Confronto Funzioni Hash

Niccolò Boanini

Indice

Sommario

Questo esercizio di Laboratorio ha come obiettivo quello di porre a confronto il metodo della divisione con il metodo della moltiplicazione per calcolare la funzione hash $h(k)$, utilizzando il concatenamento come regola per gestire le eventuali collisioni.

1 Metodi utilizzati per il calcolo di $h(k)$

1.1 Metodo della divisione

Il metodo della divisione permette di calcolare l'hash come il resto della divisione tra una chiave k e un determinato m che rappresenta la dimensione della tabella.

La funzione hash utilizzata è la seguente:

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

Nota: Risulta influente al fine di limitare le collisioni scegliere come m un numero primo non vicino a una potenza di 2.

1.2 Metodo della moltiplicazione

Il metodo della moltiplicazione calcola l'hash utilizzando la funzione qui di seguito:

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

In pratica viene estratta la parte frazionaria di kA (con $A \in (0;1)$ una costante da scegliere[1]), quindi viene moltiplicata per m e infine prelevata la parte intera inferiore.

Nota: a differenza di (1) in questo caso il valore di m non è critico. Ciò detto, tipicamente si pone $m = 2^p$, $p \in \mathbb{N}$ (ossia come potenza intera di 2).

2 Descrizione del lavoro svolto

Per completare l'esercizio sono stati effettuati diversi test volti a evidenziare il comportamento delle tabelle a seconda di specifici dati in ingresso.

L'approccio seguito in primo luogo è stato quello del *worst case analysis*, volto a studiare cioè l'evoluzione delle tabelle hash quando in ingresso vengono inseriti i dati considerati appunto *peggiori* per i relativi metodi.
Successivamente lo studio si è concentrato su una analisi più generale e casuale, con l'obiettivo di riassumere al meglio i vantaggi e gli svantaggi di ogni metodo quando in input si hanno dati non predicibili.
Osserveremo allo stesso tempo il progressivo riempimento della tabella man mano che cresce il *load factor* α.

2.1 Sviluppo e implementazione dei metodi

I test sono stati effettuati in `Python v3.10` utilizzando `Matplotlib` come libreria per la creazione dei grafici.
Al fine di confrontare al meglio i due metodi è stata creata una tabella apposita per ognuno: `T1` è riempita da oggetti della classe `User` attraverso il metodo della divisione; `T2` ospita i medesimi oggetti della classe `User` ma questi sono inseriti sfruttando il metodo della moltiplicazione[2].
Ogni oggetto è composto dall'abbinamento degli attributi `key` e `value`, dove quest'ultimo è un identificatore *human-readable*, cioè una stringa caratteristica dell'oggetto stesso (facoltativo ai fini degli esperimenti ma utile per la

[1] L'informatico statunitense Knuth raccomanda $A \approx \varphi^{-1} = (\sqrt{5}-1)/2 = 0,618033...$, ovvero l'inverso della *sezione aurea* [2]

[2] In alcuni esempi (come il 4.2), sono state utilizzate più di due tabelle per completare il confronto

comprensione e per possibili implementazioni più complesse).
Le principali funzioni del programma sono:

- `func_div(key)` e `func_mul(key)`: Funzioni hash
- `insert(User)`: Inserimento simultaneo di un utente in `T1` e `T2`
- `delete(User)`: Rimozione di un utente da `T1` e `T2`
- `searchT1(key)`: Ricerca chiave in `T1` (analoga per `T2`)
- `print_all()`: Visualizzazione su console delle tabelle, del *load factor* e del numero di collisioni rilevate su ciascuna tabella
- `plt.show()`: Visualizzazione dei grafici sulla GUI di `Matplotlib`

2.2 Ipotesi

Le ipotesi che assumeremo vere sono le seguenti:

i. Non possono esserci due `User` con la stessa chiave;

ii. Le collisioni sono gestite con il *concatenamento*;

iii. Le tabelle hanno tutte la stessa dimensione m.

3 Risultati attesi

Dalle implicazioni teoriche che abbiamo esposto in particolare nella sezione 1.1 ci aspettiamo che il metodo della divisione produca dei risultati negativi (ovvero numerose collisioni) quando m è una potenza intera di due oppure per come è definita $h(k)$ quando abbiamo tanti valori delle chiavi k che sono multipli di m.
Risulta invece difficile prevedere a priori l'esatta evoluzione della tabella quando si utilizza il metodo della moltiplicazione. Sappiamo però che qualsiasi valore di $A \in (0; 1)$ è sufficiente buono, anche se quello di Knuth sembra il più appropriato [1][2].

