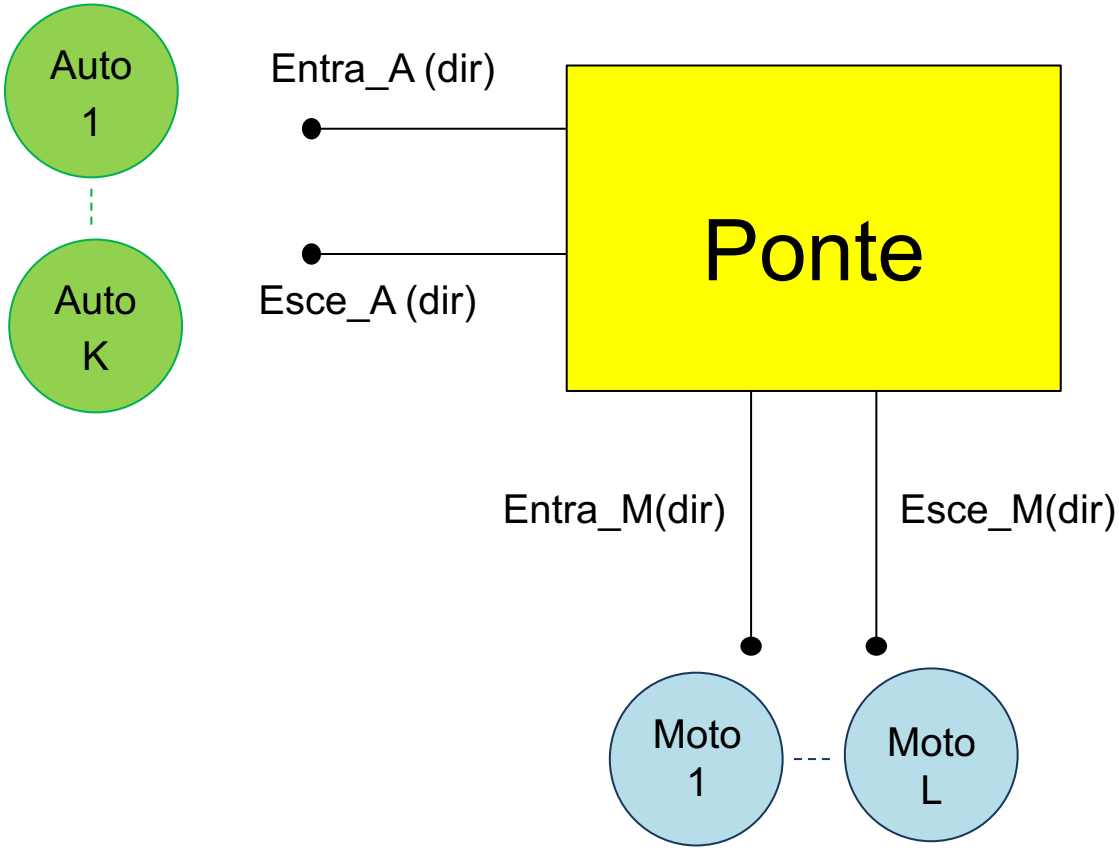
Il ponte ha una capacità massima **MAX** che esprime il **numero massimo di motoveicoli** che possono transitare contemporaneamente su di esso; a questo proposito si assuma che un'automobile valga come 4 motoveicoli.

Il ponte è talmente stretto che il transito di un'auto in una particolare direzione **d** impedisce l'accesso al ponte di qualunque altro veicolo (auto o moto) in direzione opposta a d.

Realizzare una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che, nell'accesso al ponte, dia la **precedenza alle moto**, rispetto alle auto.

