
# Bloom Filters

Marco Costa
m.costa22@studenti.unipi.it
545144

# Indice

# Introduzione

L'obiettivo della relazione è presentare, analizzare ed utilizzare i Bloom Filter: una struttura dati probabilistica ideata da Burton Howard Bloom nel 1970 [2]. Essa è particolarmente efficiente in termini di spazio nel rappresentare insiemi, in quanto a differenza delle classiche strutture dati (alberi, tabelle hash, vettori), non richiede la memorizzazione dei dati stessi ma associa un determinato numero di bit ad uno o più elementi dell'insieme. La scelta di questo valore, in accordo alla dimensione della struttura, determina la probabilità di ottenere un falso positivo nel test di appartenenza all'insieme. Come verrà mostrato nel Capitolo 1, sono sufficienti 9.6 bit per elemento per una probabilità di falso positivo dell'1%.

La struttura non si limita ad una descrizione teorica e di ricerca, bensì ritrova applicazioni pratiche in numerosi contesti, in particolare per applicazioni di rete che richiedono un'implementazione efficiente di vari algoritmi hardware [4]. Nel tempo sono state descritte numerose varianti della struttura proposta inizialmente che permettono, ad esempio, l'eliminazione di elementi [6] o l'adeguamento dinamico della probabilità di falso positivo in relazione al numero di elementi inseriti [1].

Nel Capitolo 1 verrà descritta ed analizzata la struttura, la cui implementazione sarà esposta nel Capitolo 2 e infine utilizzata, testata e confrontata con altre strutture dati nel Capitolo 3.

# 1 Algoritmo

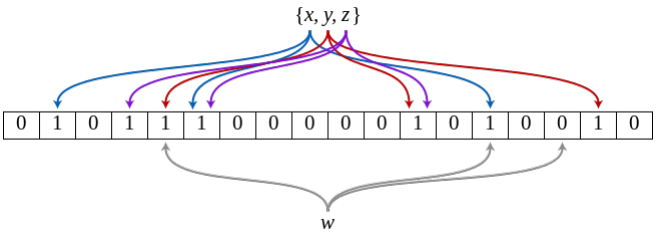



Figura 1: Esempio di Bloom Filter: gli elementi $x$,$y$,$z$ appartengono all'insieme, $w$ no.

Un Bloom Filter è un array di $m$ bits, tutti inizialmente impostati a 0. Un elemento è aggiunto all'insieme mediante l'utilizzo di $k$ funzioni hash uniformi e indipendenti[1], ognuna di esse produce una posizione del vettore che deve essere impostata ad 1. Per verificare l'appartenenza di un membro all'insieme è sufficiente computare nuovamente le $k$ funzioni hash e verificare che *tutte* le $k$ posizioni siano settate ad 1, in caso contrario l'elemento non vi appartiene (v. Figura 1).
È facile che diversi elementi dell'insieme condividano una o più posizioni all'interno del vettore, da questo deriva la natura probabilistica della struttura: le posizioni associate ad elementi diversi possono parzialmente o completamente sovrapporsi; le $k$ posizioni di un elemento non inserito potrebbero corrispondere a posizioni già occupate da altri elementi, producendo un *falso positivo* nel test di appartenenza; al contrario, se i falsi positivi sono ammessi, non lo sono i falsi negativi: se un elemento è parte dell'insieme allora tutti i suoi $k$ bit sono stati necessariamente settati ad 1; segue che le uniche risposte possibili al test di appartenenza sono: ***possibilmente nel set*** *o* ***definitivamente non nel set***.
La rimozione di un elemento non è permessa[2], infatti impostare a zero i $k$ bit associati ad un elemento potrebbe corrompere i bit associati ad altri elementi.

[1] Questo è vero solo in linea teorica: come ci sarà utile sapere più avanti, Kirsch e Mitzenmacher hanno dimostrato che l'utilizzo di combinazioni lineari di due funzioni hash permettono di implementare un Bloom Filter senza aumento di probabilità di falsi positivi [8].

[2] I Counting Bloom Filters [6] sono varianti della struttura che permettono la rimozione al prezzo di una dimensione maggiore (ma costante)

## 1.1 Risultati teorici

**Teorema 1.** *La probabilità di ottenere un falso positivo è (circa)*

$$p_f = \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{1}$$

*dove $n$ è il numero di elementi all'interno del set e le $k$ funzioni hash selezionano le posizioni dell'array con probabilità uniforme.*

*Dimostrazione.* La probabilità che un bit non venga settato ad 1 da una delle $k$ funzioni hash al momento dell'inserzione è

$$\left(1 - \frac{1}{m}\right)$$

per le $k$ funzioni hash vale

$$\left(1 - \frac{1}{m}\right)^k$$

se abbiamo inserito in precedenza $n$ elementi, la probabilità che un bit sia sempre 0 è data da

$$\left(1 - \frac{1}{m}\right)^{kn}$$

la probabilità che sia 1 dopo l'inserzione di $n$ elementi è la probabilità reciproca

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Ora supponiamo di effettuare il test di appartenenza di un elemento non nell'insieme. Affinché questo produca un falso positivo è necessario che ognuna delle $k$ posizioni computate dalla funzione hash sia impostata ad 1, il che causerebbe l'algoritmo a considerare l'elemento come parte del set

$$\begin{aligned} p_f &= \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \\ &= \left(1 - \left[1 - \frac{1}{m}\right]^{\frac{m}{m}kn}\right)^k \\ &\approx \left(1 - \left[\frac{1}{e}\right]^{\frac{kn}{m}}\right)^k \\ &= \left(1 - e^{\frac{-kn}{m}}\right)^k \end{aligned}$$

□

Lo stesso risultato è stato ottenuto da Mitzenmacher e Upfal rilassando il vincolo di avere funzioni hash con probabilità uniforme [9].

Notiamo che la probabilità di ottenere un falso positivo è funzione unicamente di $n$, $m$ e $k$. Fissando $m$ ed $n$ ed effettuando uno studio della derivata di (1) otteniamo che il valore di $k$ ($> 0$) che minimizza $p_f$ equivale a [3]:

$$k = \frac{m}{n} \ln 2 \tag{2}$$

ovvero il **numero ottimale di funzioni hash** in funzione di $m$ ed $n$. Un'analisi più approfondita di (1) ci porta al seguente risultato generalizzato:

**Teorema 2.** *Il numero ottimale di bit per elemento m/n e il numero ottimale di funzioni hash k sono funzione unicamente della probabilità desiderata* $p_f$.

$$\frac{m}{n} = -\frac{\log_2 p_f}{\ln 2} \simeq -1.44 \log_2 p_f \tag{3}$$

*e*

$$k = -\log_2 p_f \tag{4}$$

*Dimostrazione.* Sostituiamo (2) in (1):

$$\begin{aligned} p_f &= \left(1 - e^{-\left(\frac{m}{n} \ln 2\right) \frac{n}{m}}\right)^{\frac{m}{n} \ln 2} \\ &= 2^{-\frac{m}{n} \ln 2} \end{aligned}$$

$\Rightarrow$

$$\log_2 p_f = -\frac{m}{n} \ln 2 \tag{5}$$

da cui (3), e (4) sostituendo in (2). □

Da questo risultato deduciamo tre importanti caratteristiche:

1. Il numero ottimale di funzioni hash non ha relazione con $m$ ed $n$ ma solo con la probabilità di falso positivo $p_f$ desiderata
2. Conoscendo l'ordine del numero di elementi da inserire $O(n)$ possiamo calcolare la dimensione ottimale della struttura a partire da $p_f$
3. I valori $m$, $n$ e $k$ crescono con ordine logaritmico

Ad esempio, se volessimo costruire un Bloom Filter che contenga approssimativamente 10000 elementi con una probabilità percentuale di falso positivo dello 0.001% sarebbero necessarie $k = -\log_2 10^{-5} \simeq 17$ funzioni hash (o bit per elemento) e $m = -\dfrac{10^4 \times \log_2 10^{-5}}{\ln 2} \simeq 239626$ bit, ovvero circa 29.95 KB di memoria.