Per quanto riguarda in generale il *load factor*, ci aspettiamo che esso influisca sulle prestazioni quando diventa un valore superiore al 75% [3].

4 Risultati ottenuti

Di seguito vengono presentati e analizzati i test eseguiti, uno per volta.

4.1 Caso peggiore metodo divisione

Seguendo la filosofia *worst case analysis*, osserviamo il comportamento della tabella quando sono assegnati i valori critici per il metodo della divisione.

Impostiamo quindi: $m = 2^p$ (in questo caso $m = 16$) e tutte le chiavi k degli utenti `User` tra loro multiple di m, ovvero ad esempio $16, 32, 48, 64$ etc. Inseriamo in tutto 11 oggetti in tabella così da avere alla fine come *load factor* $\alpha = n/m \approx 70\%$.

Possiamo eseguire il tutto con il seguente *snippet* di codice che invocherà i metodi implementati:

```
key = m
for i in range(11):
    Hash.insert(User(key, string.ascii_uppercase[i]))
    key += m
```

In questo modo tutti gli oggetti hanno come chiave `key` un multiplo di m e come attributo `value` una lettera dell'alfabeto (da `"a"` fino a `"k"` essendoci in tutto 11 elementi).

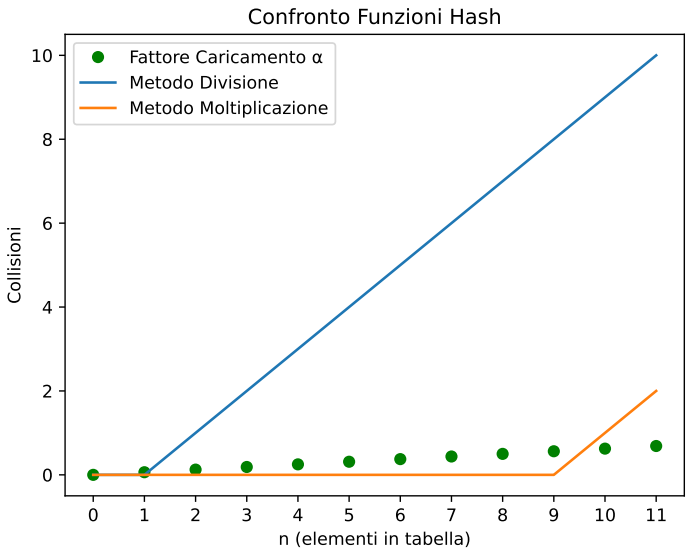

Figura 1: Caso peggiore metodo della divisione

Come si nota in Figura 1, il metodo della divisione risulta pessimo: dopo il primo elemento inserito, abbiamo tutte e sole collisioni.
Viceversa il metodo della moltiplicazione grazie alla sua funzione di hash apparentemente casuale riesce a gestire assai meglio la situazione, e si rilevano due sole collisioni rispettivamente quando si inserisce il decimo e l'undicesimo elemento. In Figura 2 è riportata la struttura grafica della tabella.

Nota: l'andamento delle collisioni con il metodo della moltiplicazione cresce in modo lineare (quindi critico) dopo che il *load factor* supera i valo-

Tabella T1 (metodo divisione):

0	→ K → J → I → H → G → F → E → D → C → B → A
1	→
2	→
3	→
4	→
5	→
6	→
7	→
8	→
9	→
10	→
11	→
12	→
13	→
14	→
15	→

Tabella T2 (metodo moltiplicazione):

0	→
1	→ H
2	→
3	→ G
4	→
5	→ F
6	→
7	→ E
8	→ D
9	→
10	→ C
11	→
12	→ K → B
13	→
14	→ J → A
15	→ I

Figura 2: Le due tabelle del caso 4.1 a confronto.

ri massimi di riferimento. Motivo per il quale in questi casi è consigliato ridimensionare la tabella, ad esempio raddoppiando il valore di m.

4.2 Caso peggiore metodo moltiplicazione (?)

Come già descritto nella sezione 1.2, il metodo della moltiplicazione soffre apparentemente di meno criticità. L'unico parametro che possiamo variare per capire se influisce o meno sulle prestazioni è A. Come si dimostra negli esempi successivi tuttavia, qualsiasi valore compreso tra 0 e 1 è sufficientemente buono, anche se basta distaccarsi di poco dal valore di Knuth[1] per ottenere risultati sensibilmente differenti.