Impostazione

- Quali sono i thread? → **auto e moto**
- Qual è la risorsa condivisa? → **il ponte**



Sincronizzazione

Possibili sospensioni di auto e moto nell'accesso al ponte:

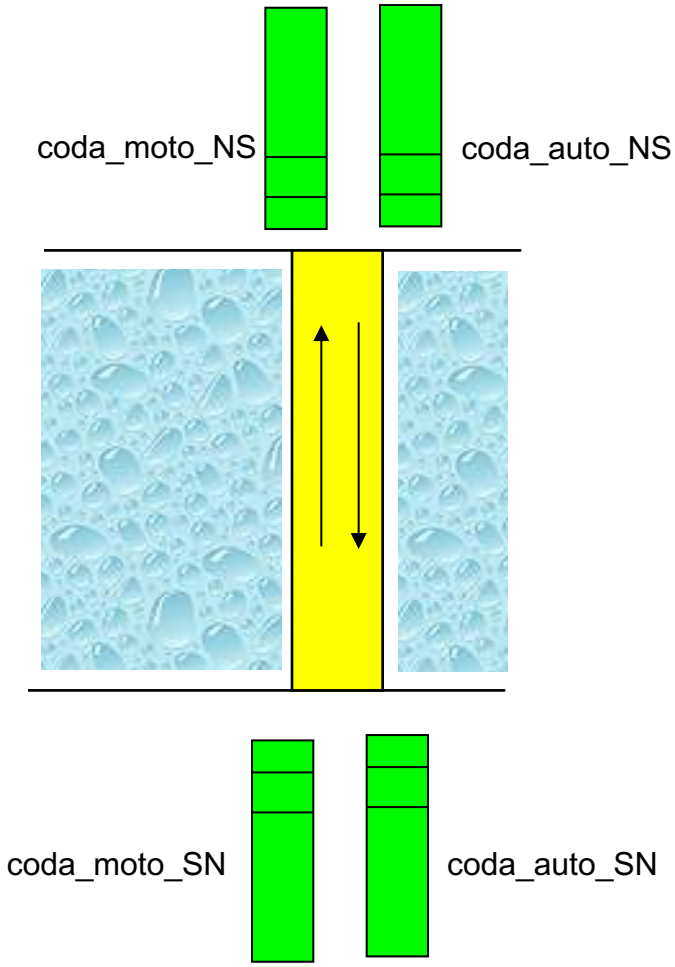
- Una **Moto** si sospende in ingresso:
  - se **il ponte è pieno**
  - se **c'è almeno un'auto** sul ponte **in direzione opposta**
- Un'**Auto** si sospende in ingresso:
  - se il **ponte è pieno**
  - se **c'è almeno un'auto o una moto** sul ponte in **direzione opposta**
  - se **c'è almeno una moto in attesa** (in qualunque direzione)

Sincronizzazione

Quante/quali condition?

La sospensione di auto e moto dipende anche dalla direzione di accesso:

- 2 code di accesso sulla riva Nord (veicoli N->S)
  - **coda_auto_NS**
  - **coda_moto_NS**
- 2 code di accesso sulla riva Sud (veicoli S->N)
  - **coda_auto_SN**
  - **coda_moto_SN**



Soluzione: thread Auto

```
public class Auto extends Thread {
        private Monitor M;
        private int dir;
        private Random r;

        public Auto(Monitor M, int D, Random R, int i) {
                this.M = M;
                this.dir=D;
                this.r=R;
                        }

        public void run() {
                try {   sleep(r.nextInt(10*1000));
                        M.entraAuto(dir);
                        sleep(r.nextInt(10*1000));
                        M.esceAuto(dir);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Soluzione: thread Moto

```
public class Moto extends Thread {
        private Monitor M;
        private int dir;
        private Random r;

        public Moto(Monitor M, int D, Random R, int i) {
                this.M = M;
                this.dir=D;
                this.r=R;
                        }

        public void run() {
                try {   sleep(r.nextInt(10*1000));
                        M.entraMoto(dir);
                        sleep(r.nextInt(10*1000));
                        M.esceMoto(dir);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Soluzione: Monitor