È importante rimarcare che ogni possibile analisi vale per strutture contenenti qualsiasi tipo di elemento, indipendentemente dalla loro dimensione intrinseca. Una tabella hash richiederebbe la memorizzazione di ogni elemento in memoria, costo che a seconda del tipo di dato può risultare proibitivo.

## 1.2 Analisi della complessità

Passiamo ora a determinare la complessità della struttura, sia in tempo che in spazio. Data la descrizione algoritmica e i risultati ottenuti nel precedente paragrafo otteniamo che:

- l'inserimento e la ricerca di un elemento nel set richiedono la computazione delle $k$ funzioni hash e, poiché questa è determinata unicamente da $p_f$, queste operazioni richiedono tempo costante $O(k)$
- la struttura occupa $m$ bit, dove $m = \frac{-n \log_2 p_f}{\ln 2}$ ed $n$ è il *numero* di elementi previsti, del tutto indipendente dal tipo dell'elemento. Il costo in spazio è banalmente $O(m)$

A differenza delle altre strutture dati utilizzate per implementare set, quali tabelle hash, alberi e array, il Bloom Filter è l'unica a richiedere un tempo *costante* per ricerca ed inserimento.
La Tabella 1 fornisce la comparazione al caso pessimo[3] con altre tipiche strutture dati.

## 1.3 Risultati attesi

La fase di test prevederà:

1. l'analisi dell'implementazione effettuata, in modo da poter verificare che questa rispetti i risultati teorici ottenuti in riferimento ai costi in tempo e spazio ed al numero di falsi positivi prodotti
2. la comparazione con due strutture dati Java utilizzabili per rappresentare insiemi: `LinkedList` ed `HashSet`

Per quanto riguarda il confronto dei costi in tempo, la prima di queste due strutture è, ovviamente, sfavorevole per questo utilizzo se non per valori di $n$ contenuti; il test di appartenenza si attende sia largamente inefficiente comparato a quello dell'implementazione. La seconda, invece, è uno degli standard *de facto* per questo scopo: sia la ricerca che l'inserimento richiedono costo (al caso medio) $O(1)$, risultando nettamente più efficienti di un Bloom Filter. Questo perché l'effettivo costo in inserimento e ricerca di quest'ultima è dato da $c_i \times k$, dove $c_i$ è il tempo di esecuzione della i-esima funzione hash.
Per quanto la scelta della funzione possa ridurre questo "costo" (come verrà analizzato nel paragrafo 2.1) esso è inevitabile e non può essere comparato a quello della singola computazione hash, seguita da accesso in memoria, effettuata da una tabella hash.

[3] tuttavia riferirsi al caso pessimo di una tabella hash è inattendibile, in quanto nella pratica predomina il caso medio

| | Double Linked List | Tabella Hash [caso medio] | Bloom Filter |
|---|---|---|---|
| Inserzione | $O(1)$ | $O(n)$ $[O(1)]$ | $O(k)$ |
| Ricerca | $O(n)$ | $O(n)$ $[O(1)]$ | $O(k)$ |
| Rimozione | $O(n)$ | $O(n)$ $[O(1)]$ | - |
| Spazio | $O(c \times n) = O(n)$ | $O(c \times n) = O(n)$ | $O(m)$ |

Tabella 1: Confronto di varie complessità al caso pessimo per la realizzazione di insiemi. La costante $c$ rappresenta la dimensione dell'elemento (e.g. 4 byte per un intero).

Il costo in spazio, invece, è la caratteristica principale dei Bloom Filter, sia LinkedList che HashSet prevedono un costo lineare rispetto al valore $n$ che include la memorizzazione dell'elemento in memoria; il Bloom Filter invece, prevede costo lineare senza memorizzazione dell'elemento in memoria.

Un altro importante aspetto da considerare riguarda la *politica di inserimento*: una politica *online* (ovvero che permetta l'inserimento di elementi nel set in modo dinamico) ha necessità di esaminare con maggiore enfasi sia il costo di inserimento (in quanto in un'ipotetica applicazione questa potrebbe influenzare il *feedback* dell'utente) che le stime teoriche effettuate al momento dell'allocazione di memoria: ad esempio, in un Bloom Filter, l'inserimento di un numero di elementi maggiore di $n$ provoca un aumento della probabilità $p_f$; in un HashSet, invece, l'aumento del numero di collisioni provoca una tendenza dei costi al caso pessimo risolvibile mediante una riallocazione della struttura, costosa a sua volta.
Una politica offline, invece, ci permette di ammortizzare il costo di inserimento limitandolo unicamente alla fase di caricamento dell'applicativo (influenzando solo in parte il feedback) e, inoltre, permette di mantenere invariate le stime effettuate in fase di allocazione.
Considereremo unicamente la politica *offline*.

## 2 Implementazione delle librerie

Realizzate con `Java 8`.

Distinguiamo tre fasi per la realizzazione della libreria:

1. interfaccia astratta che determini i metodi della struttura senza indicare la rappresentazione interna
2. scelta di una rappresentazione interna e implementazione dei metodi dell'interfaccia in una classe
3. utilizzo della classe per implementare un Set

Limitandoci alle funzioni basilari, l'interfaccia BloomFilter.java (Listing 1) definisce principalmente i metodi **set** e **isSet** per il settaggio e la verifica dei singoli bit della struttura, in modo del tutto indipendente ai risultati teorici ottenuti ed alla computazione di funzioni hash per l'aggiunta di elementi al Set, le quali sono problematiche legate ad un livello di astrazione superiore, svincolate dalla rappresentazione della struttura in memoria.

Dovendo definire in memoria un vettore di bit per implementare l'interfaccia e i metodi associati, Java fornisce sostanzialmente tre soluzioni, le quali, in base alla documentazione Oracle[4], differiscono come segue:

- array di `Boolean`: classe Java per il wrapping di una primitiva `boolean` in un oggetto. Dai 4 ai 20 bytes per elemento (in base all'implementazione).
- array di `boolean`: primitiva Java per la rappresentazione di valori booleani, 1 byte per elemento.
- `BitSet`: classe che implementa un vettore di bit di dimensione dinamica, ogni elemento richiede 1 bit.

Cercando di ottimizzare l'efficienza in spazio implementiamo l'interfaccia utilizzando un BitSet, impedendo l'esposizione di metodi che causino il ridimensionamento della struttura (classe BitSetBloomFilter.java, Listing 2).

Una volta implementata la struttura in memoria possiamo definire la classe BloomSet.java (Listing 3) che estenda le interfacce del linguaggio per la definizione di `Set<E>` per oggetti di tipo generico e l'implementazione dei metodi necessari. Poiché desideriamo che la classe rispetti i risultati teorici ottenuti, il costruttore accetta come parametri la dimensione $n$ e la probabilità di falso positivo $p_f$ e computa il numero di bit $m$ e il numero di funzioni hash $k$ mediante le equazioni (3) e (4).
L'unica scelta implementativa rimasta riguarda le funzioni hash.

[4] `docs.oracle.com`

## 2.1 Scelta della funzione Hash

La scelta e la computazione di un numero elevato di funzioni hash può risultare proibitivo; fortunatamente, Kirsch e Mitzenmacher hanno dimostrato che la combinazione di due funzioni hash nella forma $g_i(x) = h_1(x) + ih_2(x)$ con $1 \leq i \leq k$ non provoca un aumento asintotico della probabilità di falso positivo $p_f$ [8].

L'utilizzo di funzioni hash crittografiche è inefficiente ed il costo di computazione eccessivo. Delle possibili funzioni non crittografiche è stata scelta la funzione Murmur3 poiché fornisce risultati soddisfacenti sotto forma di falsi positivi, velocità di computazione e indipendenza [7], inoltre ne è presente un'implementazione all'interno della libreria `Google Guava`.

Per ottimizzare la velocità di computazione della libreria vengono ottenute due funzioni hash indipendenti a 32 bit mediante lo shift dei bit della computazione della singola funzione hash a 64 bit (v. Listing 4).

