BoolVect

Sezioni

La traccia

Scopo della prova è progettare e implementare una gerarchia di oggetti utili a rappresentare dei *vettori di valori booleani* e un insieme di operazioni che li coinvolgano.

I vettori di valori booleani

Un *vettore di valori booleani*, di seguito **BoolVect**, è una sequenza di *valori di verità* ciascuno dei quali può essere *vero* o *falso*; le posizioni della sequenza sono indicizzate dai numeri naturali (rappresentabili con un `int`) e la *dimensione* di un BoolVect è definita come 1 più la posizione più grande in cui si trova un valore di verità uguale a vero (o 0 se tutti i valori di verità sono falsi); la *taglia* di un BoolVect è la massima dimensione che esso può raggiungere e può essere infinita (nel senso che, per convenzione, può coincidere con `Integer.MAX_VALUE`). Se il numero di valori di verità uguali a vero è molto inferiore rispetto alla taglia, il BoolVect si dice *sparso*, viceversa si dice *denso*.

La *rappresentazione testuale* di un BoolVect elenca i suoi valori di verità, da quello di posizione più grande a quello di posizione 0. Ad esempio, di seguito è riportato il BoolVect `FFFFVFFVVV` a cui è sovrapposto l'elenco delle posizioni dei suoi bit:

```
posizione 9876543210
 BoolVect FFFFVFFVVV
```

la dimensione di tale BoolVect è 6, dato che il valore vero di posizione maggiore (l'unico `V` circondato da due `F`) è in posizione 5.

Dato un BoolVect è possibile *leggere* e *scrivere* il suo i-esimo valore di verità; può essere plausibile che leggere un valore di verità in una posizione oltre la dimensione del BoolVect restituisca convenzionalmente il valore di verità falso.

Nota bene: implementare correttamente e in modo ragionevolmente efficiente i metodi `equals` (e quindi `hashCode`) per i BoolVect non è cosa banale, sopratutto in presenza di una gerarchia di sottotipi; per questa ragione l'implementazione di tali metodi è del tutto facoltativa.

Operatori logici e loro estensione ai BoolVect

Ci sono vari modi di definire *operazioni* binarie tra BoolVect; un modo molto pratico è partire dagli operatori logici binari, come ad esempio gli usuali *and*, *or* e *xor*, definiti rispettivamente come segue

- il valore di verità di `a & b` è vero se e solo se entrambi i valori di verità `a` e `b` sono veri;
- il valore di verità di `a | b` è vero se e solo se almeno uno dei valori di verità `a` e `b` è vero;
- il valore di verità di `a ^ b` è vero se e solo se esattamente uno dei valori di verità `a` e `b` è vero;

e quindi derivare le operazioni *componente a componente* tra BoolVect corrispondenti.

Più formalmente l'operatore logico binario $\cdot$, è esteso componente a componente all'operazione binaria tra BoolVect $\bar{\cdot}$ tale che l'i-esimo valore di verità di `u` $\bar{\cdot}$ `v` è dato dal risultato dell'operatore logico $\cdot$ tra l'i-esimo valore di verità di `u` e l'i-esimo di `v`.

Nota bene: osservi che non è necessario che i due argomenti di una operazione binaria tra BoolVect abbiano la medesima dimensione; d'altro canto, a seconda del tipo di operazione binaria, potrebbe essere dimostrabile che la taglia, o dimensione del risultato, siano in qualche

modo controllate da quelle degli operandi (se questo fosse necessario per la correttezza dei metodi che implementerà, non scordi di annotarlo esplicitamente nella documentazione dei medesimi).

Cosa è necessario implementare

Dovrà riflettere sulla mutabilità dei BoolVect e quindi specificare con le opportune segnature e comportamenti le varie operazioni (con particolare riferimento al ruolo di dimensione e taglia) e dovrà quindi realizzare *almeno due implementazioni distinte* di tali specifiche che siano adeguate l'una al caso di BoolVect sparsi e l'altra a quelli densi.