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor {
        //costanti di direzione:
        private final int NS=0;
        private final int SN=1;
        private int MAX;
        private Lock lock = new ReentrantLock();
        private int []autoIN=new int[2]; // auto sul ponte, per ogni dir
        private int []motoIN=new int[2]; // moto sul ponte, per ogni dir
        private int totIN; //numero totale di moto equivalenti sul ponte
        // Condition  e contatori dei sospesi:
        private Condition [] codaAuto = new Condition[2]; //1 coda x dir
        private Condition [] codaMoto = new Condition[2]; //1 coda x dir
        private int []sospAuto=new int[2];
        private int []sospMoto=new int[2];
```

Soluzione: Monitor

```
//Costruttore:
public Monitor(int N) {
        this.MAX= N;
        this.totIN=0;
        for (int i=0; i<2;i++)
        {       autoIN[i]=0;
                motoIN[i]=0;
                codaAuto[i]=lock.newCondition();
                codaMoto[i]=lock.newCondition();
                sospAuto[i]=0;
                sospMoto[i]=0;
        }

}
private int altradir(int d) // ritorna la direz. opposta a d
{       if (d==NS)
                return SN;
        else
                return NS;
}
```

Soluzione: Monitor

```
public void entraMoto(int d) throws InterruptedException
{
      lock.lock();
      while (totIN==MAX || (autoIN[altradir(d)]) >0 ) {
            sospMoto[d]++;
            codaMoto[d].await();
            sospMoto[d]--;
      }
      motoIN[d]++;
      totIN++;
      lock.unlock();
}
```

Soluzione: Monitor

```
public void entraAuto(int d) throws InterruptedException
{     lock.lock();
      while ((totIN+4)>MAX || autoIN[altradir(d)] >0 ||
      motoIN[altradir(d)]>0 || sospMoto[NS]>0 ||
      sospMoto[SN]>0) {
            sospAuto[d]++;
            codaAuto[d].await();
            sospAuto[d]--;
      }
      autoIN[d]++;
      totIN+=4;
      lock.unlock();
}
```

Soluzione: Monitor

```
public void esceMoto(int d) {
      lock.lock();
      motoIN[d]--;
      totIN--;
      if (sospMoto[altradir(d)]>0 && autoIN[d]==0)
                  codaMoto[altradir(d)].signal();
      else if (sospMoto[d]>0)
                  codaMoto[d].signal();
      else if (sospAuto[altradir(d)]>0 && motoIN[d]==0
                  && autoIN[d]==0)
      //possibile inversione di direzione:
                  codaAuto[altradir(d)].signalAll();
      else if (sospAuto[d]>0 &&  totIN+4<=MAX &&
                  sospMoto[altradir(d)]==0)
                  codaAuto[d].signal();
```

Soluzione: Monitor