# 3 Applicazione e Test: `BloomBlocker`

I Bloom Filters non sono strutture unicamente teoriche ma ritrovano innumerevoli applicazioni pratiche, in molti casi vengono utilizzati per ridurre il numero di lookup su disco, nella fattispecie rappresentano un primo "scoglio" durante l'operazione di ricerca: se l'argomento di query è definitivamente non sul disco la richiesta non viene propagata in memoria secondaria. Ad esempio DBMS come Apache Cassandra e PostgreSQL utilizzano Bloom Filters per ridurre i lookup su disco di righe o colonne non esistenti, incrementando la performance di query [5].

`BloomBlocker` realizza un semplice url-blocker, ovvero un applicativo che, dato un file contenente una lista di host considerati non sicuri, espone un metodo per la verifica di un dato url all'interno della lista. La maggior parte dei browser ne implementa uno al suo interno: ogni volta che un utente richiede di visitare un sito non presente nella web cache viene verificato che questo non faccia parte di una lista di domini non attendibili e potenzialmente dannosi.

Effettuare un lookup su disco per ogni url (immaginiamo un utente che naviga sul web o peggio un server proxy) è, ovviamente, sterile ed inopportuno. Da qui la necessità di dover rappresentare la lista in memoria per poter velocizzare l'operazione di ricerca. Questo sviluppa una domanda: *un Bloom Filter, data la sua struttura probabilistica, è una struttura dati adeguata per questa operazione?* Un elemento di test può risultare *possibilmente* un url malevolo (con probabilità $1 - p_f$) o definitivamente un url non malevolo; ciò è favorevole dato che un normale utilizzo dovrebbe richiedere un numero di url sicuri decisamente maggiori. Se si volesse la certezza assoluta del risultato è possibile effettuare una ricerca all'interno del file *solo* se l'url è risultato possibilmente malevolo.

La classe `BloomBlocker.java` (Listing 5) è, sostanzialmente, un wrapper per la classe `BloomSet`: essa consiste in un metodo per effettuare il parsing del file host, un metodo **checkDomain** per la verifica di un dominio e un costruttore che, a partire da un file di host e la probabilità di falso positivo $p_f$, costruisce un oggetto BloomSet con la probabilità definita ed $n$ uguale al numero di host all'interno del file (il quale, come detto, computa il valore teorico ottimale di $m$ e $k$).

I test saranno effettuati utilizzando questo applicativo e confrontandolo con due applicazioni identiche realizzate mediante LinkedList e HashSet. I test saranno poi confrontati con i risultati attesi nella Sezione 1.3.

## 3.1 Test

File utilizzati:

- *hosts*: file di testo contenente un elenco di host malevoli riconosciuti[5]. Dimensione: 1.60MB, numero di elementi: 57563.
- *google_host*: file di testo contenente un elenco di host sicuri[6]. Nessun elemento è presente in *hosts*. Dimensione: 770KB, numero di elementi: 16331.

Il primo file sarà utilizzato per produrre gli elementi del Set, il secondo come insieme disgiunto per il test di appartenenza.

### 3.1.1 Numero di falsi positivi

Il primo test effettuato riguarda il numero di falsi positivi ottenuti. Poiché definiamo come parametro dell'oggetto `BloomBlocker` la probabilità di falso positivo $p_f$, sia $j$ il numero di elementi testati, allora il numero di falsi positivi ottenuti ci aspettiamo sia $\approx jp_f$.

In Figura 2 è mostrato il risultato del test; analizzando il grafico notiamo che:

- dallo 0.0001% allo 0.01% di probabilità $p_f$ (percentuale) non sono stati prodotti falsi positivi
- i primi falsi positivi appaiono con una probabilità dello 0.1%
- la stima teorica data dalla probabilità di falso positivo moltiplicato il numero di elementi testati è assolutamente sovrapponibile ai risultati ottenuti

Di conseguenza vengono (almeno per questo test) convalidati i risultati teorici ottenuti in (3) e (4): i parametri selezionati $m$, $n$ e $k$ provocano una (quasi) effettiva probabilità di falso positivo $p_f$.

### 3.1.2 Dimensione in memoria

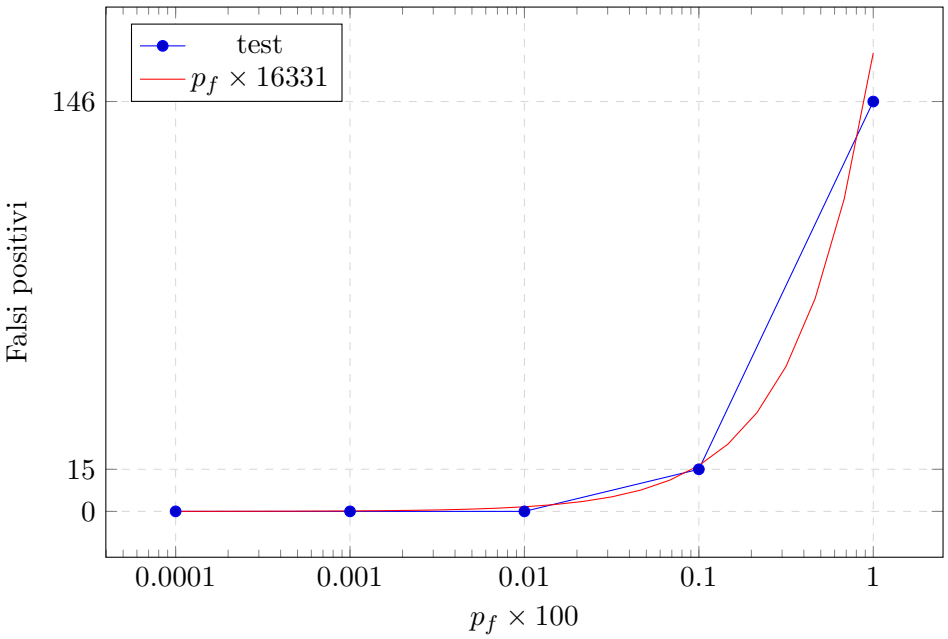
A partire dal test precedente possiamo analizzare $p_f$ in funzione della quantità di memoria occupata dal Bloom Filter. In Figura 3 è mostrato il risultato del test.

La struttura occupa unicamente gli $m$ bit computati nell'equazione (3)[7]: sono sufficienti 134 KB di memoria per una struttura con probabilità di falso

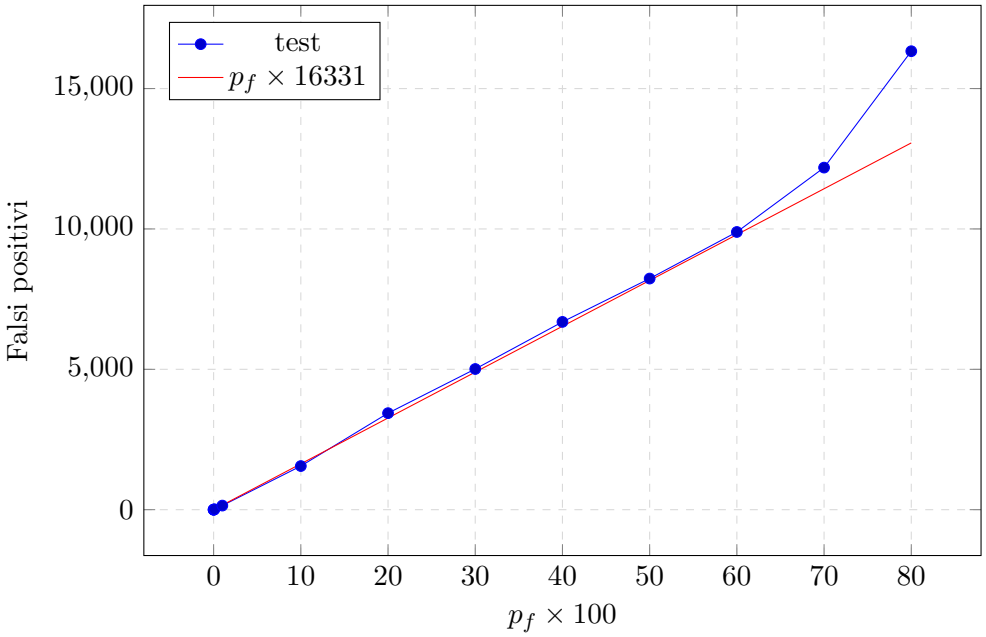
[5] Disponibile su: `https://github.com/StevenBlack/hosts`