Osservi che è possibile provvedere ben più di una sola implementazione per i casi denso e sparso. Riguardo al primo, sono possibili diverse scelte riguardo alla taglia: essa può essere fatta ad esempio coincidere col numero di bit di un `byte`, `int` o `long` (il che suggerisce l'uso di una rappresentazione basata su una variabile di tipo primitivo), ma è certamente possibile avere anche taglie arbitrarie (ottenute con rappresentazioni basate su array, o liste). Nel caso dei BoolVect sparsi, viceversa, è plausibile scegliere una rappresentazione che consenta una taglia infinita (le posizioni dei pochi valori di verità veri potrebbero essere molto grandi, anche nell'ordine di `Integer.MAX_VALUE`).

Per verificare il comportamento del suo codice le può essere utile implementare una *classe di test* che, leggendo dal flusso di ingresso un elenco di azioni, le realizzi (creando le necessarie istanze di oggetti d'appoggio).

Le azioni, indicate una per riga, sono specificate da un carattere seguito da uno o più parametri; ciascuna azione produce un BoolVect come risultato, che va emesso nel flusso di uscita.

Le azioni sono:

- `S` seguita da una posizione intera `p`, un valore di verità `a` e un vettore booleani `u`; il risultato corrisponde al BoolVect `u` in cui il `p`-esimo valore di verità è `a`;
- `G` seguita da una posizione intera `p`, e un BoolVect `u`; il risultato corrisponde al `p`-esimo valore di verità di `u`;
- `&` seguito da due BoolVect `u` e `v`; il risultato corrisponde a `u & v`;
- `|` seguito da due BoolVect `u` e `v`; il risultato corrisponde a `u | v`;
- `^` seguito da due BoolVect `u` e `v`; il risultato corrisponde a `u ^ v`.

Ad esempio, avendo

```
S 0 V FFF
S 1 F VVV
G 0 FFV
G 1 FFF
G 1 FVF
& FVFV VVFF
| FVFV VVFF
^ FVFV VVFF
```

nel flusso di ingresso, la classe di test emette

```
V
VFV
V
F
V
VFF
VVFV
VFFV
```

nel flusso d'uscita (sono stati omessi gli `F` non significativi, ossia quelli di posizione maggiore rispetto a quella della `V` di posizione maggiore).

Nota bene: le implementazioni scelte devono poter funzionare, quantomeno nel caso di BoolVect sparsi, su dimensioni molto maggiori di quelle dell'esempio precedente!

La soluzione

Per prima cosa occorre progettare una *interfaccia* che rappresenti le competenze dei BoolVect; valuteremo in un secondo momento se sarà il caso di interporre una *classe astratta* tra questa interfaccia e le implementazioni concrete.

Va invece categoricamente esclusa sin dal principio l'idea di sviluppare un tipo per rappresentare i valori di verità del vettore: per tale scopo è più che sufficiente il tipo primitivo `boolean` o, eventualmente, il corrispondete tipo *wrapper* (o *involucro*) `Boolean`. Non c'è alcuna ragione plausibile di progettare un ulteriore tipo! L'esigenza di "convertire" dai caratteri `V` e `F` ai valori di verità e viceversa è banalmente soddisfatta rispettivamente dalle espressioni `c == 'V'` e `v ? 'V' : 'F'` (dove `c` è di tipo `char` e `v` è di tipo booleano).

L'interfaccia BoolVect

Le competenze informalmente descritte nella traccia sono:

- indicare la propria *taglia*,
- indicare la propria *dimensione*,
- *leggere* il valore di verità di data posizione,
- *scrivere* il valore di verità dato nella posizione assegnata,
- effettuare le operazioni booleane *and*, *or* e *xor*.

Per tradurre questa descrizione informale in una specifica precisa, descritta da una interfaccia, è necessario fare alcune scelte, considerandone attentamente le conseguenze.

Per quanto riguarda le prime tre competenze, essere corrispondono a tre metodi *osservazionali*

```
int dimensione();

int taglia();

boolean leggi(final int pos) throws IndexOutOfBoundsException;
```

[sorgente]

Unico dettaglio degno di nota è l'eccezione sollevata dal metodo di lettura, che accadrà senz'altro se la posizione richiesta è negativa; riguardo alle posizioni che eccedono la dimensione è plausibile (come discusso nella traccia) che tale funzione restituisca il valore `false`. Si può decidere abbastanza liberamente cosa prescrivere nel caso in cui sia addirittura ecceduta la taglia: sollevare eccezione, o restituire sempre `false`; la prima soluzione, come vedremo, è più consistente col caso della scrittura.