```
public void esceAuto(int d) {
      lock.lock();
      autoIN[d]--;
      totIN-=4;
      if (sospMoto[altradir(d)]>0 && autoIN[d]==0)
            codaMoto[altradir(d)].signalAll();
      if (sospMoto[d]>0)
            codaMoto[d].signalAll(); //1 auto vale 4 moto
      if (sospAuto[altradir(d)]>0 && motoIN[d]==0
            && autoIN[d]==0)
            codaAuto[altradir(d)].signalAll();
      else if (sospAuto[d]>0)
            codaAuto[d].signal();
      lock.unlock();
 }
}
```

Esercizio 1 (1/4)

Si consideri il sito naturalistico di Carrick-a-rede (Ballintoy, Nord Irlanda).
Il sito ospita un lungo e stretto **ponte** pedonale di corda che permetta l'accesso a una piccola **isola** aperta al pubblico per le visite.
L'ingresso all'isola è consentito a due tipi di visitatori:

- **visitatore normale** (una persona)
- **visitatore con zaino da trekking** (equivalente a due persone)

Esercizio 1 (2/4)

Poiché l'isola è impervia e in alcuni tratti i bordi delle scogliere sono sdrucciolevoli, l'isola è presidiata da alcune **guide**.
Una **guida** è una singola persona dello staff con l'incarico di presidiare il sito.
Ogni guida **può ciclicamente entrare**, **presidiare** per un tempo arbitrario e poi **uscire**.

Per tutti, l'**accesso** e l'**uscita** avvengono attraverso il **ponte** che pertanto viene percorso dagli utenti sia in direzione IN (per accedere all'isola), sia in direzione OUT (per uscire dall'isola).

Esercizio 1 (3/4)

Accesso e Uscita dall'isola:
Per evitare situazioni di eccessivo affollamento, **l'isola ha una capacità limitata pari a MAX** persone (visitatori normali, visitatori con zaino e guide) oltre la quale non sarà consentito l'accesso a nessun utente.
Si assuma, inoltre, che **l'uscita di una guida** G dall'isola **non possa avvenire se G è la sola guida** a presidiare il sito in quel momento.

Vincoli sul ponte:
Per motivi di sicurezza il ponte può essere percorso al più da **NP** persone.
Inoltre, poiché è molto stretto, **non è consentito il transito contemporaneo sul ponte di visitatori con zaino in direzioni diverse** (ma è consentito il transito di visitatori normali e visitatori con zaino in direzioni opposte)