Utilizzando gli stessi dati di 4.1, analizziamo le varie evoluzioni della tabella al variare di A. In particolare si pone:

$$\overbrace{A = (\sqrt{5}-1)/2 = 0,618033...}^{\text{Knuth}} \quad , \quad \overbrace{A = 0,848, \quad , \quad A = 0,126}^{\text{valori casuali } \in (0;1)}$$

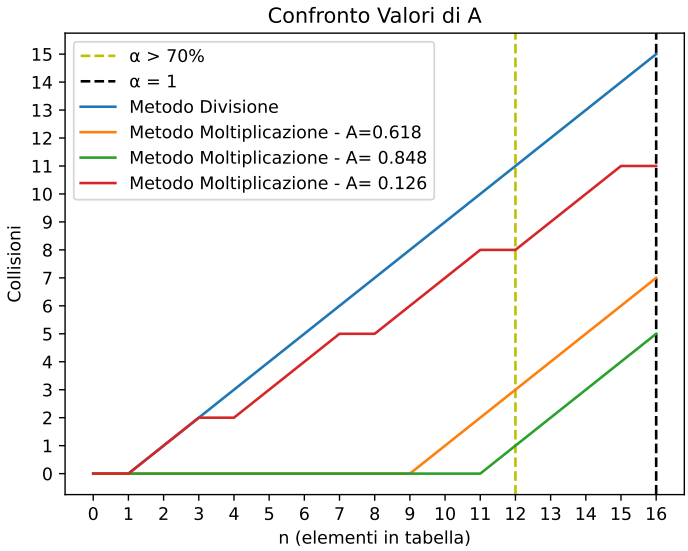

Figura 3: Evoluzione tabella al variare di A

Il risultato è riportato in Figura 3.
Innanzitutto come si nota, tutti e tre i valori utilizzati portano a un risultato migliore rispetto al metodo della divisione in termini di collisioni.
É poi interessante scoprire che ponendo $A = 0.126$ si hanno meno collisioni rispetto al caso $A = (\sqrt{5}-1)/2$. Questo perché il valore di Knuth è *probabilisticamente* migliore, cioè si comporta in generale sempre sufficientemente bene ma in casi specifici ci possono essere alternative migliori.

Pertanto complessivamente possiamo dire che non esiste un *caso peggiore* utilizzando il metodo della moltiplicazione, anche se certi valori di A risultano più o meno adatti a seconda dello scenario.

4.3 Caso randomico

Confrontiamo adesso i due metodi ponendo in ingresso numerosi valori pseudo-casuali e osserviamo come si comportano le tabelle di conseguenza.

In tutti e tre gli scenari, abbiamo $m = 1500$ e un universo delle chiavi pari a $m \cdot 20 = 30000$, quindi molto ampio. Per generare i numeri casuali, sono stati inseriti tutti i numeri da 1 fino a $m \cdot 20$ in una lista sfruttando la funzione `list(range(1,m*20))`, che è stata poi mescolata con la funzione `shuffle` della libreria `random`. Infine sono stati effettivamente inseriti in tabella i primi $m + 20$ elementi[3]. Lo snippet di riferimento è il seguente:

```
tot_ele = m+20
key_universe = list(range(1,m*20))
random.shuffle(key_universe)

for i in range(tot_ele):
        Hash.insert(User(key_universe[i]-1, "i"))
```

Nota: l'attributo `value` è semplicemente l'indice dell'elemento all'interno della lista (non ha molta importanza).

Passiamo dunque a commentare quanto ottenuto, che è riportato graficamente in Figura 4. Notiamo subito la cosa più importante: l'andamento in tutti e tre i (sotto)casi è *simile*: dei due metodi non ce n'è uno predominante. Entrambi causano all'incirca lo stesso numero di collisioni.
In dettaglio nel *Primo caso* il metodo della divisione appare leggermente svantaggiante, soprattutto dopo un certo elemento di valori inseriti.
Situazione sostanzialmente di "pareggio" invece nel *Secondo caso*, con i due metodi che in termini di collisioni praticamente si equivalgono.
Infine nel *Terzo caso* risulta lievemente meno performante il metodo della moltiplicazione, nonostante sia stato impostato ad A il valore di Knuth.

Importante sottolineare che se eseguiamo più e più volte il programma, ci ritroviamo sempre in uno dei tre contesti appena esposti, il che ci fa riflettere ancora una volta sull'efficacia e sulle differenze dei metodi. Il tutto sarà descritto completamente nella sezione successiva delle Conclusioni.

