ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

APPROCCI COMPUTAZIONALI ALLA CONGETTURA DI COLLATZ

Elaborato in
Programmazione

Relatore
Pietro Di Lena

Presentata da
Nicola Cielo Lugaresi Secci

Anno Accademico 2019/2020

Introduzione

Cenni storici

La congettura di Collatz è una congettura matematica tuttora irrisolta ed è nota anche come congettura di Ulam, congettura di Syracuse o congettura 3n+1. Il problema ha probabilmente avuto origine da Lothar Collatz nel 1950, e si sarebbe diffuso oralmente tra vari matematici (tra cui si ricorda Helmut Hasse e Shizuo Kakutani). Inizialmente la sua popolarità è stata limitata: non appare in pubblicazioni fino agli anni '70, probabilmente perché in quel periodo si preferiva trattare problemi interconnessi riguardanti multiple aree matematiche e la Congettura appare in contrasto molto isolata. Inoltre era difficile fornire dimostrazioni interessanti studiando un problema di questo tipo e all'epoca sarebbe stato dannoso per la reputazione di un matematico pubblicare studi sul tema o anche solo mostrarvi interesse [11]. In effetti, data la sua difficoltà, pochi risultati degni di una pubblicazione sono stati raggiunti allo stato attuale. Solo dal 1971 il problema inizia ad apparire in pubblicazioni ufficiali, tra cui ricordiamo la trasposizione scritta di una lezione del 1970 di H. S. M. Coxeter [6] e gli appunti di Hasse per lezioni svolte nel 1975 [8]. Da questo punto la congettura è diventata via via più famosa, come esempio di problema estremamente semplice da presentare ma di difficoltà molto elevata. Stanislaw Ulam in particolare era interessato da questa tipologia di problemi e ha contribuito alla loro popolarizzazione, non a caso la Congettura di Collatz è anche nota come "problema di Ulam". In tempi più recenti i matematici Jeffrey Lagarias e Paul Erdős si sono espressi in questo modo sull'argomento: ([11])

> *"This is an extraordinarily difficult problem, completely out of reach of present day mathematics."* (Jeffrey Lagarias)
>
> *"Mathematics is not yet ready for such problems."* (Paul Erdős)

Nonostante l'assenza di una dimostrazione definitiva, la maggior parte dei matematici è concorde nel ritenere la congettura vera, e sono stati realizzati vari studi sul tema in ambiti diversi da quello puramente matematico.

Definizione del problema

La congettura è espressa come segue. Sia n un qualunque numero intero positivo, è definita così una funzione f:

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \\ 3n+1 & \text{se } n \text{ è dispari} \end{cases}$$

$$n \in \mathbb{N} \wedge n \geq 1$$

La funzione viene applicata iterativamente, creando una successione di valori detta sequenza di Hailstone[1] (o numeri di Hailstone). Il numero n viene posto come primo termine della sequenza; per ogni successivo elemento viene applicata la funzione usando come input il risultato del passo precedente:

$a_0 = n$
$a_1 = f(n)$
...
$a_{i+1} = f(a_i)$

Ad esempio, con $n = 7$:

$f(7) = 22$
$f(22) = 11$
$f(11) = 34$
$f(34) = 17$
...

La congettura di Collatz sostiene che è sempre possibile giungere in un numero finito di operazioni al termine $a_i = 1$ indipendentemente dal numero iniziale n scelto (o in altri termini: applicando ricorsivamente la funzione f a un qualunque numero intero positivo n si arriverà a 1 in un tempo finito).
Esempio di sequenza di Hailstone, con $n = 7$:
$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$
Si noti che è perfettamente possibile proseguire la sequenza, applicando ancora la funzione al numero 1. Alle successive iterazioni abbiamo

$f(1) = 4$
$f(4) = 2$
$f(2) = 1$

ossia un ciclo infinito (4; 2; 1), noto come "ciclo triviale" o "1-cycle". Una volta entrati nel ciclo triviale si incontrerà periodicamente il numero 1, dunque dato un numero di partenza n per cui la congettura è verificata esistono infiniti valori di i tali che $a_i = 1$. Assumendo che la congettura sia vera per

[1] Il nome deriva da Brian Hayes [9]: i numeri della sequenza appaiono salire e scendere ripetutamente fino a raggiungere 1, in modo simile alla grandine (in inglese: "hailstone") che sale e scende nelle nuvole temporalesche prima di precipitare al suolo.

ogni n intero positivo allora qualunque sequenza di Hailstone si conclude con il ciclo triviale.
Il "total stopping time" ("tempo di arresto") di n è definito come il più piccolo valore di i per cui $a_i = 1$, ossia il numero di iterazioni necessarie per raggiungere 1 per la prima volta[2]. Nel precedente esempio con $n = 7$ si ha un tempo di arresto pari a 16.
Qualunque numero con tempo di arresto finito rispetta la congettura: per promuovere la leggibilità del testo a seguire indicheremo come "verificato" un numero con tempo di arresto finito. Conseguentemente l'operazione di "verifica" è una qualunque sequenza di azioni volta a controllare se un numero rispetta la congettura o ne è controesempio. Abbrevieremo inoltre con "$TST(x)$" il tempo di arresto di un dato numero x.