Maggior attenzione è richiesta dalle competenze che possono produrre cambiamenti nel BoolVect: è necessario riflettere sulla *mutabilità* dei BoolVect. L'operazione di scrittura può essere specificata sia come un metodo *mutazionale* (che cambi lo stato del BoolVect su cui è invocato), che come un metodo di *produzione* (che restituisca un nuovo BoolVect a ogni invocazione). La seconda scelta appare però troppo onerosa: una sequenza di invocazioni su un BoolVect di dimensione elevata produrrebbe una grande quantità di valori intermedi, probabilmente destinati ad avere una vita molto corta.

Appare quindi più ragionevole che l'interfaccia consideri le implementazioni di BoolVect *mutabili*.

```
void scrivi(final int pos, final boolean val) throws
IndexOutOfBoundsException;
```

[sorgente]

Anche in questo caso, l'eccezione riguarda certamente il caso in cui la posizione è negativa. Similmente, se si tenta di scrivere un valore di verità vero oltre la taglia non può che venir sollevata una eccezione. Secondo la traccia, infatti, la taglia è la massima dimensione possibile: anche qualora potesse aumentare durante la vita del BoolVect (e vedremo in seguito che ha poco senso), al momento della scrittura posizionare un valore di verità vero oltre la taglia farebbe aumentare la dimensione oltre la taglia, fatto vietato dalla traccia. Se viceversa il valore fosse falso (e la posizione sempre maggiore o uguale alla taglia), potrebbe essere sensato non sollevare alcuna eccezione e lasciare inalterato il vettore.

D'altro canto, aggiungere all'interfaccia un metodo che consenta di aumentare la taglia (ad esempio, prima della scrittura, per evitare l'eccezione di cui sopra), sarebbe una pessima idea perché escluderebbe tutte le implementazioni basate su rappresentazioni che non consentano tale adattamento (ad esempio, quella basata su un `long` che vedremo in seguito).

Scelta la mutabilità, anche le operazioni booleane sono specificate in modo che modifichino il BoolVect su cui sono invocate, rendendolo uguale al risultato dell'operazione.

```
void and(final BoolVect other) throws NullPointerException;

void or(final BoolVect other) throws NullPointerException,
IllegalArgumentException;

void xor(final BoolVect other) throws NullPointerException,
IllegalArgumentException;
```

[sorgente]

A prescindere dalle ovvie eccezioni legate al fatto che l'argomento sia `null`, anche in questo caso è necessario tener conto di un problema legato alla taglia: non è detto che il risultato possa sempre essere rappresentato modificando il primo operando.

Con un po' di riflessione risulta evidente che i metodi relativi alle operazioni booleane dovrebbero sollevare un'eccezione quando la taglia del primo operando è minore della dimensione del risultato: ad esempio l'or tra un BoolVect di taglia 2 e uno di dimensione 3 produce un BoolVect di dimensione 3, che evidentemente non può essere memorizzato nel primo operando di taglia 2. Questo certamente non può accadere nel caso dell'and, perché il risultato non può in nessun caso avere dimensione maggiore di quella del primo operando.

Per concludere, può essere utile aggiungere un metodo per rendere un BoolVect uguale ai valori specificati tramite una stringa; tale metodo può essere comodo per "inizializzare" un BoolVect in modo uniforme (rispetto alle varie implementazioni); tale metodo ammette una elementare implementazione di *default* a partire da un metodo che renda tutti i valori di verità pari al valore falso.

```
void pulisci();

default void daString(final String vals) throws
NullPointerException, IllegalArgumentException {
  pulisci();
  final int len = vals.length();
  for (int i = 0; i < len; i++) scrivi(i, vals.charAt(len - i - 1)
== 'V');
}
```

[sorgente]

Una implementazione parziale

A ben pensare, alcuni dei metodi dell'interfaccia possono essere sviluppati a partire dai soli metodi `leggi` e `scrivi`, per questa ragione sarebbe possibile aggiungere all'interfaccia stessa alcune implementazioni di *default*; va però osservato che anche i metodi `toString` e `equals` potrebbero essere scritti a partire dai soli `leggi` e `scrivi`, ma tali metodi non potrebbero essere realizzati come metodi di *default* (una interfaccia non può sovrascrivere i metodi di `Object`). Per tale ragione può aver senso introdurre una classe astratta (priva di stato, fatto positivo dal punto di vista dell'incapsulamento).