Esercizio 1 (4/4)

Realizzare un'applicazione concorrente in Java basata sul monitor nella quale ogni utente (visitatore normale, visitatore+zaino o guida) sia rappresentato da un thread distinto.

La politica di sincronizzazione dovrà tenere in considerazione tutti i vincoli dati, ed inoltre dovrà dare la **precedenza agli utenti in uscita** dall'isola; a **parità di direzione**:

- **in uscita**:
  - i **visitatori con zaino** dovranno avere la precedenza sui visitatori normali.
  - i **visitatori normali** avranno la precedenza sulle guide

- **in entrata**:
  - le **guide** avranno la precedenza sui visitatori con zaino;
  - le **visitatori con zaino** avranno la precedenza sui visitatori normali.

Impostazione

Quali thread?

- thread iniziale
- visitatori normali
- visitatori con zaino (=2 visitatori)
- guide

Quale risorsa comune?

- il sito naturalistico, cioè l'isola e il ponte

Associamo alla risorsa un "monitor", che controlla gli accessi e le uscite in base alla specifica politica di accesso. La sincronizzazione viene realizzata mediante variabili condizione.

Comportamento threads

Comportamento di ogni Visitatore (normale o con zaino):

Ogni visitatore si comporta come segue:

1. **Imbocca il ponte in direzione IN**;
2. Percorre il ponte (direzione IN)
3. **Esce dal ponte in direzione IN** per entrare nell'isola ;
4. Visita l'isola
5. **Imbocca il ponte in direzione OUT** uscendo dall'isola;
6. Percorre il ponte (direzione OUT)
7. **Esce dal ponte in direzione OUT**.

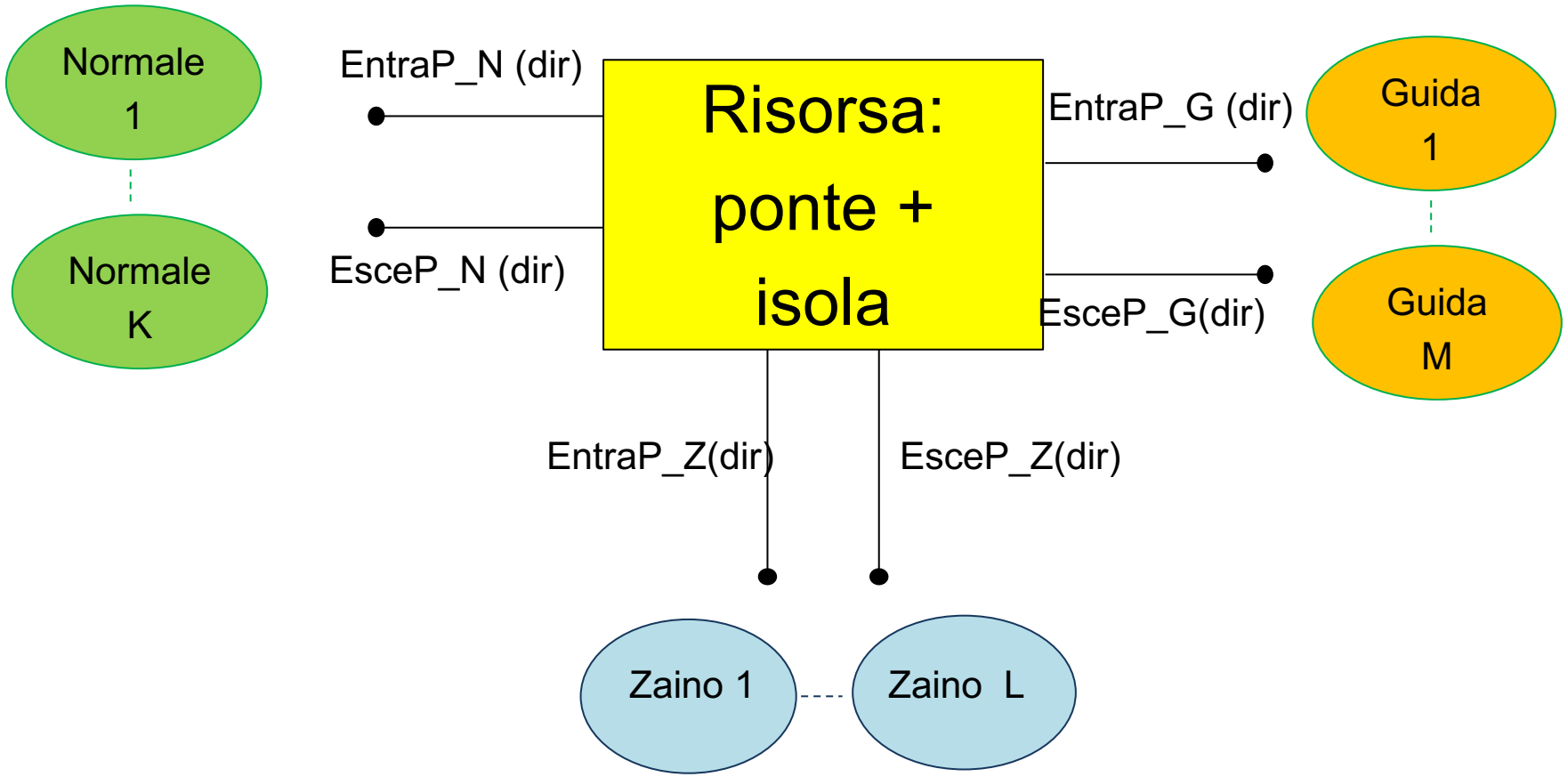
Comportamento di ogni Guida:

ripete ciclicamente le seguenti fasi

1. **Imbocca il ponte in direzione IN**;
2. Percorre il ponte (direzione IN)
3. **Esce dal ponte in direzione IN** per entrare nell'isola;
4. Presidia l'isola per un certo tempo
5. **Imbocca il ponte in direzione OUT** uscendo dall'isola;
6. Percorre il ponte (direzione OUT)
7. **Esce dal ponte in direzione OUT**.

Impostazione

- Quali sono i thread? → **normali, con zaino e guide**
- Qual è la risorsa condivisa? → **ponte+isola**



Struttura dei thread