Albero di Collatz

Possiamo ora fare una considerazione elementare: una volta dimostrato che il valore n iniziale giunge a 1 con TST finito, chiaramente anche i suoi iterati godranno della stessa proprietà. Infatti:

$$TST(n) = TST(f(n)) + 1$$

In generale ricorsivamente a_{i+1} è verificato se a_i è verificato (e viceversa). In altri termini: completare con successo l'operazione di verifica su un qualunque numero verifica contemporaneamente tutti gli iterati intermedi.
Possiamo fare un'ulteriore considerazione elementare, che appare evidente confrontando le sequenze di Hailstone di alcuni numeri (nell'esempio che segue utilizziamo come n iniziale i valori 6, 8 e 13).
6: 6, 3, 10, 5, 16, 8, 4, 2, 1
8: 8, 4, 2, 1
13: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
Gran parte delle sequenze sono ripetute, rendendo superfluo il loro ricalcolo. Per quanto sia una osservazione elementare ha implicazioni notevoli sul tipo di approccio che può essere adottato per verificare un numero: una quota considerevole dei calcoli necessari potrebbe essere ridondante.
Introduciamo ora il concetto dell'albero di Collatz: si tratta di un grafo orientato in cui ogni nodo corrisponde a un numero e il nodo "1" è posto alla radice, gli archi connettono i nodi che possono essere raggiunti attraverso la funzione f. La figura 1 mostra un grafo in esempio, rappresentante i nodi

[2]Non sarebbe corretto dire "il numero di iterazioni necessarie per raggiungere il ciclo triviale", poiché i tempi di arresto di 4 e 2 non sono nulli.

da 1 a 30 (27 escluso, per motivi che discuteremo in seguito) e i loro iterati intermedi. Eseguire la funzione di Collatz equivale a scendere nel grafo dal nodo di partenza fino alla radice. La rappresentazione in forma di grafo mostra in modo più chiaro il motivo della ripetizione delle sequenze.

Ci chiediamo ora se sia possibile esplorare il grafo in senso inverso, cioè salendo dalla radice. La funzione f non è iniettiva, poiché esistono coppie di valori distinti la cui immagine è la stessa (ad esempio $f(5) = f(32) = 16$), pertanto f non è invertibile. Questo non ci impedisce di elaborare comunque un processo per risalire l'albero, ma non sarà una funzione matematica.
Consideriamo un qualsiasi nodo dell'albero di Collatz ed assumiamo che sia l'immagine di un altro numero in f (in altri termini: assumiamo che esista almeno un nodo "padre" per il nodo scelto). Ricordiamo per riferimento la funzione f:

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \qquad (1) \\ 3n+1 & \text{se } n \text{ è dispari} \qquad (2) \end{cases}$$
$$n \in \mathbb{N} \wedge n \geq 1$$

Il nodo figlio potrebbe essere stato ottenuto attraverso la funzione (1) o (2) a seconda della parità del nodo padre. Non è un caso che nel grafo di esempio tutti i nodi hanno al massimo due nodi padre: i nodi che appaiono a una giunzione tra due rami sono quelli che possono essere raggiunti da due numeri diversi, seguendo rispettivamente (1) e (2). Osserviamo inoltre che tutti i nodi hanno almeno un nodo padre, poiché è sempre possibile percorrere a ritroso il percorso (1). Infatti, dato un numero $x \in \mathbb{N}$,

$$\exists!\ y \in \mathbb{N} \mid y = 2x$$

ed essendo y necessariamente pari si ha la sicurezza che $f(y) = x$ seguendo il caso (1). Esempio: 7 ha come padre $2 * 7 = 14$, infatti $f(14) = 7$.
Si noti che le foglie del grafo presentato ad esempio non sono realmente "foglie" terminali, è sempre possibile fare crescere l'albero in altezza aggiungendo ulteriori nodi padre.
Il caso (2) è leggermente più complesso. Il nodo padre dovrebbe avere valore $\frac{x-1}{3}$, e per appartenere a $\mathbb{N}$ è necessario che $(x-1)$ sia divisibile per 3. Questa condizione è necessaria ma non sufficiente, consideriamo ad esempio il numero 13: $\frac{13-1}{3} = 4 \in \mathbb{N}$, tuttavia è pari: $f(4) = 2 \neq 13$. Dunque $\frac{x-1}{3}$ deve essere dispari $\Rightarrow$ $(x-1)$ dispari $\Rightarrow$ x pari.

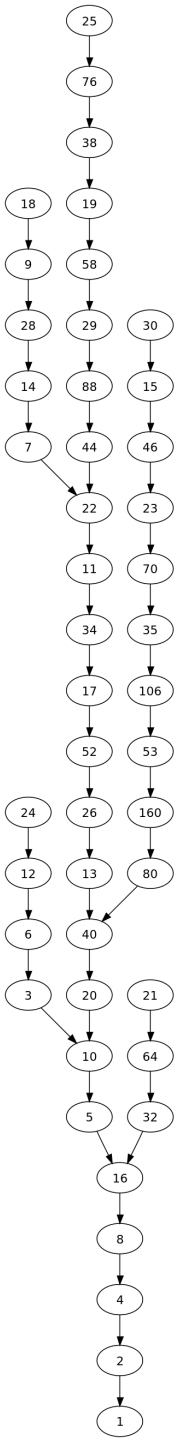


Figura 1: Albero di Collatz incompleto, ottenuto riportando in forma di grafo le sequenze di Hailstone dei primi 30 numeri interi (eccetto 27, per motivi di spazio).

Si può dimostrare che le due condizioni

$$(x-1) \bmod 3 = 0 \wedge x \bmod 2 = 0$$

sono equivalenti alla condizione

$$x \bmod 6 = 4$$

Riassumendo, un nodo x ha i seguenti nodi padre:

- $2x, \qquad \forall x \in \mathbb{N}$
- $\frac{x-1}{3}, \qquad x \in \mathbb{N} \wedge x \bmod 6 = 4$

Si noti che applicando queste regole a 4 si trovano i nodi-padre $\{8, 1\}$, portando come previsto al ciclo triviale.
A questo punto siamo in grado di costruire indefinitamente l'albero di Collatz a partire dalla radice 1, e il problema originario di "raggiungere 1 con un numero finito di operazioni partendo da un qualunque numero n" può essere riformulato come "raggiungere qualunque n partendo da 1 con un numero finito di operazioni". Questa formulazione alternativa presenta vantaggi e svantaggi rispetto a quella originaria, che verranno presentati in seguito. Indicheremo con "metodo diretto" le operazioni che seguono la formulazione originaria, percorrendo l'albero verso la radice. Analogamente il "metodo inverso" saranno operazioni secondo la formulazione alternativa, dalla radice verso le foglie.