Ma c'è di più. È verosimile che le versioni *totali* di `leggi` e `scrivi` dell'interfaccia (che si devono prendere cura del valore della posizione e, nel caso, sollevare eccezioni) possano essere realizzate in modo molto semplice a patto di avere a disposizione due metodi *parziali* (che potremmo chiamare rispettivamente `leggiParziale` e `scriviParziale`) che operino sotto la pre-condizione che la posizione sia sempre compresa tra 0 (incluso) e la taglia (esclusa); inoltre, implementare tali versioni parziali per i sottotipi sarà senz'altro più semplice che implementare le versioni totali dell'interfaccia.

Il codice di questa parte della classe astratta è elementare

```
  protected abstract boolean leggiParziale(final int pos);

  protected abstract void scriviParziale(final int pos, final boolean
  val);

  @Override
  public boolean leggi(final int pos) throws
  IndexOutOfBoundsException {
    if (pos < 0) throw new IndexOutOfBoundsException("La posizione
  non può essere negativa.");
    return pos < dimensione() ? leggiParziale(pos) : false;
  }

  @Override
  public void scrivi(final int pos, final boolean val) throws
  IndexOutOfBoundsException {
    if (pos < 0) throw new IndexOutOfBoundsException("La posizione
  non può essere negativa.");
    if (pos >= taglia() && val)
      throw new IndexOutOfBoundsException(
          "Non è possibile scrivere un valore di verità vero in
  posizione maggiore o uguale alla taglia.");
    scriviParziale(pos, val);
  }
```

[sorgente]

Un'altra cosa di cui è possibile occuparsi a questo livello sono le operazioni booleane; ciascuna di esse può essere implementata con un ciclo della forma

```
final int maxDimension = Math.max(dimensione(), other.dimensione());
  for (int pos = 0; pos <= maxDimension; pos++)
    scrivi(pos, leggi(pos) OP other.leggi(pos));
```

dove `OP` è uno degli operatori booleani tra `&`, `|` e `^` di Java e il BoolVect `other` è, oltre a `this`, quello su cui operare. Sebbene a questo punto sia assolutamente plausibile ripetere questo ciclo tre volte (uno per ciascuna delle tre operazioni `and`, `or` e `xor`) sarebbe più elegante poter parametrizzare il ciclo rispetto all'operatore logico.

Parametrizzare gli operatori booleani

Purtroppo non è possibile avere un operatore del linguaggio come parametro di una funzione; per ottenere lo scopo desiderato, in Java è necessario definire una interfaccia che contenga un metodo che rappresenti una funzione che applica l'operatore ai suoi argomenti e ne restituisce il valore

```
public interface BooleanOperator {
  boolean apply(final boolean a, final boolean b);
}
```

[sorgente]

Grazie a questo approccio (che mima ad esempio quello visto per gli ordinamenti, basati su implementazioni dell'interfaccia `Comparable` che contiene il solo metodo `compareTo`), è possibile scrivere un metodo parziale che tramuti un operatore booleano in una operazione componente a componente tra BoolVect

```
public void componenteAComponente(BooleanOperator op, BoolVect
other)
    throws IndexOutOfBoundsException {
  final int dimensioneMax = Math.max(dimensione(),
other.dimensione());
  for (int pos = 0; pos <= dimensioneMax; pos++)
    scrivi(pos, op.apply(leggi(pos), other.leggi(pos)));
}
```

[sorgente]

