# Confronto tra funzioni hash

Rebecca Ceccatelli

giugno 2022

## 1 Introduzione

L'intento di questa relazione è confrontare l'efficienza di due metodi di calcolo della funzione hash di una tabella hash: il metodo delle divisioni e il metodo delle moltiplicazioni.

## 2 Nozioni teoriche fondamentali e descrizione del problema

Le tabelle ad indirizzamento diretto, dato un universo $—U—$ di chiavi, altro non sono che un array di $—U—$ locazioni in cui la chiave $k$ viene utilizzata come indice per indirizzare l'array: data la coppia *(v,k)*, il valore $v$ viene memorizzato direttamente nella locazione *T[k]*. Questo tipo di struttura dati è efficiente fintanto che l'universo $—U—$ delle chiavi è sufficientemente piccolo: se $—U—$ è troppo grande e, soprattutto, se è molto maggiore rispetto al numero di chiavi $|K|$ effettivamente presenti ($|U| >> |K|$), allora la tabella ad indirizzamento diretto occupa inutilmente troppo spazio.
Le tabelle hash sono strutture dati che nascono per risolvere questo problema. Infatti, data la coppia *(v,k)*, il valore $v$ viene memorizzato in una posizione generata da una funzione delle chiave $k$, non più direttamente nella posizione identificata dalla chiave stessa: non *T[k]* ma *T[h(k)]*. La funzione *h(k)* a cui viene delegato il calcolo dell'indice viene detta *funzione hash*, ed il suo fine ultimo è realizzare un hash uniforme semplice. La funzione hash non è una funzione iniettiva: è possibile che, date due chiavi $k$ e $k'$, con $k \neq k'$, accada *h(k) = h(k')*. Questo evento si chiama *collisione*.
Esistono vari metodi di calcolo delle funzioni hash. In questa relazione ci si concentra sul metodo delle divisioni e su quello delle moltiplicazioni.

Secondo il metodo delle divisioni, date una tabella hash di dimensione $m$ e una chiave $k$, la funzione hash è definita come il resto della divisione intera tra $k$ ed $m$.

$$h(k) = k\%m \tag{1}$$

Il vantaggio principale di questo metodo è la velocità di esecuzione, dato che consiste in una semplice divisione. Lo svantaggio è che il dimensionamento della tabella non è libero, perchè alcuni valori di $m$ devono essere evitati. Il problema deriva dal fatto che, se la base del sistema di numerazione scelto è $b$ e $m$ è una potenza intera di b scrivibile come $b^d$, allora la funzione hash è in grado di rilevare soltanto le $d$ cifre meno significative della chiave. Conseguentemente, tutte le chiavi che hanno le stesse $d$ cifre meno significative uguali vann a collidere tra loro.
Esempio. Siano $m = 100$ e $b = 10$. Allora, dato che $m = b^d$, si ha $100 = 10^2$ e perciò $d = 2$. Vengono rilevate soltanto le ultime 2 cifre meno significative, quindi chiavi come 99, 699 e 523423499 vanno tutte a collidere: $99\%100 = 99, 699\% = 100 = 99, 523423499\%100 = 99$.
La soluzione sarebbe scegliere $m$ come un numero primo non scomponibile, ma questo escluderebbe le potenze del 2, notoriamente predilette nei sistemi informatici.

Il metodo delle moltiplicazioni ovvia a questo problema. La funzione è definita come:

$$h(k) = \lfloor m(kA\%1) \rfloor \tag{2}$$

dove $k$ è la chiave di ingresso, $m$ è la dimensione della tabella, A è una costante casuale compresa tra $(0,1)$. Di fatto questo metodo di calcolo permette di generare una posizione casuale all'interno della tabella.
Lo svantaggio di questo metodo è che è leggermente più lento del precedente, in quanto i calcoli sono più laboriosi. Il grande vantaggio è che il valore di $m$ non è più critico per la funzione hash; anzi, la scelta di $m$ come potenza esatta del 2 è ideale, perchè si riesce a ricondurre il calcolo a moltiplicazioni tra interi gestite tramite l'operazione di shift.