Metodo diretto e metodo inverso

Osservazioni generali

Possiamo fare una serie di osservazioni per comprendere meglio l'andamento della funzione f e altri aspetti collegati alla congettura. Tali osservazioni saranno usate successivamente per semplificare i calcoli richiesti, aumentando in questo modo la velocità di esecuzione dei programmi.
Iniziamo con tre semplici osservazioni:

1. 2^a verificato $\forall\ a \in \mathbb{N}^{0+}$
2. $x > f(x)$ se x pari
3. $x < f(x) \wedge f(x)$ pari se x dispari

La prima osservazione è dimostrabile in modo semplice: nella sequenza di Hailstone il numero 2^a viene progressivamente dimezzato fino al raggiungimento di 1 senza mai incontrare un valore dispari, con tempo di arresto pari ad a.
Le due successive osservazioni sono altrettanto elementari e derivano dalla funzione f originaria. Possiamo interpretarle in modo intuitivo: incontrare valori pari nella sequenza di Hailstone ci "avvicina" al numero 1 (cioè porta a un andamento decrescente, l'iterato successivo sarà più piccolo), viceversa incontrare valori dispari ci "allontana" dall'obiettivo.
L'ultima osservazione ci porta a un ulteriore ragionamento: se l'iterato corrente x è dispari allora certamente $f(x) \neq 1$ (poiché pari, e comunque maggiore di x), dunque dovremo certamente proseguire con una successiva iterazione. Sappiamo già che $f(x)$ è pari, dunque possiamo concatenare due iterazioni consecutive in una versione "contratta" della funzione:

$$g(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \\ \dfrac{3n+1}{2} & \text{se } n \text{ è dispari} \end{cases}$$
$$n \in \mathbb{N} \wedge n \geq 1$$

Questa funzione alterata fornisce un piccolo vantaggio nella creazione di un metodo di verifica di un numero (poiché si "salta" un passaggio ogni volta che si incontra un numero dispari), notare però che questo altera il TST finale. Alcuni matematici hanno trovato questa forma più conveniente per trattare il problema ([7]), in particolare in campo probabilistico si preferisce la formulazione contratta.
Notiamo tuttavia che non possiamo similmente "contrarre" il metodo inverso dell'albero, poiché saltare nodi porterebbe alla creazione di un albero incompleto.

Considerazioni su controesempi e tempi di arresto

Questo elaborato non punta a dimostrare o confutare teoricamente la congettura, piuttosto prende in esame alcune strategie computazionali utilizzabili per cercare controesempi. Man mano che gli studiosi del tema estendono l'intervallo di valori verificati il costo computazionale aumenta a livelli onerosi anche per i processori più recenti, pertanto il tema dell'efficienza è estremamente importante.
Un controesempio per la congettura, se esistesse, apparterrebbe a una delle seguenti tipologie:

- Ciclo infinito non triviale: l'iterazione giunge a un ciclo infinito diverso dal noto $(4; 2; 1)$
- Traiettoria divergente infinita: l'iterazione continua ad avere un andamento crescente senza mai giungere a 1

Il caso della crescita infinita non può essere riconosciuto da un programma poiché richiederebbe tempo infinito per l'operazione di verifica. Il caso del ciclo non triviale sarebbe invece computabile, poiché il numero finito di elementi nel ciclo consentirebbe al programma di riconoscere la periodicità (per quanto il numero di elementi potrebbe essere tanto grande da superare la memoria disponibile).

Va fatta però una considerazione: tutti i numeri incontrati con il metodo inverso sono parte dell'albero e per questo automaticamente verificati[3]. Questo significa che la ricerca di un controesempio col metodo inverso fornirebbe solamente informazioni indirette (ovvero il fatto che *non* è presente un controesempio nell'intervallo considerato), ma come aspetto positivo non è necessario inserire in un algoritmo inverso controlli di questo tipo.
Parliamo ora più nel dettaglio dei tempi di arresto. Siano due numeri x e y tali che

$$x \in \mathbb{N}, y \in \mathbb{N}, y \gg x$$

Se la congettura fosse effettivamente valida e ogni numero convergesse a 1 dopo un numero finito di operazioni, ci si aspetterebbe a livello puramente intuitivo che il tempo di arresto di y sia maggiore di quello di x (o in altri termini, che un numero molto grande richieda un tempo superiore rispetto a un numero più piccolo). In realtà in molti casi questo non corrisponde al vero, si prenda come esempio la serie di tempi di arresto per i numeri da 1 a 30: ([3])

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
TST	0	1	7	2	5	8	16	3	19	6	14	9	9	17	17	4
N	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**		
TST	12	20	20	7	7	15	15	10	23	10	111	18	18	18		

Il tempo di arresto appare oscillare tra valori compresi nell'intervallo $0 - 23$, eccetto per quello del numero 27 che sale improvvisamente a 111.
Considerando i tempi di arresto per intervalli più grandi notiamo altri "picchi" relativi, via via più alti ma comunque senza un pattern immediatamente prevedibile che regola le oscillazioni. Il grafico in figura 3 raffigura i tempi di arresto dei primi 9999 numeri interi. Apparentemente non c'è alcuna correlazione tra il valore di N e il suo tempo di arresto.
Questo punto rende il problema particolarmente difficile da trattare: prendiamo come esempio il più elementare algoritmo utilizzabile per verificare un intervallo di valori.

```
per x in [a;b]:
   finché x!=1 ripeti:
      x=f(x)
```

[3]Ciò è corretto se l'albero viene costruito sempre a partire dalla radice 1. Se si applicassero le regole di creazione dell'albero a partire da un qualunque numero in $\mathbb{N}$ si potrebbe teoricamente incontrare un ciclo infinito, formando un grafo ciclico che sarebbe isolato da tutto il resto dell'albero. Tuttavia iniziare la costruzione dell'albero da un nodo arbitrario comporterebbe il rischio di non trovare mai il nodo cercato, poiché potrebbe essere in un diverso ramo.