[6] `https://github.com/googlehosts/hosts`

[7] in realtà la dimensione è leggermente maggiore ed è dovuta al padding effettuato dalla classe `BitSet`



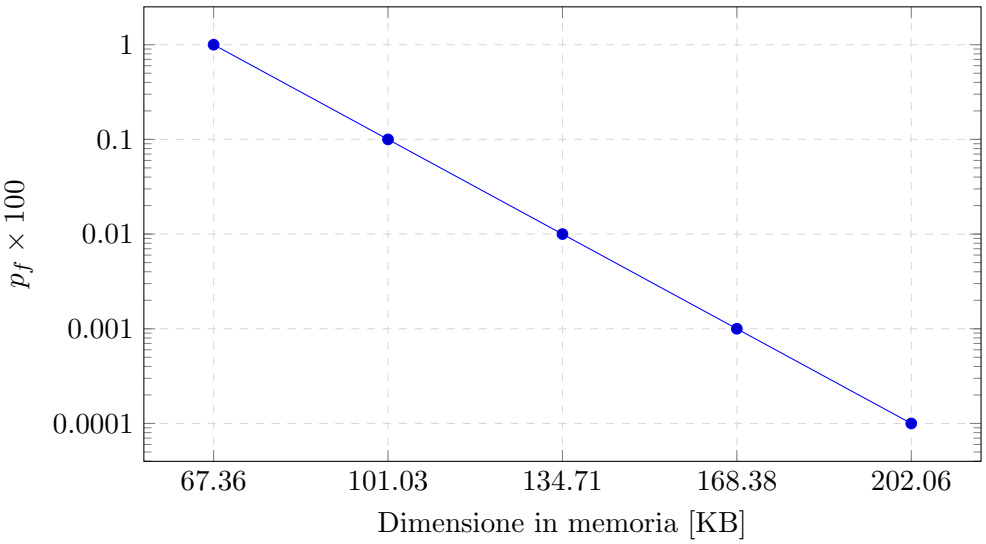

(a) Probabilità di falso positivo compresa tra lo 0.0001% e 1%. **Nota:** l'asse x è scalato logaritmicamente



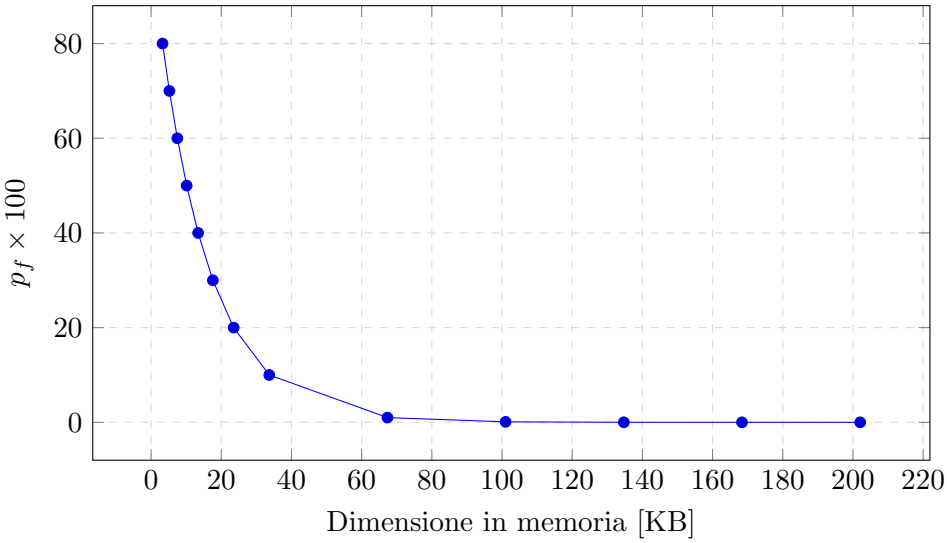

(b) Probabilità di falso positivo compresa tra lo 0.0001% e l'80%.

Figura 2: Test sui falsi positivi: in blu in risultato del test, in rosso la stima teorica.

positivo dello 0.01% contenente 57.563 stringhe di testo e che, in questo test, non ha prodotto falsi positivi. Con poco più di 200 KB la probabilità scende allo 0.0001%.




(a) Dettaglio della Figura (b). Probabilità sotto l'1%




(b) Probabilità tra lo 0.0001% e l'80%

Figura 3: Variazione della probabilità di falso positivo in relazione alla dimensione della struttura

### 3.1.3 Confronto con `LinkedList` e `HashSet`

I test sono stati effettuati utilizzando:

- la libreria `Java Instrumentation` per quanto concerne il calcolo della dimensione in memoria degli oggetti
- il tool `OpenJDK JMH` per il benchmark dei tempi di esecuzione; sono state impostate 5 iterazioni di warmup e 5 di misurazione. Il risultato mostrato è la media dei tempi.
- processore Intel Core i5-8250U e 8 GB di memoria.

Passiamo ora a confrontare le tre strutture dati e analizzare i risultati in accordo alla Sezione 1.3.
In Figura 4 è evidente come, a fronte di quasi sessantamila stringhe inserite all'interno delle strutture, il Bloom Filter richieda una quantità di memoria microscopica rispetto alle concorrenti, persino un Bloom Filter con probabilità di falso positivo esageratamente alta ($10^{-80}$) richiede meno della metà dello spazio necessario ad un HashSet. Un BloomSet con $p_f = 10^{-7}$ necessita di poco più di 200 KB di memoria unicamente legata ai bit, HashSet e LinkedList di circa 6 MB causate dalla linearità in spazio e dalla memorizzazione dell'elemento in memoria.

In Figura 5 sono rappresentati i tempi di esecuzione per il test di appartenenza sugli elementi del test set. Oltre alla struttura realizzata è stata valutata la performance dell'implementazione presente nella libreria `Google Guava`, la quale è sicuramente una delle più efficienti del linguaggio `Java`.