# 3 Esperimenti

## 3.1 Scopo degli esperimenti

Lo scopo degli esperimenti che verranno successivamente illustrati è avvalorare la seguente tesi: sebbene il calcolo della funzione hash con il metodo delle moltiplicazioni richieda una complessità computazionale leggermente superiore, questo è da preferire al metodo delle divisioni perché mediamente riduce il numero delle collisioni all'interno della tabella e perché impone meno restrizioni sul dimensionamento $m$ della stessa.

## 3.2 Descrizione degli esperimenti e prestazioni attese

I test più significativi in questa relazione sono tre:

- Il primo test (Test1) vuole mettere a confronto, generando un grafico, la complessità temporale degli algoritmi di inserimento in una tabella hash

in cui la funzione hash è calcolata con il metodo delle divisioni e in una tabella hash in cui, invece, la funzione hash è calcolata con il metodo delle moltiplicazioni.

  - Prestazione attesa: ci si aspetta che la complessità temporale dell'algoritmo di inserimento in una tabella hash gestita con il metodo delle moltiplicazioni sia superiore rispetto a quella dell'algoritmo di inserimento in una tabella hash gestita con il metodo delle divisioni, a causa della maggiore laboriosità dei calcoli effettuati nel primo metodo.

- Il secondo test (Test2) vuole mettere a confronto, generando un grafico, il numero di collisioni avvenute in una tabella hash in cui la funzione hash è calcolata con il metodo delle divisioni e in una tabella hash in cui, invece, la funzione hash è calcolata con il metodo delle moltiplicazioni.

  - Prestazione attesa: : ci si aspetta che, al crescere del fattore di caricamento $\alpha$ di entrambe le tabelle, il numero di collisioni nella tabella hash gestita con il metodo delle moltiplicazioni sia significativamente minore rispetto a quello nella tabella hash gestita con il metodo delle divisioni.

- Il terzo test (Test3) vuole evidenziare, generando un grafico, il diverso tasso di collisione di una tabella hash gestita con il metodo delle divisioni e di una tabella hash gestita con il metodo delle moltiplicazioni, nel caso in cui entrambe ricevano in input una serie di chiavi multiple della dimensione $m$ della tabella.

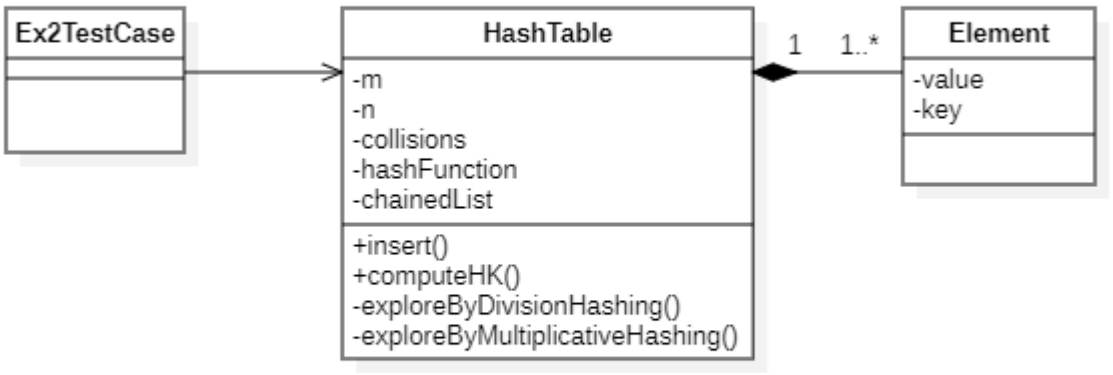
  - Prestazione attesa: dato che questa è la peggiore configurazione possibile di input per una tabella hash gestita con il metodo delle divisioni, ci si aspetta che l'incremento del numero di collisioni in questa tabella sia lineare; al contrario, si prevede che la tabella hash gestita con il metodo delle moltiplicazioni riesca a distribuire molto meglio le chiavi e che quindi mantenga un tasso di collisione basso.

## 3.3 Sviluppo del codice

Al fine di realizzare gli esperimenti precedentemente descritti è stato sviluppato del codice in Python per implementare le strutture dati interessate e per generare automaticamente un vasto campione di dati da utilizzare per tracciare i grafici tramite il pacchetto matplotlib. La piattaforma di test ha operato su un processore Intel a 2 GHz, su sistema operativo Windows.