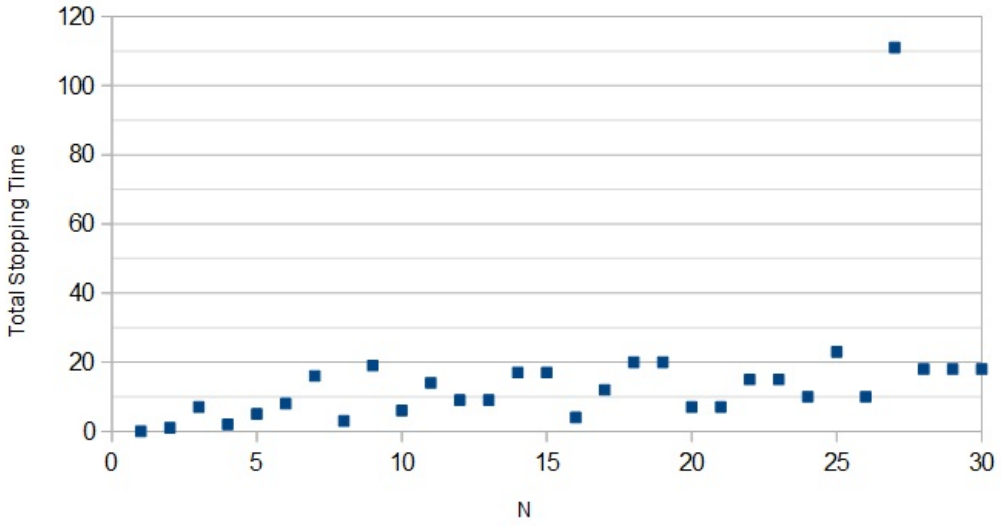


Figura 2: Grafico dei tempi di arresto con $N \leq 30$

Questo algoritmo esegue iterativamente le operazioni della congettura su ciascun elemento dell'intervallo fino ad arrivare a 1. Possiamo stimare come segue il tempo di esecuzione complessivo, assumendo che nessun elemento nell'intervallo abbia tempo di arresto infinito:

$$t \sum_{x=a}^{b} TST(x)$$

in cui t corrisponde al tempo medio necessario per compiere una singola iterazione. Come abbiamo visto però è molto difficile stimare euristicamente il TST senza eseguire le iterazioni della congettura: ad esempio potrebbe discostarsi di molto dalla media dei TST di numeri "vicini" (appartenenti a un "piccolo" intorno di quel numero).
Curiosamente l'andamento pseudorandomico del tempo di arresto può rendere vantaggioso l'uso di considerazioni di tipo probabilistico, pur trattandosi di un fenomeno deterministico. Queste considerazioni possono portare a stime relativamente accurate, ma conviene comunque tenere conto che il margine di errore potrebbe essere significativo. Similmente, se anche il consumo di memoria del programma fosse dipendente dal TST (es. in una implementazione del metodo inverso) potrebbe superare largamente le previsioni.