A questo punto, usando le classi anonime, è possibile dare una implementazione parametrizzata rispetto all'operatore logico delle operazioni booleane tra BoolVect a livello della classe astratta

```
@Override
public void and(final BoolVect other) throws NullPointerException {
  componenteAComponente(
      new BooleanOperator() {
        @Override
        public boolean apply(boolean a, boolean b) {
          return a & b;
        }
      },
      Objects.requireNonNull(other, "L'argomento non può essere
null."));
}

@Override
public void or(final BoolVect other) throws NullPointerException,
IllegalArgumentException {
  try {
    componenteAComponente(
        new BooleanOperator() {
          @Override
          public boolean apply(boolean a, boolean b) {
            return a | b;
          }
        },
        Objects.requireNonNull(other, "L'argomento non può essere
null."));
  } catch (IndexOutOfBoundsException e) {
    throw new IllegalArgumentException(
        "La taglia di questo vettore è minore della dimensione del
risultato.");
  }
}

@Override
public void xor(final BoolVect other) throws NullPointerException,
IllegalArgumentException {
  try {
    componenteAComponente(
        new BooleanOperator() {
          @Override
          public boolean apply(boolean a, boolean b) {
            return a ^ b;
          }
        },
        Objects.requireNonNull(other, "L'argomento non può essere
null."));
  } catch (IndexOutOfBoundsException e) {
    throw new IllegalArgumentException(
        "La taglia di questo vettore è minore della dimensione del
risultato.");
  }
}
```

[sorgente]

Questo approccio è più complesso della duplicazione del codice, ma è ben più versatile: una volta messo in piedi esso potrebbe essere usato facilmente per estendere i comportamenti dei BoolVect a tutte le possibili operazioni booleane componente a componente!

Per concludere, si noti come, nel caso delle due operazioni il cui risultato potrebbe eccedere la taglia del primo operando, l'eccezione `IndexOutOfBoundsException` che potrebbe essere sollevata dal metodo `scrivi` venga sollevata al livello logico di una opportuna `IllegalArgumentException`.

La classe astratta si chiude con due metodi non ottimizzati che sovrascrivono quelli di `Object`

```
@Override
public String toString() {
  if (dimensione() == 0) return "F";
  final StringBuilder b = new StringBuilder();
  for (int i = dimensione() - 1; i >= 0; i--)
b.append(leggiParziale(i) ? 'V' : 'F');
  return b.toString();
}

@Override
public boolean equals(Object obj) {
  if (!(obj instanceof BoolVect)) return false;
  BoolVect altro = (BoolVect) obj;
  if (dimensione() != altro.dimensione()) return false;
  for (int i = dimensione() - 2; i >= 0; i--)    if (leggi(i) !=
altro.leggi(i)) return false;
  return true;
}
```

[sorgente]

Il contratto di `equals` (e l'overview della classe astratta) devono specificare molto chiaramente che è responsabilità delle sottoclassi provvedere una plausibile implementazione di `hashCode` (che non ha senso implementare al livello della classe astratta).

Per concludere osserviamo che la decisione di lasciare alle classi concrete l'implementazione dei metodi `taglia` e `dimensione` dell'interfaccia è legata al fatto che, come vedremo, tali funzioni sono molto semplici da implementare in modo efficiente avendo accesso alla rappresentazione.

Le implementazioni concrete

Avendo costruito con tanta attenzione la classe astratta appena descritta, il compito di implementare (e documentare) le sottoclassi concrete risulta immensamente semplificato (anche perché è sostanzialmente possibile evitare ogni ripetizione di codice).

Le implementazioni concrete devono provvedere i soli metodi (già documentati):

- `dimensione`, `taglia` e `pulisci`,
- `leggiParziale` e `scriviParziale`,
- `hashCode`;

inoltre possono (in modo del tutto facoltativo, se si ritiene utile e semplice farlo) provvedere delle implementazioni ottimizzate per i metodi:

- `and`, `or`, `xor`, e
- `equals`.

La traccia richiede (almeno) due implementazioni, una adatta al caso *denso* e una a quello *sparso*. È molto importante notare che la scelta di quale implementazione utilizzare (tra quelle che provvederete) sarà prerogativa esclusiva dell'utente di tali classi, pertanto non dovrà essere il vostro codice a prendere tale decisione (ad esempio in fase di costruzione o di modifica dei BoolVect, sopratutto non sulla scorta della frazione corrente di valori di verità veri).