# 4 Documentazione del codice

## 4.1 Diagramma di classe




## 4.2 Classi e unittest

- Classi:
  - **Element**: rappresenta l'elemento inserito in una **HashTable**. Custodisce la chiave *key* e il dato satellite *value*.
  - **HashTable**: implementa una tabella hash, e può essere gestita con una funzione hash secondo il metodo delle divisioni oppure secondo il metodo delle moltiplicazioni. Custodisce gli attributi *m, n, collisions, hashFunction* (una stringa che indica il metodo di calcolo scelto) e *chainedList* (una lista di $m$ liste di **Element**. Implementa i seguenti metodi:
    - *exploreByDivisionHashing(k)::int*: calcola e ritorna l'indice *h(k)* della lista a cui accedere nella tabelle secondo il metodo delle divisioni.
    - *exploreByMultiplicativeHashing(k)::int*: calcola e ritorna l'indice *h(k)* della lista a cui accedere nella tabella secondo il metodo delle moltiplicazioni.
    - *computeHK(k)::int*: a seconda del metodo di calcolo scelto nella taebella al momento della sua creazione, utilizza l'uno o l'altro

dei metodi precedenti per calcolare l'indice *h(k)* e rendere la scelta della funzione hash trasparente al cliente.

  - *insert(element)::void*: inserisce l'elemento *element* nella posizione corretta della tabella, avvalendosi del precedente metodo *computeHK()*.
  - *search(k)::Element,int*: cerca l'elemento con chiave $k$ all'interno della tabella e, se lo trova, ritorna l'elemento stesso e l'indice a cui si trova, avvalendosi del precedente metodo *computeHK()*.
  - *delete(element)::void*: rimuove l'elemento *element* dalla tabella, avvalendosi del precedente metodo *search()*.

- Unittest, file *Tests*: contiene i tests necessari per svolgere gli esperimenti e trarre le conclusioni sulla relazione.
  - *generateRandomKeys(n):[]*: helper method che ritorna un array di chiavi casuali, ognuna generata casualmente all'interno dell'intervallo *[0,n)*.
  - *testCompareCollisions()*: genera un grafico che mette a confronto il numero di collisioni ottenute inserendo una serie di chiavi casuali in due tabelle hash, gestite rispettivamente con il metodo delle divisioni e con il metodo delle moltiplicazioni.
  - *testDivisionHashingWorstCase()*: genera un grafico che mette a confronto il numero di collisioni in due tabelle hash, gestite rispettivamente con il metodo delle divisioni e con il metodo delle moltiplicazioni, quando la serie di chiavi inserite è multipla della dimensione $m$ della tabella. Questa configurazione di input corrisponde al peggior caso possibile per una tabella hash gestita con il metodo delle divisioni.
  - *computeMovingAverage(array)::[]*: helper method che calcola la media mobile dei dati nel vettore *array*.

# 5 Presentazione e analisi dei risultati sperimentali

## 5.1 Complessità temporale del metodo *insert()* in due tabelle hash gestite con i diversi metodi di calcolo

Il grafico in *figura 1* conferma che, preso uno stesso insieme di chiavi casuali da inserire in entrambe le tabelle, la complessità temporale dell'algoritmo di inserimento in una tabella hash gestita con il metodo delle moltiplicazioni è superiore a quella dell'algoritmo di inserimento in una tabella hash gestita con il metodo delle divisioni. Si osserva che i picchi e le irregolarità nel grafico sono da interpretare come rumore.

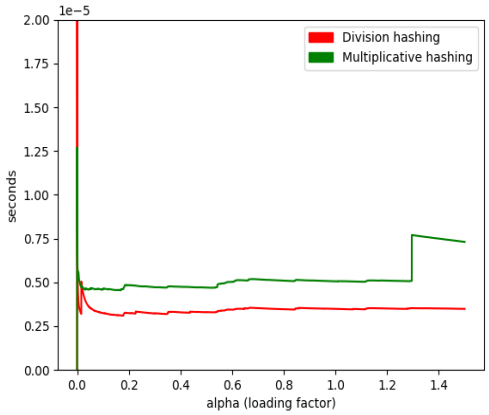



Figure 1: Complessità temporale del metodo *insert()* in una tabella hash gestita con il metodo delle divisioni e in una gestita con il metodo delle moltiplicazioni.