[3]Un valore a piacimento di poco superiore a m per vedere l'evoluzione della tabella fino (e oltre) $\alpha = 100\%$

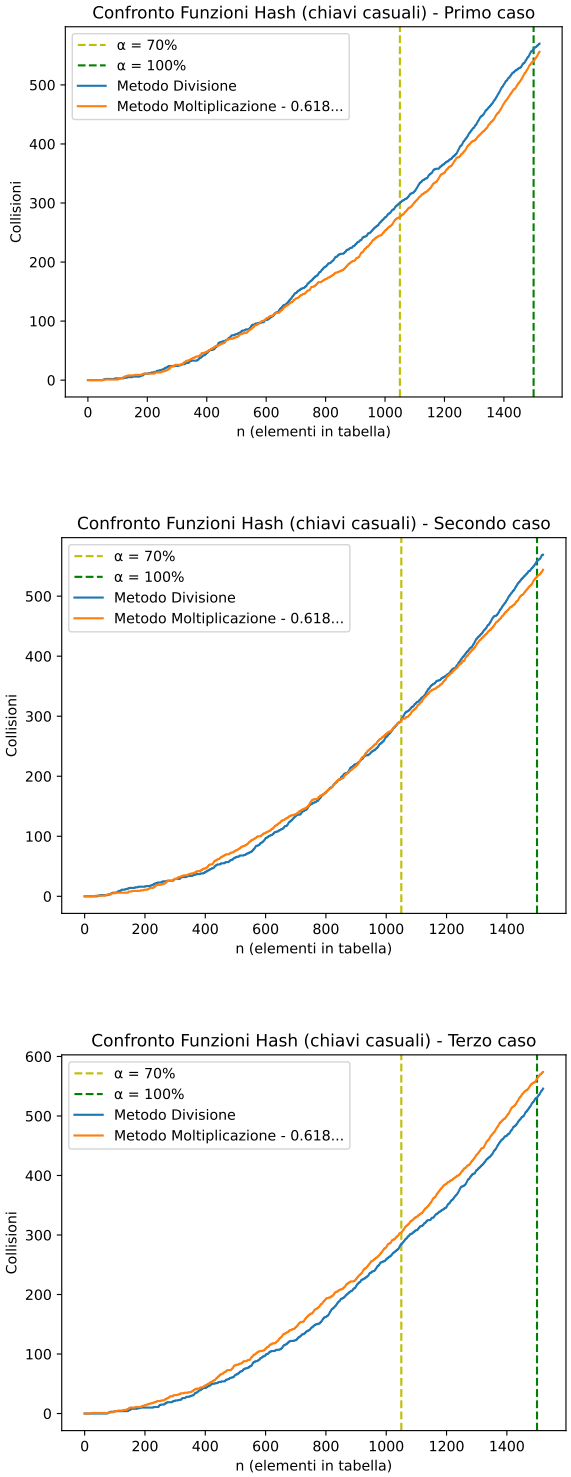

Figura 4: I tre casi a confronto

5 Conclusioni

Dagli esperimenti condotti abbiamo avuto come già detto la conferma di certe implicazioni teoriche. Se da una parte il metodo della divisione risulta più semplice da implementare, dall'altra appare vulnerabile a certi dati posti in ingresso. Alcuni di questi possono rendere alcune celle della tabella intasate in poco tempo e appesantire drasticamente l'esecuzione, come visto nell'esempio 4.1. Questo porta a fare riflessioni soprattutto nell'ambito della sicurezza informatica: un eventuale attacco dall'esterno potrebbe "dirottare" le chiavi degli elementi in ingresso verso i valori critici per il caso specifico, causando le spiacevoli conseguenze suddette.
D'altro canto il metodo della moltiplicazione non è critico rispetto al valore della dimensione della tabella, ma richiede tuttavia una scelta appropriata del valore di A. Inoltre l'operazione di moltiplicazione risulta più impegnativa per il calcolatore quando la dimensione della tabella non è una potenza di due ed è nettamente più conveniente l'operazione di divisione quando la tabella ha un numero primo come dimensione (viceversa se la dimensione è una potenza di due l'operazione di moltiplicazione equivale a un semplice AND bit a bit, quindi semplice e veloce in tal caso) [4].

Detto ciò, possiamo dire che su larga scala i due metodi si comportano sostanzialmente allo stesso modo, come abbiamo visto nel caso di Figura 4, pertanto a seconda delle esigenze si può scegliere l'uno o l'altro. Necessario però sottolineare che oggigiorno esistono metodi e funzioni più avanzate per ridurre le collisioni come l'hash perfetto.

Riferimenti bibliografici

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein, (2010): *Introduzione agli algoritmi e strutture dati*

[2] B. Preiss (1998): Data Structures and Algorithms with Object-Oriented Design Patterns - *Fibonacci Hashing*

[3] Wikipedia: *Hash Table*

[4] Stack Overflow: *What Are The Disadvantages Of Hashing Function Using Multiplication Method*