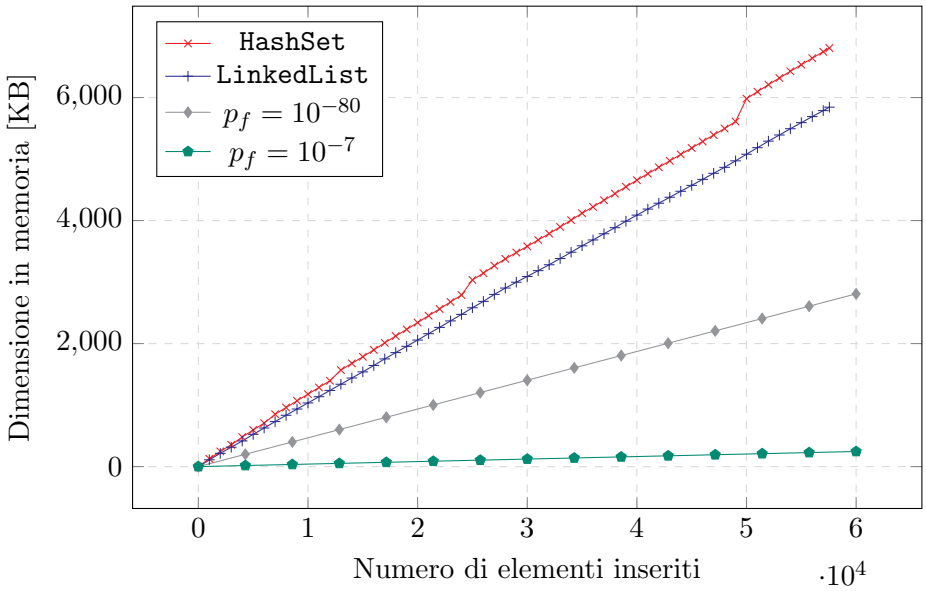
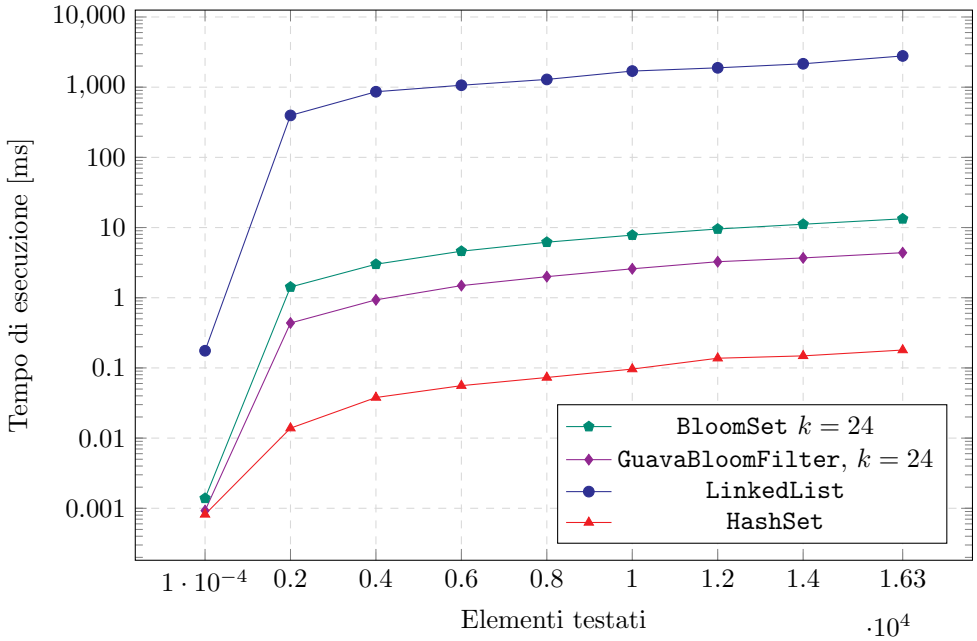
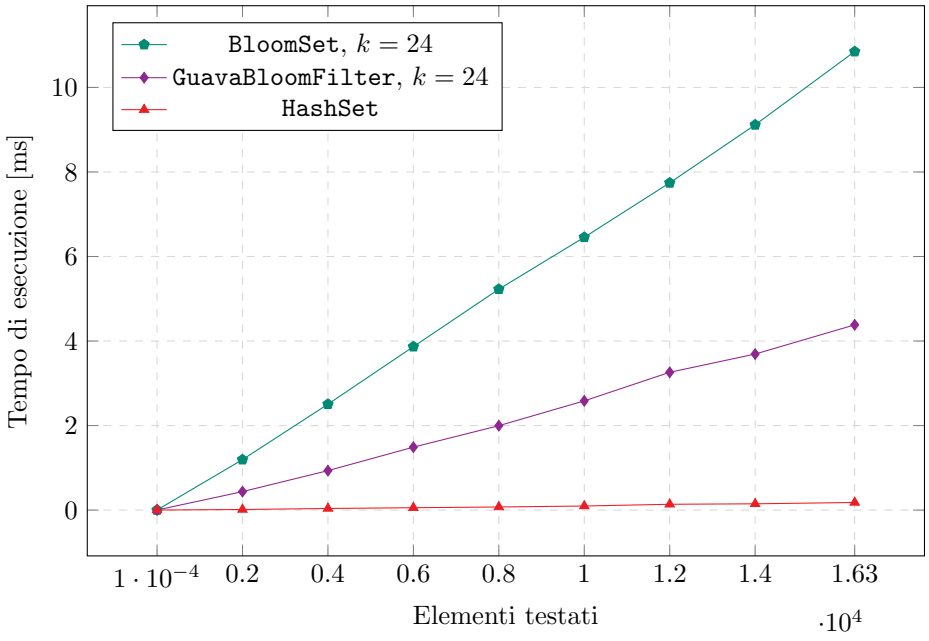



Figura 4: Dimensione in memoria delle strutture. È stato comparato un HashSet ed una LinkedList con tre Bloom Filter aventi diverse probabilità.




(a) Comparazione tra i vari ordini di grandezza necessari. **Nota:** l'asse y è scalato logaritmicamente




(b) Dettaglio della Figura (a)

Figura 5: Benchmark sull'operazione di test di appartenenza

## 3.2 Conclusioni

Dai test possiamo concludere che le previsioni effettuate nella Sezione 1.3 sono state rispettate.
Come ci aspettavamo HashSet è nettamente più performante di un BloomFilter (anche comparandolo ad una migliore implementazione) ma, come tutte le cose belle, ha un costo (ed anche elevato, a seconda dei contesti). Ne concludiamo che grazie ad un notevole risparmio di memoria, soprattutto in contesti che richiedono la gestione di grandi quantità di oggetti di dimensioni arbitrarie o vincolano le risorse allocabili, a fronte di una perdita di efficienza questa struttura può risultare una scelta particolarmente valida a patto di effettuare preventivamente una stima teorica sulla soglia di falso positivo accettabile dal contesto.

Per quanto concerne `BloomBlocker`, essa si posiziona come applicativo che avvantaggia particolarmente questo tipo di struttura in determinate circostanze, come ad esempio parte di un web browser. Poiché generalmente la memoria RAM richiesta da questo genere di applicativi risulta elevata e dovendo soddisfare tempi di reazione umani (oltre ad attuare un singolo test di appartenenza alla volta), potrebbe risultare preferibile ottimizzare il consumo in spazio piuttosto che in tempo e di conseguenza affidarsi ad una struttura di questo genere.

## Riferimenti bibliografici


[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] James Blustein and Amal El-Maazawi. Bloom filters. a tutorial, analysis, and survey. *Halifax, NS: Dalhousie University*, pages 1–31, 2002.

[4] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[6] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[7] Marc Antoine Gosselin-Lavigne, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. A performance evaluation of hash functions for ip reputation lookup using bloom filters. In *2015 10th International Conference on Availability, Reliability and Security*, pages 516–521. IEEE, 2015.

[8] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[9] Eli Upfal and Michael Mitzenmacher. *Probability and computing: randomized algorithms and probabilistic analysis*, volume 160. Cambridge university press, 2005.

# Appendice

## Sorgenti

Listing 1: BloomFilter.java

```
package bloom;

import java.io.Serializable;

/**
 * Interfaccia astratta per l'implementazione di BloomFilter.
 * Espone metodi per il calcolo delle stime teoriche ottimali.
 *
 * @author Marco Costa
 */
public abstract class BloomFilter implements Serializable {
    private static final double ln2 = Math.log(2);

    private static final double logbase2(double x) {
        return Math.log(x) / ln2;
    }

    /**
     * Calcola il valore ottimo di K a partire dalla probabilità pf.
     *
     * @param pf la probabilità di falso positivo in (0, 1)
     * @return il valore ottimo di K
     */
    public static final int computeK(double pf) {
        /* K = -log_2(pf) */
        return new Double(Math.round(-logbase2(pf))).intValue();
    }

    /**
     * Calcola il valore ottimo di M a partire dalla probabilità pf
         e dal
     *  numero di inserimenti previsti N.
     *
     * @param n il numero di inserimenti previsti
     * @param pf la probabilità di falso positivo in (0, 1)
     * @return il valore ottimo di M
     */
    public static final int computeM(int n, double pf) {
        /* M = (n * -log_2(pf))/(ln(2)) */
        return new Double(Math.ceil((n * -logbase2(pf)) /
            ln2)).intValue();
    }
```

```
    /* metodi per il settaggio di bit a 1 */
    abstract public void set(int index);
    abstract public void set(int[] index);

    /* metodi per la verifica di bit con valore 1 */
    abstract public boolean isSet(int index);
    abstract public boolean isSet(int[] index);

    /* metodi per la gestione dell'array */
    abstract public void clear();
    abstract public int length();
    abstract public int size();
    abstract public boolean isEmpty();
}
```