## 5.2 Numero di collisioni in due tabelle hash gestite con i diversi metodi di calcolo

Il grafico in *figura 2* conferma che, preso uno stesso insieme di chiavi casuali da inserire in entrambe le tabelle, la tabella hash gestita con il metodo delle moltiplicazioni ha un tasso di collisione inferiore rispetto a quella gestita con il metodo delle divisioni. Questa differenza è minima se il fattore di caricamento $\alpha$ è piccolo, ma si accentua sempre di più all'aumentare di $\alpha$.

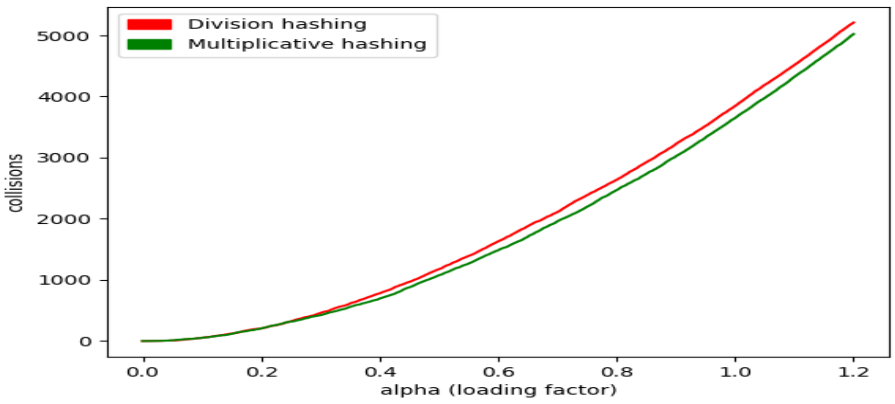



Figure 2: Paragone tra il numero di collisioni in una tabella hash gestita con il metodo della divisioni e in una gestita con il metodo delle moltiplicazioni.

### 5.3 Inserimento di una serie di chiavi multiple di $m$ in due tabelle gestite con i diversi metodi di calcolo

Il grafico in *figura 3* evidenzia uno dei vantaggi della tabella hash gestita con il metodo delle moltiplicazioni: la scelta del dimensionamento $m$ della tabella è molto più libera rispetto a quella in una tabella hash gestita con il metodo delle divisioni.
Infatti, il grafico mostra qual è il comportamento di quest'ultima nel caso peggiore possibile dal suo punto di vista: se l'input è un insieme di chiavi multiple di $m$, ogni chiave va a collidere nella stessa posizione delle altre e quindi ad ogni nuovo inserimento si verifica una collisione. Al contrario, la tabella hash gestita con il metodo delle moltiplicazioni ha un fattore casuale all'interno della sua funzione hash e quindi riesce a proteggersi meglio da configurazioni di input malevole, smistando le chiavi e riducendo conseguentemente il tasso di collisione.

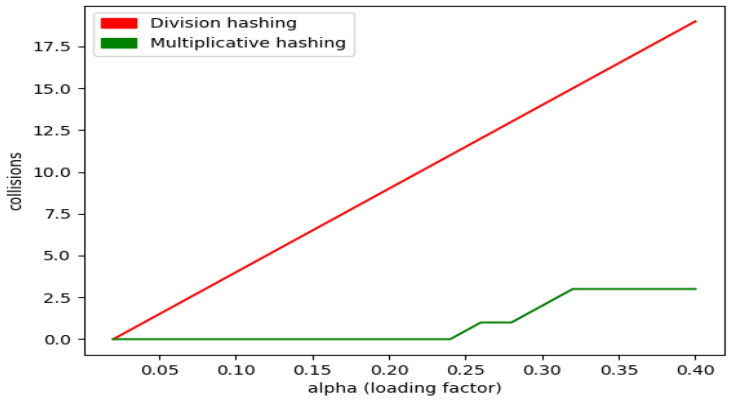



Figure 3: Inserimento di una serie di chiavi multiple di $m$ in una tabella hash gestita con il metodo delle divisioni e in una gestita con il metodo delle moltiplicazioni.

## 6 Conclusioni

Gli esperimenti confermano la tesi: è vero che il calcolo della funzione hash con il metodo delle moltiplicazioni richiede una complessità temporale leggermente superiore, ma questo è da preferire al metodo delle divisioni perchè mediamente riduce il numero di collisioni all'interno della tabella e perchè impone meno restrizioni sul dimensionamento $m$ della stessa.