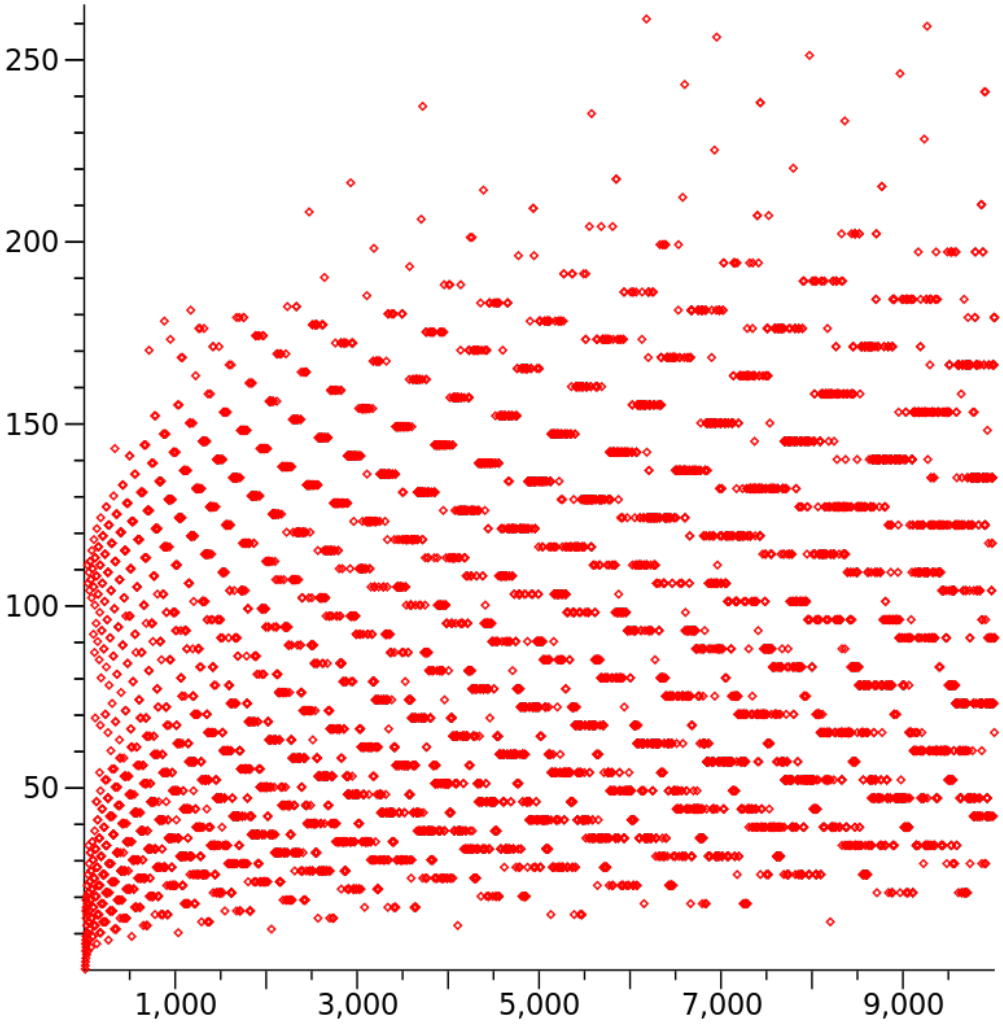


Figura 3: Grafico dei tempi di arresto con $N < 10^4$

Confronto tra metodo diretto e inverso

Abbiamo visto che seguendo il metodo diretto si giunge a 1 da n in un numero finito TST di operazioni, che nella rappresentazione ad albero equivale all'altezza del nodo di partenza (con altezza 0 alla radice).
Ci domandiamo ora quante operazioni richieda lo stesso compito nel metodo inverso. Appare intuitivo che questo numero dipenda da come viene fatto crescere l'albero prima di raggiungere il numero, ossia dalla politica adottata per costruirlo (procedendo in ampiezza, privilegiando determinati rami a discapito di altri...). Tuttavia è chiaro che la sequenza di nodi che collega 1 a n trovata costruendo l'albero è la stessa sequenza di Hailstone letta in ordine inverso: il metodo dell'albero non consente di raggiungere un numero in meno di TST operazioni.
Dunque per stimare il costo del metodo inverso scegliamo il numero $TST(x)$ come lower bound: il metodo inverso ha teoricamente lo stesso costo del metodo diretto, ma solo nel caso in cui sia possibile fare crescere l'albero da 1 scegliendo di volta in volta il ramo che porterà al numero cercato. In uno scenario reale non è però possibile conoscere in anticipo i rami ottimali, dunque si dovrà adottare una strategia più prudente costruendolo in "ampiezza" (cioè facendo crescere tutti i rami alla stessa altezza prima di proseguire al livello successivo). Ne consegue che in uno scenario reale il costo effettivo sarà sempre superiore al TST[4].
Sappiamo inoltre che ogni nodo del grafo ha uno o due nodi padre, dunque il numero di nodi che si incontrano a un certo livello sarà sempre compreso tra il numero di nodi del livello inferiore e il doppio di quest'ultimo. Non possiamo calcolare con esattezza il numero di nodi dell'albero dalla radice fino a una certa altezza a meno di costruirlo sperimentalmente, ma possiamo dire con sicurezza che sarà inferiore al numero di nodi che avrebbe un albero binario della stessa altezza. Dunque consideriamo come upper bound nella stima il valore $2^{TST+1} - 1$.
Sperimentalmente rileviamo che il numero di rami dell'albero appare crescere con un fattore di circa 1.264 (il fattore appare stabilizzarsi intorno al livello 25, si veda [2] e la Figura 4). Ne consegue che il grafo ha una crescita esponenziale, seppur certamente più lenta di quella di un albero binario.
Riassumendo, metodo diretto e inverso possono portare ugualmente alla veri-

[4]In verità questo vale "solo" per l'insieme $\mathbb{N} - \{1, 2, 4, 8, 16\}$, il motivo appare evidente osservando la Figura 1: quei valori appaiono nell'albero prima della prima biforcazione. Questo significa che per verificare questi casi col metodo diretto o inverso si visita la stessa sequenza di nodi in un senso o nell'altro, portando a un costo equivalente. Tali casi sono però estremamente triviali, e anche volendoli verificare sarebbe comunque da preferirsi il metodo diretto.

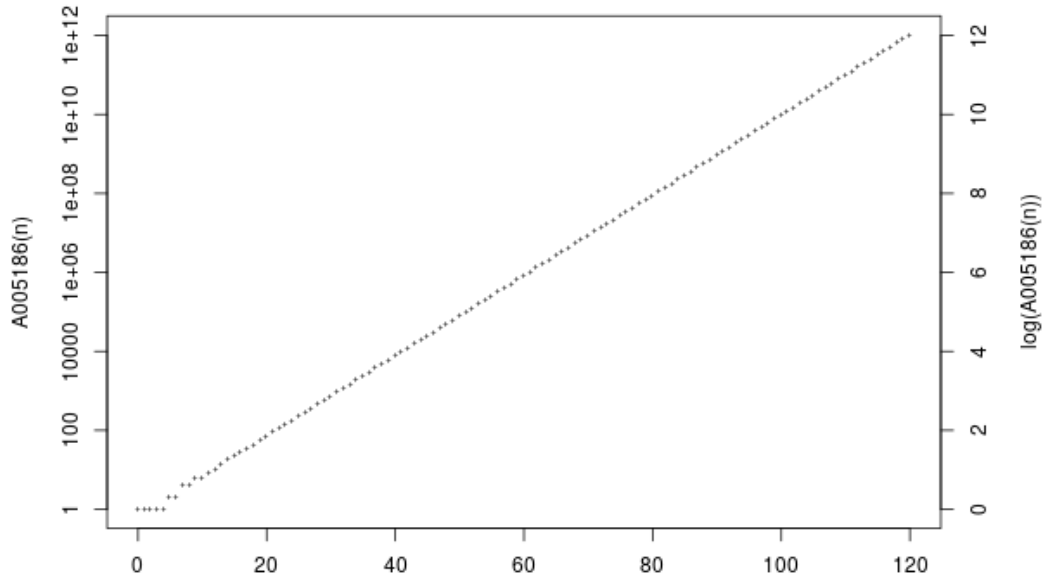


Figura 4: Grafico dell'ampiezza dell'albero di Collatz in funzione del livello, scala logaritmica (fonte: [2])