Listing 2: BitSetBloomFilter.java

```
package bloom;

import java.util.BitSet;

/**
 * Implementazione dell'interfaccia astratta BloomFilter mediante
     BitSet array.
 *
 * @see BloomFilter
 * @author Marco Costa
 */
public class BitSetBloomFilter extends BloomFilter {
    private final BitSet array;
    private final int size;

    /**
     * Creazione di un nuovo BloomFilter implementato mediante
         BitSet di dimensione size bit.
     *
     * @param size la dimensione in bit
     */
    public BitSetBloomFilter(int size) {
        this.size = size;
        array = new BitSet(size); /* bit settati a 0 */
    }

    /**
     * Verifica che pos sia un indice valido del BloomFilter.
     *
     * @param pos l'indice
     * @throws IndexOutOfBoundsException se pos non è un indice
         valido
```

```
 */
private void checkPosition(int pos) throws
    IndexOutOfBoundsException {
    if ((pos < 0) || (pos > size))
        throw new IndexOutOfBoundsException("Indice " + pos + "
            non valido");
}

/**
 * Imposta a true l'indice index del Bloom Filter.
 *
 * @param index l'indice
 * @throws IndexOutOfBoundsException se index non è un indice
     valido
 */
public void set(int index) throws IndexOutOfBoundsException {
    checkPosition(index);
    array.set(index);
}

/**
 * Imposta a true tutti gli indici in index del Bloom Filter.
 * Se uno degli indici contenuti in index non è valido
     l'operazione
 *  non viene eseguita.
 *
 * @param index il vettore di indici
 * @throws IndexOutOfBoundsException se index contiene un indice
     non valido
 */
public void set(int[] index) throws IndexOutOfBoundsException {
    for(int i : index)
        checkPosition(i);
    for(int i : index)
        array.set(i);
}

/**
 * Restituisce true sse l'indice index è settato a true.
 *
 * @param index l'indice della struttura
 * @return tt sse array[index] = 1
 * @throws IndexOutOfBoundsException se index non è un indice
     valido
 */
public boolean isSet(int index) throws
    IndexOutOfBoundsException {
    checkPosition(index);
    return array.get(index);
```

```
}

/**
 * Restituisce true sse tutti gli indici in index sono settati a
    true.
 *
 * @param index il vettore di indici
 * @return tt sse forall i in index -> array[i] = 1
 * @throws IndexOutOfBoundsException se index contiene un indice
    non valido
 */
public boolean isSet(int[] index) throws
    IndexOutOfBoundsException {
    for (int i : index)
        if (!isSet(i))
            return false;

    return true;
}

/**
 * Reimposta tutti i bit della struttura a false.
 */
public void clear() {
    array.clear();
}

/**
 * Restituisce la dimensione effettiva in bit in memoria della
    struttura. Può non coincidere con la
 * dimensione impostata.
 *
 * @return la dimensione in bit in memoria
 */
public int size() {
    return array.size();
}

/**
 * Restituisce la dimensione logica della struttura. L'indice
    dell'ultimo bit più uno.
 *
 * @return la dimensione logica della struttura
 */
public int length() {
    return array.length();
}
```

```
    /**
     * Restituisce true sse tutti gli indici della struttura sono
         settati a false.
     *
     * @return tt sse forall i in size -> array[i] = 0
     */
    public boolean isEmpty() {
        return array.isEmpty();
    }
}
```

Listing 3: BloomSet.java

```
package bloom;

import hash.BloomHash;
import hash.HashFactory;

import java.io.Serializable;
import java.util.AbstractSet;
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;

/**
 * Optimal Bloom Set.
 * Set di elementi di tipo E realizzato mediante Bloom Filter e
 *  calcolo di stime teoriche ottimali.
 *
 * @param <E> il tipo dell'elemento, deve essere Serializzabile
 * @author Marco Costa
 */
public class BloomSet<E extends Serializable> extends AbstractSet<E>
        implements Set<E>, Cloneable, Serializable {
    private final BloomFilter array;

    private final int k;
    private final int m;

    private int n = 0;

    private final BloomHash hash = HashFactory.getHashFunction();

    /**
     * Creazione di un nuovo Bloom Filter con numero di elementi n e
         probabilità
     *  di falso positivo pf.
     * La dimensione in memoria e il numero di funzioni Hash sono
         calcolate a partire
```

```
 *  dai due parametri.
 *
 * @param n ordine del numero di elementi dell'insieme
 * @param pf probabilità di falso positivo richiesta in (0, 1)
 */
public BloomSet(int n, double pf) {
    if((pf <= 0) || (pf >= 1))
        throw new IllegalArgumentException("pf deve essere
            compreso tra 0 e 1");
    if(n <= 0)
        throw new IllegalArgumentException("n deve essere
            maggiore di 0");

    k = BloomFilter.computeK(pf);
    m = BloomFilter.computeM(n, pf);

    array = new BitSetBloomFilter(m);
}

/**
 * Aggiunta di un elemento al Set.
 *
 * @param e l'elemento da aggiungere
 * @return tt
 */
@Override
public boolean add(E e) {
    array.set(hash.hashObject(e, k, m));
    n++;
    return true;
}

/**
 * Aggiunta di una collezione di elementi al Set.
 *
 * @param c la collezione da aggiungere
 * @return tt
 */
@Override
public boolean addAll(Collection<? extends E> c) {
    for(E item : c)
        add(item);
    return true;
}

/**
 * Restituisce true se l'elemento è possibilmente nel set, false
 *  se l'elemento è definitivamente non nel set.
 * La probabilità di falso positivo è pf.
```

```
 *
 * @param o l'elemento da verificare
 * @return true se l'elemento è possibilmente nel set,
 *         false se l'elemento è definitivamente non nel set
 */
@Override
public boolean contains(Object o) {
    return array.isSet(hash.hashObject(o, k, m));
}

/**
 * Restituisce true se la collezione di elementi è possibilmente
     nel set,
 * false se la collezione è definitivamente non nel set.
 * La probabilità di falso positivo è pf ^ c.size().
 *
 * @param c la collezione da verificare
 * @return true se la collezione è possibilmente nel set,
 *         false se la collezione è definitivamente non nel set
 */
@Override
public boolean containsAll(Collection<?> c) {
    for(Object o : c)
        if(!contains(o)) return false;

    return true;
}

/**
 * Il BloomFilter non supporta iterazione.
 * Lancia una UnsupportedOperationException.
 *
 * @return
 */
@Override
public Iterator<E> iterator() {
    throw new UnsupportedOperationException("La struttura non
        supporta iterazione");
}

/**
 * Il BloomFilter non supporta rimozione.
 * Lancia una UnsupportedOperationException.
 *
 * @param o
 * @return
 */
@Override
public boolean remove(Object o) {
```

```
        throw new UnsupportedOperationException("La struttura non
            supporta la rimozione");
    }

    /**
     * Restituisce il numero di elementi aggiunti al Set.
     *
     * @return
     */
    @Override
    public int size() {
        return n;
    }

    /**
     * Restituisce true se nessun elemento è stato aggiunto al Set.
     *
     * @return
     */
    @Override
    public boolean isEmpty() {
        return array.isEmpty();
    }

    /**
     * Reimposta la struttura come vuota.
     */
    @Override
    public void clear() {
        array.clear();
    }

    /**
     * Restituisce la dimensione fisica in memoria.
     *
     * @return
     */
    public int dimension() {
        return array.size();
    }

}
```