```
public class Normale extends Thread{
      private Monitor M;

      public Normale(...){ // costruttore..
      }
  public void run(){
      ...
      M.EntraP_N(IN);
      <percorre ponte>
      M.EsceP_N(IN);
      <visita isola>
      M.EntraP_N(OUT);
      <percorre ponte in uscita>
      M.EsceP_N(OUT);
}
```

Struttura dei thread

```
public class Zaino extends Thread{
      private Monitor M;

      public Zaino(...){ // costruttore..
      }
 public void run(){
      ...
      M.EntraP_Z(IN);
      <percorre ponte>
      M.EsceP_Z(IN);
      <visita isola>
      M.EntraP_Z(OUT);
      <percorre ponte in uscita>
      M.EsceP_Z(OUT);
}
```

Struttura dei thread

```
public class Guida extends Thread{
      private Monitor M;
      public Guida(...){ // costruttore..
      }
   public void run(){
      while (..)
      {     M.EntraP_G(IN);
            <percorre ponte>
            M.EsceP_G(IN);
            <presidia isola>
            M.EntraP_G(OUT);
            <percorre ponte in uscita>
            M.Esceé_G(OUT);
      }
}
```

Monitor: sito naturalistico

Variabili di stato:

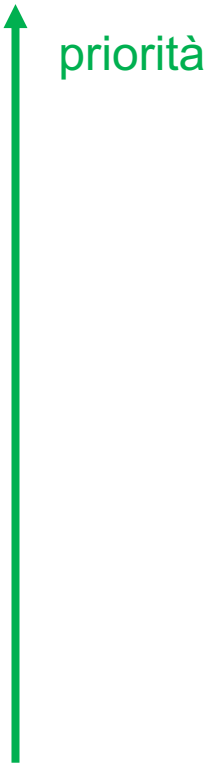
Per il ponte:
quante guide, normali e zaini in ogni direzione ci sono.

Per l'isola:
quanti visitatori nella tomba (zaini valgono 2), quante guide.

Politica di Sincronizzazione: scala delle priorità

Priorità - accesso al ponte:

- in **uscita** dall'isola
  - zaini
  - normali
  - guide
- in **entrata** nell'isola.
  - guide
  - zaini
  - normali



→ **6 livelli di priorità**

Politica di Sincronizzazione

Un thread Normale si sospende entrando nel ponte:

Direzione IN (verso l'isola):

- se il ponte è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se l'isola è piena (**necessario**, altrimenti il ponte si potrebbe riempire di processi in attesa di entrare nell'isola..)
- se non ci sono guide nell'isola

Direzione OUT (uscita dall'isola):

- se il ponte è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità..)