Seguendo l'esempio dei polinomi, presentato nel libro di testo di Liskov et. al. e durante le lezioni ed esercitazioni, si possono individuare due rappresentazioni adatte, rispettivamente, ai due casi:

- un *array* di valori booleani, uno per ciascun valore di verità del BoolVect;
- un *insieme* di posizioni corrispondenti a tutti e soli i valori di verità veri del BoolVect.

La rappresentazione con un array condurrà a un BoolVect di taglia *finita* (pari al numero di elementi dell'array), mentre quella basata sull'insieme a un BoolVect di taglia *illimitata*. Le due rappresentazioni, oltre che per la loro semplicità, risultano quindi particolarmente interessanti anche perché consentono di confrontarsi (e mostrare la propria comprensione della traccia) con due casi distinti secondo il tipo di taglia.

Sostituendo una *lista* all'array è possibile trattare il caso denso con taglia illimitata, aumentando di poco la complessità del codice (rischiando di complicarsi la vita, senza mostrare però competenze più avanzate che nel caso dell'array). Si osservi che non ha però senz'altro senso che, in questo caso, il metodo `taglia` restituisca la dimensione corrente della lista: la taglia deve indicare la dimensione massima quindi i casi sono due:

- o non si è disposti a modificare la dimensione della lista (e allora l'implementazione basata su array è senz'altro più semplice e

preferibile);
- o viceversa si è sempre disposti ad aumentare la dimensione della lista (ad esempio a seguito di una scrittura), per cui è *scorretto* indicare una taglia finita.

Viceversa, sostituire una *lista* (o un array) all'insieme nella rappresentazione sparsa pone notevoli complessità implementative (ad esempio: la necessità di mantenere distinti gli elementi e di effettuare ricerche efficienti per ottenere la lettura e scrittura). Avendo a disposizione gli insiemi, l'uso di una lista appare una inutile complicazione.

Per finire, le *mappe* sono una pessima rappresentazione sia per il caso denso (se la chiave è la posizione) che quello sparso (se tra i valori vengono memorizzati anche quelli falsi).

Caso denso basato su array

La classe concreta più elementare è quella che deriva dalla classe astratta e rappresenta i valori del BoolVect in un array di tipo `boolean[]`. Per comodità aggiungiamo alla rappresentazione anche il valore precalcolato della dimensione

```
private final boolean[] valore;

private int dimensione = 0;


public ArrayBoolVect(final int taglia) {
  if (taglia <= 0) throw new IllegalArgumentException("La taglia
deve essere positiva.");
  valore = new boolean[taglia];
}
```

[sorgente]

l'invariante (oltre alla banale questione della nullità di `valore`) dovrà garantire che `dimensione` coincida sempre con la dimensione del BoolVect, ossia che in `dimensione - 1` si trovi l'ultimo `true` dell'array.

I primi tre metodi da implementare sono del tutto banali

```
@Override
public int taglia() {
  return valore.length;
}

@Override
public int dimensione() {
  return dimensione;
}

@Override
public void pulisci() {
  Arrays.fill(valore, false);
}
```

[sorgente]

ma anche letture e scritture (che possono assumere, dato che sono parziali, che la posizione sia sempre valida) sono molto semplici da scrivere

```
@Override
public boolean leggiParziale(final int pos) {
  return valore[pos];
}

@Override
public void scriviParziale(final int pos, final boolean val) {
  valore[pos] = val;
  if (val && pos >= dimensione) dimensione = pos + 1;
  else if (!val && pos == dimensione - 1)
    while (dimensione > 0 && !valore[dimensione - 1]) dimensione--;
}
```

[sorgente]

l'unico accorgimento è che la scrittura si occupi di tenere aggiornata la dimensione precalcolata (che può crescere se viene aggiunto un valore di verità vero, o deve essere diminuita se viene posto a falso il valore di verità che era in posizione più alta).