fica di un singolo numero, ma con costi estremamente diversi (crescita lineare in funzione del TST per il metodo diretto, crescita esponenziale per il metodo inverso). È chiaro che per verificare singoli valori è sempre preferibile utilizzare il metodo diretto, tuttavia il metodo inverso potrebbe essere vantaggioso per verificare allo stesso tempo più numeri in un intervallo.
La presenza di "picchi" di elevato TST è però un notevole ostacolo, anche se tali numeri rappresentano una percentuale molto piccola rispetto all'ampiezza totale dell'intervallo da verificare. Per via della crescita esponenziale aumentare la profondità dell'albero di soli 3 livelli causa il raddoppiamento del costo e conseguentemente dei tempi di esecuzione (stima effettuata assumendo che il fattore di crescita 1.264 sia costante a qualunque altezza). Dunque i picchi, per quanto rari, concorrono da soli alla quasi totalità del tempo speso in operazioni di verifica.
Se assumiamo che l'albero cresca costantemente con un fattore di 1.264 possiamo dire (con una certa approssimazione) che il numero di nodi fino all'altezza h sia circa 1.264^h.

Modelli probabilistici

Lo scopo di questa sezione è trattare di alcuni modelli che puntano a prevedere l'andamento delle traiettorie dei numeri lungo le sequenze di Hailstone. Una analisi approfondita in questo campo va al di là dello scopo del presente lavoro di tesi, ma può comunque portare spunti interessanti da utilizzare nella progettazione degli algoritmi.
Secondo alcuni modelli probabilistici le traiettorie dei numeri iterati con la funzione f originale incontrano numeri pari circa il 66% sul totale ([11]), tali modelli però sono più complessi di quelli che usano invece la versione alterata $g(x)$ e come anticipato in una precedente sezione si preferisce usare quest'ultima per questo tipo di analisi. Nel caso della versione modificata della funzione di Collatz i numeri pari e dispari appaiono in numero pressoché equivalente, come se risultassero da lanci di una moneta stocasticamente indipendenti[5]. Le successive osservazioni saranno fatte sottointendendo l'uso di $g(x)$ al posto della funzione originale.
I modelli probabilistici ([10] e [11]) suggeriscono l'esistenza di due diverse categorie di numeri, divisi a seconda della tipologia di traiettoria che assumono i loro iterati seguendo la funzione di Collatz. Questo fenomeno non è facilmente visibile per numeri con TST relativamente basso (inferiore a qualche centinaia), ma diventa più evidente graficando le traiettorie più lunghe usando un'asse verticale in scala logaritmica. I numeri appartenenti alla prima categoria seguirebbero una decrescita con una pendenza pressoché costante. I numeri appartenenti alla seconda categoria (ben più rara) seguirebbero invece un primo tratto di crescita fino a un picco, poi una decrescita altrettanto regolare. I due tipi di traiettorie sono rappresentati in Figura 5.
Secondo questi modelli la maggior parte dei numeri raggiunge 1 in circa $6.95\log(n)$ iterazioni, mentre i "picchi" richiedono $21.55\log(n)$ iterazioni. In generale è previsto che nessuna traiettoria richieda più di $41.68\log(n)$ iterazioni, suggerendo quindi che non esistano traiettorie divergenti.

Anche se si tratta di valori probabilistici che potrebbero essere in futuro confutati da prove empiriche sono comunque un utile punto di partenza nello studio di un algoritmo di verifica, poiché ci consente di stimare più agevolmente quali dovrebbero essere i tempi di arresto dei numeri presi in esame.
Occorre però ricordare che i valori sopra riportati si riferiscono alla funzione alterata $g(x)$, che porta a TST inferiori di circa il 33% rispetto a quelli

[5]Ricordiamo che nella formulazione iniziale del problema un numero dispari è seguito immediatamente da un numero pari, mentre la formulazione alternativa concatena questi due iterati in uno singolo. Se il numero di iterati pari e dispari è quasi uguale in una sequenza ottenuta con g allora la sequenza ottenuta con f dallo stesso numero di partenza avrà circa il 50% di numeri in più, portando all'atteso 66% di numeri pari.