Politica di Sincronizzazione

Un thread Zaino si sospende entrando nel corridoio:

Direzione IN (verso l'isola):

- se non c'è posto per 2 persone nel ponte
- se c'è almeno uno zaino nel ponte in dir OUT
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se nell'isola non c'è posto per 2 persone
- se non ci sono guide nell'isola

Direzione OUT (uscita dall'isola):

- se non c'è posto per 2 persone nel ponte
- se c'è almeno uno zaino nel ponte in dir IN
- se c'è un processo più prioritario in attesa (v. scala priorità..)

Politica di Sincronizzazione

Un thread Guida si sospende **entrando** nel ponte:

Direzione IN (verso l'isola):

- se il ponte è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se l'isola è piena

Direzione OUT (uscita dall'isola):

- se è l'unica guida nell'isola
- se il ponte è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità..)

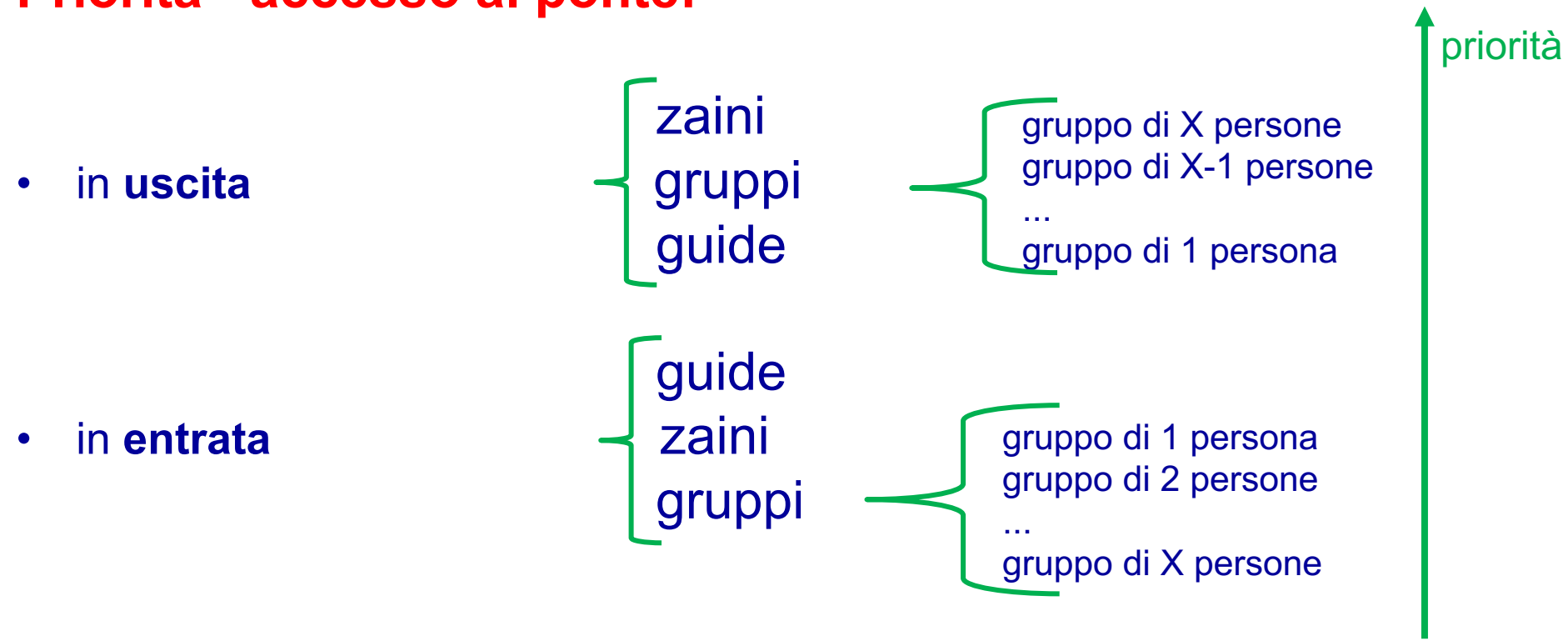
Esercizio 2

Estendere il problema dell'es.1 sostituendo i visitatori nomali con gruppi di consistenza numerica variabile, al massimo di X persone; ad esempio, se X=25, ogni gruppo ha una numerosità arbitraria compresa nell'intervallo [1, 25].
Si progetti una politica di gestione che tenga conto dei vincoli dati e che, inoltre:

- Nell'accesso al ponte si dia sempre la **precedenza agli utenti in uscita dall'isola**.
- Nella **direzione di entrata nell'isola**:
  le guide abbiano la precedenza sugli zaini; gli zaini abbiano la precedenza sui gruppi; tra i gruppi venga data la precedenza ai gruppi meno numerosi.
- Nella **direzione di uscita dall'isola**:
  Gli zaini abbiano la precedenza sui gruppi; tra i gruppi, venga data la precedenza ai gruppi più numerosi. Le guide abbiano la priorità minima.

Politica di Sincronizzazione: scala delle priorità

Priorità - accesso al ponte:



→ 4 + 2*X livelli di priorità (X massima numerosità)

Se X=25 → 2 *25 + 4 = **54 livelli di priorità**