Listing 4: Murmur3_Hash.java

```
package hash;

import com.google.common.hash.HashFunction;
import com.google.common.hash.Hashing;
```

```java
import java.io.*;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;

/**
 * Classe Singleton per il calcolo di funzioni hash per Bloom Filter
 *  mediante funzione Murmur3.
 * Basata su Same Performance Less Hashing.
 *
 * g_i(x) = h1(x) + i*h2(x)
 *
 * @author Marco Costa
 */
public class Murmur3_Hash implements BloomHash {
    private static final Murmur3_Hash Instance;

    private final HashFunction hash1;

    private static final int SEED_1 = 0;

    static {
        Instance = new Murmur3_Hash();
    }

    private Murmur3_Hash() {
        hash1 = Hashing.murmur3_128(SEED_1);
    }

    public static Murmur3_Hash getInstance() { return Instance; };

    /**
     * Conversione di un oggetto in array di byte.
     *
     * @param in l'oggetto da convertire
     * @return l'array di byte
     * @throws IOException se non è possibile convertire l'oggetto
     */
    private static byte[] objectToBytes(Object in) throws
        IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();

        try (ObjectOutput out = new ObjectOutputStream(bos);)
        {
            out.writeObject(in);
            out.flush();
        }
        catch (IOException ex) {
            throw new IOException();
```

```
    }

    return bos.toByteArray();
}

/**
 * Computazione delle k posizioni risultanti dall'operazione di
     combinazione
 *  g_i(x) = h1(x) + ih2(x)
 *
 * @param h1 la prima f. hash
 * @param h2 la seconda f. hash
 * @param k il numero di iterazioni (o posizioni da computare)
 * @param bound limite superiore per ogni posizione
 * @return il vettore di posizioni
 */
private int[] computeHash(int h1, int h2, int k, int bound) {
    int[] out = new int[k];

    for(int i = 0; i < k; i++)
    {
        out[i] = h1 + (i + 1) * h2;
        if(out[i] < 0)
            out[i] = ~out[i];

        out[i] = out[i] % bound;
    }

    return out;
}

/**
 * Effettua l'hash di un oggetto serializzabile e ne restituisce
     il vettore
 *  di k posizioni, tale che ogni posizione sia < bound.
 *
 * @param o l'oggetto
 * @param k il numero di posizioni
 * @param bound il limite superiore per ogni posizione
 * @return il vettore di posizioni
 */
public int[] hashObject(Object o, int k, int bound) {
    if(!(o instanceof Serializable))
        throw new IllegalArgumentException("l'oggetto deve
            essere serializzabile");

    byte[] byteObject;

    try {
```

```
            byteObject = Murmur3_Hash.objectToBytes(o);
        } catch (IOException e) {
            throw new IllegalArgumentException("l'oggetto non può
                essere convertito in byte");
        }

        long hash64 = hash1.hashBytes(byteObject).asLong();
        int h1 = (int) hash64;
        int h2 = (int) (hash64 >>> 32);

        return computeHash(h1, h2, k, bound);
    }

    /**
     * Effettua l'hash di una stringa UTF-8 e ne restituisce il
         vettore
     *  di k posizioni, tale che ogni posizione sia < bound.
     *
     * @param s la stringa UTF-8
     * @param k il numero di posizioni
     * @param bound il limite superiore per ogni posizione
     * @return il vettore di posizioni
     */
    public int[] hashObject(String s, int k, int bound) {
        long hash64 = hash1.hashString(s,
            StandardCharsets.UTF_8).asLong();
        int h1 = (int) hash64;
        int h2 = (int) (hash64 >>> 32);

        return computeHash(h1, h2, k, bound);
    }

}
```

Listing 5: BloomBlocker.java

```
package bloom;

import hash.BloomHash;
import hash.HashFactory;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Collection;
```

```
import java.util.function.Predicate;
import java.util.regex.Pattern;
import java.util.stream.Stream;

/**
 * BloomBlocker.
 *
 * @author Marco Costa
 */
public class BloomBlocker {
    private final BloomSet<String> set;

    /**
     * Restituisce la lista dei domini contenuti all'interno del
         file "filename".
     *
     * @param filename il nome del file di host
     * @return la lista dei domini
     * @throws IOException se il file non esiste
     */
    public static final ArrayList<String> loadHostfile(String
        filename) throws IOException {
        ArrayList<String> list = new ArrayList<>();

        try (BufferedReader br = new BufferedReader(new
            FileReader(filename))) {
            String line;
            int i = 0;
            while ((line = br.readLine()) != null) {
                if(line.startsWith("0.0.0.0")) /* 0.0.0.0
                    web.domain.com */
                {
                    final String[] split = line.split("0.0.0.0");
                    list.add(split[1].trim());
                }
            }
        }

        return list;
    }

    /**
     * Costruisce un nuovo BloomBlocker a partire da una lista di
         url malevoli
     *  e la probabilità di falso positivo desiderata.
     *
     * @param hostnames la lista di host
     * @param pf la probabilità di falso positivo
     */
```

```
    public BloomBlocker(ArrayList<String> hostnames, float pf) {
        set = new BloomSet<>(hostnames.size(), pf);
        set.addAll(hostnames);
    }

    /**
     * Costruisce un nuovo BloomBlocker a partire da un file host
     *  e la probabilità di falso positivo desiderata.
     *
     * @param filename il nome del file host
     * @param pf la probabilità di falso positivo
     * @throws IOException se il file non esiste
     */
    public BloomBlocker(String filename, float pf) throws
        IOException {
        this(loadHostfile(filename), pf);
    }

    /**
     * Verifica la presenza di un dominio all'interno del file.
     *
     * @param domain il dominio da verificare
     * @return tt se è nel set con probabilità (1 - pf)
     *         ff se non è nel set co probabilità 1
     */
    public boolean checkDomain(String domain) {
        return set.contains(domain);
    }

    public int dimension() {
        return set.dimension();
    }


}
```

Listing 6: BloomHash.java

```
package hash;

import java.io.IOException;
import java.io.Serializable;

/**
 * @author Marco Costa
 */
public interface BloomHash {

    public int[] hashObject(Object o, int k, int bound);
```

```
    public int[] hashObject(String s, int k, int bound);
}
```

Listing 7: HashFactory.java

```
package hash;

/**
 * @author Marco Costa
 */
public class HashFactory {
    public static BloomHash getHashFunction() {
        return Murmur3_Hash.getInstance();
    }
}
```

## Test

Listing 8: ErrorTest.java

```
package Test;

import bloom.BloomBlocker;
import jdk.nashorn.internal.ir.debug.ObjectSizeCalculator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

/**
 * Test di falso positivo.
 * @author Marco Costa
 */
public class ErrorTest {

    private static final double bitToKB = (8 * 1024);

    /* funzione da iterare */
    private static int[] testFunction(ArrayList<String> hosts,
        ArrayList<String> domains, float pf) throws IOException {
        BloomBlocker blocker = new BloomBlocker(hosts, pf);
        int[] stats = new int[2];
        stats[0] = 0;

        for(String s : domains)
            if(blocker.checkDomain(s)) stats[0]++;
```

```
    stats[1] = blocker.dimension();
    return stats;
}


public static void main(String[] args) throws IOException {
    ArrayList<String> hosts = BloomBlocker.loadHostfile("hosts");
    ArrayList<String> safe_domains =
        Utils.parseFile("google_host");
    StringBuilder data = new StringBuilder();

    data.append("pf, errors, dim");
    data.append('\n');

    System.out.println("Elementi nel set: " + hosts.size());
    System.out.println("Elementi di test: " +
        safe_domains.size());
    System.out.println("************************************");
    for(int i = 6; i >= 2; i--)
    {
        float pf = new Double(Math.pow(10, -i)).floatValue();
        int[] result = testFunction(hosts, safe_domains, pf);
        double KBdimension = result[1] / bitToKB;

        System.out.println("PF: " + pf);
        System.out.println("Dimensione: " + KBdimension + " KB");
        System.out.println("Falsi positivi: " + result[0]);
        System.out.println("************************************");

        data.append((pf * 100) + ", " + result[0] + ", " +
            KBdimension + "\n");
    }

    Utils.printCSV("error_test_1hash_partial.csv", data);

    for(float pf = 0.1f; pf <= 0.9f; pf += 0.1f)
    {
        int[] result = testFunction(hosts, safe_domains, pf);
        double KBdimension = result[1] / bitToKB;

        System.out.println("PF: " + pf);
        System.out.println("Dimensione: " + KBdimension + " KB");
        System.out.println("Falsi positivi: " + result[0]);
        System.out.println("************************************");

        data.append((pf * 100) + ", " + result[0] + ", " +
            KBdimension + "\n");
    }
```

```
        Utils.printCSV("error_test_1hash_full.csv", data);

    }

}
```