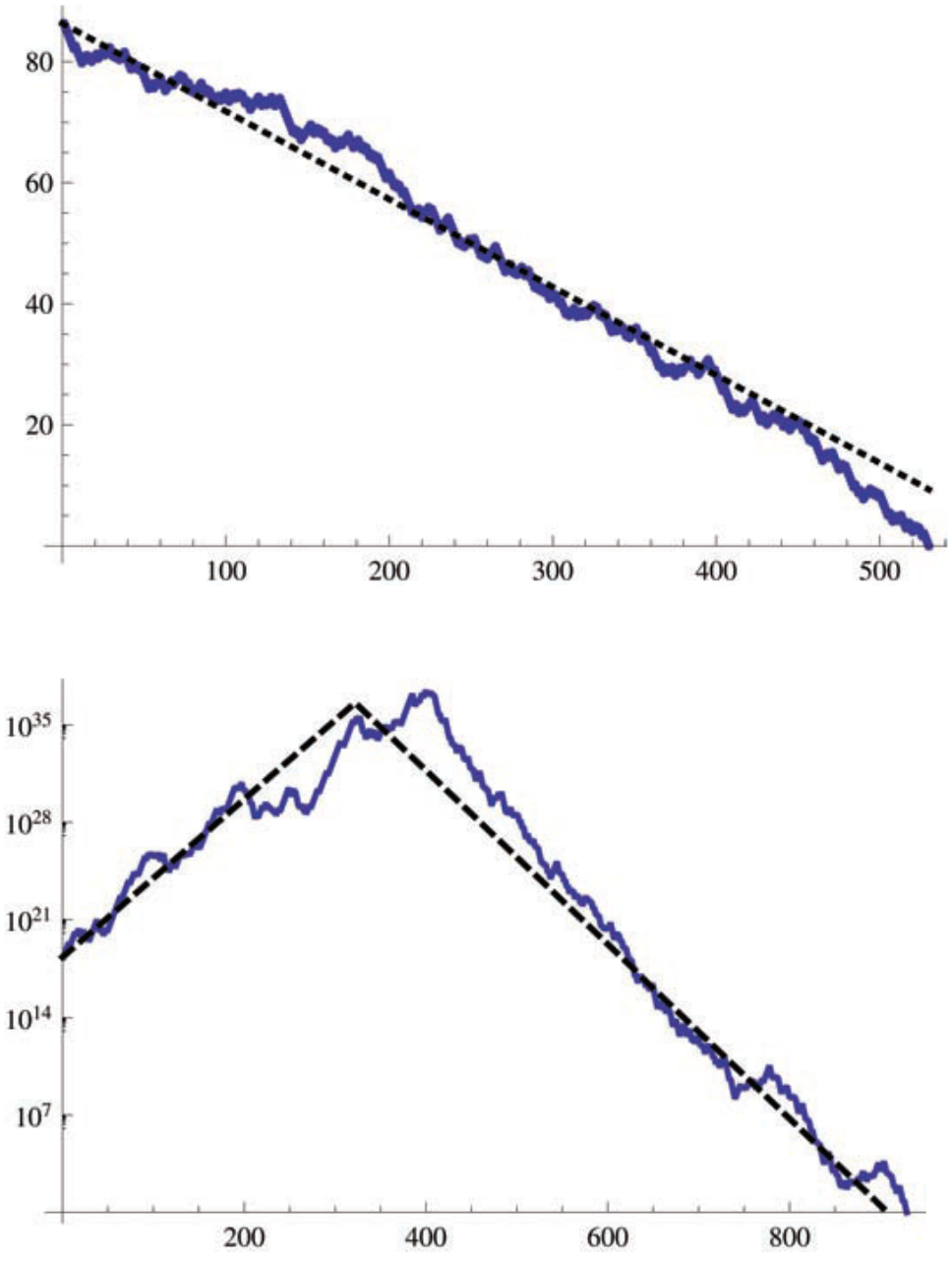


Figura 5: Due grafici presentati in [11]. Sopra: traiettoria di $n = 100\lfloor \pi * 10^{35} \rfloor$, sotto: traiettoria di $n = 1980976057694878447$. L'asse verticale è in scala logaritmica. La linea tratteggiata mostra l'andamento atteso secondo il modello probabilistico.

prodotti con $f(x)$. Il metodo inverso non può permettersi di "saltare" iterati intermedi poiché rischierebbe di perdere valori utili, pertanto la stima va aumentata del 50%. Dunque:

$$TST(x) \approx \begin{cases} 32.33 * \log(x) & \text{se } x \text{ è un picco} \\ 10.43 * \log(x) & \text{altrimenti} \end{cases} \leq 62.52 * \log(x) \quad \text{in generale}$$

Sono comunque da effettuare considerazioni sull'uso di memoria del programma, tali punti saranno trattati successivamente.

Automi cellulari

Motivazioni dell'approccio con CA

Uno dei problemi affrontati per sviluppare un programma di verifica riguarda il modo in cui i numeri sono memorizzati all'interno del computer, e come strategie diverse possono influenzare il tipo e il numero di operazioni che saranno effettivamente svolte dalla macchina per iterare Collatz. In questo capitolo spiegheremo cosa sono gli Automi Cellulari e perché possono essere utili nelle operazioni di verifica della congettura, oltre a presentarne alcuni (tratti dalla letteratura sul tema) progettati specificamente per questo scopo.
Un automa cellulare (CA) è un modello di computazione composto da una griglia di celle n-dimensionale (spesso bidimensionale, minimo una dimensione). Ogni cella della griglia assume un particolare stato, ad esempio uno tra i vari possibili potrebbe rappresentare una "cella vuota" mentre i rimanenti potrebbero indicare le condizioni di una "cella piena". La griglia viene popolata assegnando lo stato di ogni cella, ciò corrisponde alla configurazione iniziale del CA. Per passare all'istante di tempo successivo della computazione (una "generazione" successiva) viene applicato iterativamente su ogni cella un set di regole. Queste regole prendono in considerazione lo stato delle "celle vicine" e della cella corrente per decidere lo stato che assumerà quest'ultima nella generazione seguente. Il concetto di "celle vicine" è variabile, ogni CA potrebbe considerare come suoi "vicini" un insieme di celle diverso.
Molto famoso è il "Game of Life" sviluppato da John Conway nel 1970. Ogni cella della griglia bidimensionale può essere "viva" o "morta" e considera come suoi vicini le otto celle circostanti (ortogonalmente e diagonalmente). Le regole del CA causano "nascita" di cellule (cioè passaggio di stato di una cella da morto a vivo) quando hanno tre vicini vivi, mentre la "morte" di una cella può essere causata da un eccesso di vicini (quattro o più celle vive) o una carenza (una cella viva o nessuna).
Usare nel programma di verifica una normale variabile per numeri interi (es. *int* da 4 byte in linguaggio C) per contenere un iterato della congettura è una scelta più che ragionevole fintanto che si resta entro i limiti di tale variabile,

ma per rendere possibili calcoli con numeri ancora più grandi occorre usare variabili più capienti (es. *long* da 8 byte) al prezzo di una minore efficienza nei calcoli del processore. Se anche questo non fosse sufficiente occorrerebbe usare metodi alternativi per archiviare il numero, creando strutture dati apposite che lo memorizzino in aree di memoria arbitrariamente grandi. Esistono varie librerie che offrono questa funzione oltre a implementare varie operazioni aritmetiche tra numeri espressi in tale forma, visto che anche le più elementari come la somma non possono essere eseguite da una singola operazione del processore. Possiamo esprimere un iterato della congettura nello spazio di un CA e scegliere un set di regole tale da replicare il comportamento della funzione di Collatz.