Data questa rappresentazione, per le operazioni booleane è difficile fare meglio di quanto implementato nel supertipo; per questa ragione la classe si può concludere con la sola ottimizzazione dei metodi di `Object`

```
@Override
public boolean equals(Object obj) {
  if (obj instanceof ArrayBoolVect) return Arrays.equals(valore,
((ArrayBoolVect) obj).valore);
  return super.equals(obj);
}

@Override
public int hashCode() {
  return Arrays.hashCode(valore);
}
```

[sorgente]

Caso sparso bastato su insieme

Se i valori di verità veri sono pochi, si può usare una struttura dati ben più onerosa di un array, sfruttando però il fatto che (appunto) conterrà pochi elementi. Gli interi sono naturalmente ordinati e conoscere la posizione più grande è comodo per calcolare la dimensione; la rappresentazione più pratica è quindi quella di un insieme ordinato

```
private final SortedSet<Integer> positions = new TreeSet<>();
```

[sorgente]

L'invariante è banalmente quello della nullità; i primi tre metodi (come nel caso denso) sono di immediata implementazione

```
@Override
public int taglia() {
  return Integer.MAX_VALUE;
}

@Override
public int dimensione() {
  return positions.size() > 0 ? 1 + positions.last() : 0;
}

@Override
public void pulisci() {
  positions.clear();
}
```

[sorgente]

Il fatto di non dover precalcolare la dimensione rende molto semplici anche i metodi parziali di lettura e scrittura

```
@Override
public boolean leggiParziale(final int pos) {
  return positions.contains(pos);
}

@Override
public void scriviParziale(final int pos, final boolean val) {
  if (val) positions.add(pos);
  else positions.remove(pos);
}
```

[sorgente]

In questo caso, vale però la pena di ottimizzare le operazioni booleane almeno nel caso in cui anche gli operandi siano sparsi; in tali circostanze, i metodi `and`, `or` e `xor` si riducono a operazioni su insiemi. Tali operazioni possono peraltro essere implementate in modo diretto a partire dai metodi di `Set` (come illustrato nell'approfondimento sul "Collections framework").

```
@Override
public void and(BoolVect other) throws NullPointerException {
  Objects.requireNonNull(other, "L'argomento non può essere
null.");
  if (other instanceof SetBoolVect)
positions.retainAll(((SetBoolVect) other).positions);
  else super.and(other);
}

@Override
public void or(BoolVect other) throws NullPointerException,
IllegalArgumentException {
  Objects.requireNonNull(other, "L'argomento non può essere
null.");
  if (other instanceof SetBoolVect) positions.addAll(((SetBoolVect)
other).positions);
  else super.or(other);
}

@Override
public void xor(BoolVect other) throws NullPointerException,
IllegalArgumentException {
  Objects.requireNonNull(other, "L'argomento non può essere
null.");
  if (other instanceof SetBoolVect) {
    Set<Integer> intersection = new TreeSet<>(positions);
    intersection.retainAll(((SetBoolVect) other).positions);
    positions.addAll(((SetBoolVect) other).positions);
    positions.removeAll(intersection);
  } else super.xor(other);
}
```

[sorgente]

Anche i metodi di `Object` si prestano a simili ottimizzazioni

```
@Override
public boolean equals(Object obj) {
  if (obj instanceof SetBoolVect) return
positions.equals(((SetBoolVect) obj).positions);
  return super.equals(obj);
}

@Override
public int hashCode() {
  return positions.hashCode();
}
```

[sorgente]

Caso denso basato su `long`

Sebbene una implementazione per ciascun genere di BoolVect sia sufficiente a superare la prova, riflettendo sull'opportunità di ottimizzare le operazioni booleane non si può non ricordare che il linguaggio Java mette a disposizione operatori booleani bit-a-bit per tutti i tipi numerici primitivi.

Questo suggerisce l'idea di usare un `long` per rappresentare BoolVect di taglia 64 (tanti sono i bit che tale tipo può memorizzare in Java)

```
private long bits = 0;
```

[sorgente]

L'invariante è sempre soddisfatto; i primi tre metodi sono ancora di immediata implementazione

```
@Override
public int taglia() {
  return Long.SIZE;
}

@Override
public int dimensione() {
  return Long.SIZE - Long.numberOfLeadingZeros(bits);
}

@Override
public void pulisci() {
  bits = 0;
}
```

[sorgente]

la dimensione può essere per pigrizia calcolata col metodo statico `Long.numberOfLeadingZeros`; l'uso dell'operatore di *shift* rende molto facili anche i metodi parziali di lettura e scrittura

```
@Override
public boolean leggiParziale(final int pos) {
  return (bits & (1L << pos)) != 0;
```
