



## Traccia B

### Esercizio 1

Siano date le seguenti dichiarazioni di classe:

```
class Riga{
    friend class Matrice;
public:
    Riga();
    Riga(int size); //è compresa una invocazione di reset()
    Riga(const Riga & riga); //copia profonda
    ~Riga(); //distruttore
    void reset(); //pone a zero tutti gli elementi
    int getSize() const;
    int& operator[](int index); //assert(0 <= index && index < size);
    int operator[](int index) const; //assert(0 <= index && index < size);
    Riga operator+(const Riga & riga) const; //assert(size == riga.size);
    Riga operator-(const Riga & riga) const; //assert(size == riga.size);
    const Riga & operator=(const Riga & riga); //assert(size == riga.size);
    bool operator==(const Riga & riga) const; //assert(size == riga.size);
    bool operator!=(const Riga & riga) const; //assert(size == riga.size);
private:
    int *vett;    int size;
    void setSize(int size); //cambia la dimensione
};

class Matrice {
public:
    Matrice(int rows, int columns);
    ➔ Matrice(const Matrice & mat); //copia profonda
    ➔ ~Matrice();
    ➔ Matrice operator-(const Matrice & matrice) const;
    ➔ Riga& operator[](int index);
    ➔ bool operator!=(const Matrice & matrice) const;
private:
    Riga *mat;    int rows;    int columns;
};
```

Implementare le funzioni della classe **Matrice** indicate dalla freccia. Indicare **ove necessario** eventuali istruzioni di **assert** (prestare quindi attenzione alle **assert** della classe **Riga**).

**NOTE:** Il metodo "**const Riga & operator=(const Riga & riga);**" vieta l'assegnamento tra righe di dimensioni diverse! Ciò evita che su un oggetto di tipo **Matrice** si possa cambiare la dimensione di qualche riga. Tale operatore permette però di cambiare in blocco il contenuto di una riga a parità di dimensione. Si noti inoltre che la classe **Matrice** è dichiarata "amica" di **Riga**. La classe **Matrice** non contiene altri metodi oltre quelli sopra dichiarati. Segue il codice di alcuni metodi della classe **Riga**:

<b>Riga::Riga()</b> : size(0) { vett = new int [size]; }
<b>Riga::Riga(int size)</b> : size(size) { vett = new int [size]; reset(); }
<b>Riga::Riga(const Riga &amp; riga)</b> : size(riga.size) { vett = new int [size]; for(int i = 0; i < size; i++) vett[i] = riga[i]; }
<b>const Riga &amp; Riga::operator= (const Riga &amp; riga)</b> { assert(size == riga.size); for(int i = 0; i < size; i++) vett[i] = riga.vett[i]; return *this; }

## Esercizio 2

Con riferimento alle classi `NodoInt`, `ListaInt`, `Iteratore`, data una lista di interi determinare l'ampiezza della più lunga sequenza contenente numeri progressivi.

Ad esempio data la lista [1, 4, 3, 4, 5, 6, 2, 3, 1, 2] il risultato sarà **4** (con riferimento alla sequenza evidenziata).

Scrivere un metodo che implementi tale funzionalità:

```
int computeB(ListaInt & list);
```

## Esercizio 3

Progettare le interfacce di una classe **InsiemeInt** usando l'ereditarietà a partire dalla classe **ListaInt**. Un insieme non ammette duplicati. La classe insieme dovrà **esclusivamente** avere:

- Un costruttore di **default** ed uno di **copia** (profonda)
- Un metodo di inserimento che avvisi se l'elemento da inserire è già presente o meno nell'insieme
- Un metodo di eliminazione che avvisi se l'elemento da eliminare era presente o meno nell'insieme
- Un metodo che svuota l'insieme
- Un metodo che verifichi se un elemento è nell'insieme
- Un metodo che ci dice se l'insieme è vuoto
- Un metodo che restituisca il numero di elementi dell'insieme
- Un operatore che dica se due insiemi sono uguali

**Si implementino infine i due metodi sottolineati**

## Soluzioni Traccia B

```
Matrice::Matrice(const Matrice & matrice) : rows(matrice.rows), columns(matrice.columns)
{
    mat = new Riga[rows];
    for(int i = 0; i < rows; i++)
    {
        mat[i].setSize(columns);
        mat[i] = matrice.mat[i];
    }
}

Matrice::~~Matrice()
{
    delete []mat;
}

Matrice Matrice::operator-(const Matrice & matrice) const
{
    assert(rows == matrice.rows && columns == matrice.columns);
    Matrice out(rows,columns);
    for(int i = 0; i < rows; i++)
        out[i] = mat[i] - matrice.mat[i];
    return out;
}

Riga& Matrice::operator[](int index)
{
    assert(0 <= index && index < rows);
    return mat[index];
}

bool Matrice::operator!=(const Matrice & matrice) const
{
    assert(rows == matrice.rows && columns == matrice.columns);
    for(int i = 0; i < rows; i++)
        if(mat[i] != matrice.mat[i])
            return true;
    return false;
}
```

---

**//con due iteratori**

```
int computeB_1(ListaInt & list)
{
    Iteratore iter1(list), iter2(list);
    iter1.VaiInTesta();
    iter2.VaiInTesta();
    iter2.MoveNext();
    int partial = 1, max = 1;
    while(!iter2.isNull())
    {
        if(iter1.getCurrentValue()+1 == iter2.getCurrentValue())
            partial++;
        else
        {
            if(partial > max)
                max = partial;
            partial = 1;
        }
        iter1.MoveNext();
        iter2.MoveNext();
    }
    if(partial > max)
        max = partial;
    return max;
}
```

//oppure con un solo iteratore

```
int computeB_2(ListaInt & list)
{
    Iteratore iter(list);
    iter.VaiInTesta();
    int partial = 1, max = 1;
    int prec = iter.getCurrentValue();
    iter.MoveNext();
    for( ; !iter.isNull() ; prec = iter.getCurrentValue(), iter.MoveNext())
        if(prec + 1 == iter.getCurrentValue())
            partial++;
        else
        {
            if(partial > max)
                max = partial;
            partial = 1;
        }
    return (partial > max) ? partial : max;
}
```

---

```
#ifndef INSIEME_INT_H
#define INSIEME_INT_H
#include "lista.h"
class InsiemeInt : protected ListaInt
{
    public:
        InsiemeInt();
        InsiemeInt(const InsiemeInt & set);
        bool add(int elem);
        bool remove(int elem);
        void clear();
        bool contains(int elem) const;
        bool isEmpty() const;
        int size() const;
        bool operator==(const InsiemeInt& set) const;
};

#endif
```

```
bool InsiemeInt::remove(int elem)
{
    NodoInt * tmp = cerca(elem);
    if(tmp == NULL)
        return false;
    del(tmp);
    return true;
}
```

```
bool InsiemeInt::isEmpty() const
{
    return getSize() == 0;
}
```