Listing 9: SpaceTest.java

```
package Test;

import bloom.BloomBlocker;
import jdk.nashorn.internal.ir.debug.ObjectSizeCalculator;

import javax.rmi.CORBA.Util;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;

/**
 * Test di calcolo dimensioni in memoria.
 * @author Marco Costa
 */
public class SpaceTest {

    private static double[] testFunction(ArrayList<String> hosts,
        int n) throws IOException {
        HashSet<String> hashSet = new HashSet<>(n);
        LinkedList<String> arraySet = new LinkedList<>();

        double[] data = new double[2];

        for(int i = 0; i < n; i++)
        {
            String o = hosts.get(i);
            hashSet.add(o);
            arraySet.add(o);
        }

        data[0] = ObjectSizeCalculator.getObjectSize(hashSet) /
            1024.f; // Byte -> KB
        data[1] = ObjectSizeCalculator.getObjectSize(arraySet) /
            1024.f;

         return data;
    }

    public static void main(String[] args) throws IOException {
        ArrayList<String> hosts = BloomBlocker.loadHostfile("hosts");
```

```
        StringBuilder hashData = new StringBuilder();
        StringBuilder arrayData = new StringBuilder();

        hashData.append("n, dim\n");
        arrayData.append("n, dim\n");

        for(int i = 1; i < hosts.size(); i += 1000)
        {
            double[] KBsize = testFunction(hosts, i);

            hashData.append(i + ", " + KBsize[0] + "\n");
            arrayData.append(i + "," + KBsize[1] + "\n");
        }

        int size = hosts.size();
        double[] KBsize = testFunction(hosts, size);

        hashData.append(size + ", " + KBsize[0] + "\n");
        arrayData.append(size + ", " + KBsize[1] + "\n");

        Utils.printCSV("size_test_hash.csv", hashData);
        Utils.printCSV("size_test_array.csv", arrayData);
    }
}
```

Listing 10: BloomSpeedTestJMH.java

```
package Test;

import bloom.BloomBlocker;
import bloom.BloomSet;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.results.format.ResultFormatType;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.concurrent.TimeUnit;

/**
 * Benchmark JMH per BloomBlocker.
 * @author Marco Costa
 */
public class BloomSpeedTestJMH {
    private static final double pf = 0.0000001; // K = 24
```

```
@State(Scope.Benchmark)
public static class MyState {
    public String[] domains;
    public BloomSet<String> set;

    @Param({"1","2001","4001","6001","8001","10001","12001","14001","16331"})
    public int n;

    @Setup(Level.Trial) /* setup effettuato una volta */
    public void doSetup() throws IOException {
        ArrayList<String> hosts =
            BloomBlocker.loadHostfile("hosts");
        ArrayList<String> domainsList =
            Utils.parseFile("google_host");
        domains = new String[domainsList.size()];
        domains = domainsList.toArray(domains);

        set = new BloomSet<>(hosts.size(), pf);
        set.addAll(hosts);
    }

}

@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public boolean[] testFunction(MyState s) {

    boolean[] res = new boolean[s.domains.length];

    for(int i = 0; i < s.n; i++) {
        res[i] = s.set.contains(s.domains[i]);
    }

    return res; /* evita dead code */
}

public static void main(String[] args) throws RunnerException {
    Options opt = new OptionsBuilder()
            .include(BloomSpeedTestJMH.class.getSimpleName())
            .shouldDoGC(true)
            .resultFormat(ResultFormatType.CSV)
            .result("speed_test_bloom.csv")
            .forks(1)
            .build();

    new Runner(opt).run();
}
```

```
}
```

Listing 11: HashSetSpeedTestJMH.java

```
package Test;

import bloom.BloomBlocker;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.results.format.ResultFormatType;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.concurrent.TimeUnit;

/**
 * @author Marco Costa
 */
public class HashSetSpeedTestJMH {

    @State(Scope.Benchmark)
    public static class MyState {
        public String[] domains;
        public HashSet<String> set;

        @Param({"1","2001","4001","6001","8001","10001","12001","14001","16001","16331"})
        public int n;

        @Setup(Level.Trial)
        public void doSetup() throws IOException {
            ArrayList<String> hosts =
                BloomBlocker.loadHostfile("hosts");
            ArrayList<String> domainsList =
                Utils.parseFile("google_host");
            domains = new String[domainsList.size()];
            domains = domainsList.toArray(domains);

            set = new HashSet<>(hosts.size());
            set.addAll(hosts);
        }

    }
```

```
    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MILLISECONDS)
    public boolean[] testFunction(MyState s) {

        boolean[] res = new boolean[s.domains.length];

        for(int i = 0; i < s.n; i++) {
            res[i] = s.set.contains(s.domains[i]);
        }

        return res; /* evita dead code */
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(HashSetSpeedTestJMH.class.getSimpleName())
                .shouldDoGC(true)
                .resultFormat(ResultFormatType.CSV)
                .result("speed_test_hashset.csv")
                .forks(1)
                .build();

        new Runner(opt).run();
    }

}
```

Listing 12: LinkedListSpeedTestJMH.java

```
package Test;

import bloom.BloomBlocker;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.results.format.ResultFormatType;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.concurrent.TimeUnit;

/**
 * @author Marco Costa
 */
public class LinkedListSpeedTestJMH {
```

```
@State(Scope.Benchmark)
public static class MyState {
    public String[] domains;
    public LinkedList<String> set;

    @Param({"1","2001","4001","6001","8001","10001","12001","14001","16331"})
    public int n;

    @Setup(Level.Trial)
    public void doSetup() throws IOException {
        ArrayList<String> hosts =
            BloomBlocker.loadHostfile("hosts");
        ArrayList<String> domainsList =
            Utils.parseFile("google_host");
        domains = new String[domainsList.size()];
        domains = domainsList.toArray(domains);

        set = new LinkedList<>();
        set.addAll(hosts);
    }

}


@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public boolean[] testFunction(MyState s) {

    boolean[] res = new boolean[s.domains.length];

    for(int i = 0; i < s.n; i++) {
        res[i] = s.set.contains(s.domains[i]);
    }

    return res; /* evita dead code */
}

public static void main(String[] args) throws RunnerException {
    Options opt = new OptionsBuilder()
            .include(LinkedListSpeedTestJMH.class.getSimpleName())
            .shouldDoGC(true)
            .resultFormat(ResultFormatType.CSV)
            .result("speed_test_linkedlist.csv")
            .forks(1)
            .build();

    new Runner(opt).run();
```

```
    }

}
```

Listing 13: GuavaSpeedTestJMH.java

```
package Test;

import bloom.BloomBlocker;
import bloom.BloomSet;
import com.google.common.hash.BloomFilter;
import com.google.common.hash.Funnels;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.results.format.ResultFormatType;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.concurrent.TimeUnit;

/**
 * @author Marco Costa
 */
public class GuavaSpeedTestJMH {
    @State(Scope.Benchmark)
    public static class MyState {
        public String[] domains;
        public BloomFilter<String> set;

        @Param({"1","2001","4001","6001","8001","10001","12001","14001","16001","16331"})
        public int n;

        @Setup(Level.Trial)
        public void doSetup() throws IOException {
            ArrayList<String> hosts =
                BloomBlocker.loadHostfile("hosts");
            ArrayList<String> domainsList =
                Utils.parseFile("google_host");
            domains = new String[domainsList.size()];
            domains = domainsList.toArray(domains);

            set =
                BloomFilter.create(Funnels.stringFunnel(StandardCharsets.UTF_8),
                hosts.size(), 0.0000001);
            for(String s : hosts)
```

```
                set.put(s);
            }

        }

        @Benchmark
        @BenchmarkMode(Mode.AverageTime)
        @OutputTimeUnit(TimeUnit.MILLISECONDS)
        public boolean[] testFunction(MyState s) {

            boolean[] res = new boolean[s.domains.length];

            for(int i = 0; i < s.n; i++) {
                res[i] = s.set.mightContain(s.domains[i]);
            }

            return res; /* evita dead code */
        }

        public static void main(String[] args) throws RunnerException {
            Options opt = new OptionsBuilder()
                    .include(GuavaSpeedTestJMH.class.getSimpleName())
                    .shouldDoGC(true)
                    .resultFormat(ResultFormatType.CSV)
                    .result("speed_test_guava.csv")
                    .forks(1)
                    .build();

            new Runner(opt).run();
        }
    }
```