Automi Cellulari nella letteratura

Solitamente i Cellular Automata che effettuano le iterazioni della congettura richiedono che il numero sia espresso in una particolare base, ogni cifra è memorizzata in una cella del sistema attraverso il suo stato. Ogni base presenta pro e contro, poiché certe operazioni matematiche possono essere più o meno agevolate da questa scelta.
Nella letteratura sono descritti numerosi automi cellulari, la presente lista non ambisce ad essere esaustiva né a illustrare nel dettaglio il loro funzionamento. Invitiamo a consultare le fonti riportate qualora interessassero le dimostrazioni e le regole dei vari automi.

CA in base 3

Quando un numero è espresso in base 3 per determinarne la parità occorre sommare insieme le sue cifre e verificare se il risultato è pari o dispari.
Un automa cellulare proposto da Sitan Chen ([5]) opera su una griglia tridimensionale con due livelli: le celle nel livello inferiore contengono le cifre degli iterati e possono assumere tre stati diversi (corrispondenti chiaramente ai tre valori che assume una cifra in base 3). Il livello superiore è invece usato per sommare insieme le cifre del livello inferiore e determinarne la parità, le celle in questo livello possono assumere quattro stati diversi. Questo automa produce iterati secondo la funzione "compressa" $g(x)$: incontrare un numero dispari concatena in una singola iterazione anche la successiva divisione per due, portando ad un TST di circa 30% inferiore.
Un altro automa che opera in base 3 è stato proposto da Mario Bruschi ([4]). Questo CA è monodimensionale, cioè esiste una singola riga di celle invece di una griglia multidimensionale. Le celle possono assumere quattro valori

diversi; l'autore suggerisce che l'automa sia molto veloce da calcolare anche su una macchina di tipo sequenziale, data la struttura lineare dei dati.

CA in base 4

La parità di un numero espresso in base 4 è più semplice da determinare, poiché è sufficiente controllare la parità dell'ultima cifra del numero. Inoltre i numeri che terminano con una sequenza di "0" sono certamente multipli di 4, pertanto corrispondono nella funzione di Collatz originaria a una sequenza di almeno due numeri pari consecutivi. Troncare gli "0" terminali equivale ad eseguire un certo numero di divisioni per quattro in una singola iterazione, portando alla convergenza della traiettoria in un numero di operazioni inferiore al valore di TST. Purtroppo questo grande vantaggio è anche il motivo per cui non possiamo usare questo tipo di CA per il calcolo degli iterati col metodo inverso: come abbiamo visto in precedenza non possiamo "saltare" elementi della sequenza di hailstone poiché rischieremmo di ignorare iterati intermedi (che potrebbero appartenere all'intervallo che occorre verificare) o addirittura interi rami dell'albero.
Chen in [5] propone un CA che opera in base 4 su una griglia bidimensionale, ogni cella può assumere 5 stati (quattro di questi stati hanno inoltre un attributo "pari" o "dispari", portando a un totale di 9 stati distinti). Secondo le sue stime, questo processo dovrebbe ridurre il numero di iterazioni complessive di circa il 36%.

CA in base 2

Come per la base 4, la base 2 porta benefici in termini di un controllo di parità semplificato e offre la possibilità di dividere più volte per due nella stessa iterazione semplicemente eliminando gli "0" terminali. Se non importa riconoscere i singoli iterati ma solo raggiungere 1 il più rapidamente possibile (come nel caso del metodo diretto), CA di questo tipo dovrebbero fornire prestazioni ottime rispetto ad altre basi.
Uno svantaggio derivante dall'adottare la base 2 (la più piccola possibile) deriva dal fatto che i numeri in base 2 hanno naturalmente più cifre di quelle dei numeri espressi in basi più grandi: il numero di iterazioni totali si riduce, ma aumenta il numero di operazioni per ogni iterazione.
Bruschi in [4] propone un altro CA monodimensionale le cui celle assumono valori 0 o 1. Anche in questo caso la computazione avviene scorrendo le caselle sequenzialmente e "segnandone" alcune.
Chen in [5] propone invece un CA bidimensionale con celle a tre stati diver-

si. In media poter eseguire in singola iterazione tutte le divisioni per due consecutive porta a un numero totale di iterazioni nettamente inferiore al TST della normale $f(x)$: una riduzione di quasi il 70%. L'autore fa menzione inoltre di come applicare parallelismo per migliorare ulteriormente le prestazioni.

CA in base 6

Nella realizzazione dell'algoritmo abbiamo infine scelto di utilizzare un CA che opera in base 6, descritto da Stephen Wolfram ([13]) ed altri studiosi. Questo automa ha la particolarità di non essere stato creato esplicitamente per la computazione degli iterati della Congettura, bensì semplicemente per la moltiplicazione per 3. In seguito lo indicheremo con $CA_{\times 3}$
Per utilizzare questo CA si procede come segue. Per prima cosa occorre rappresentare il numero iniziale in base 6 e inserire le varie cifre in una griglia bidimensionale. Per chiarezza assumiamo che la griglia sia rappresentata con coordinate crescenti da sinistra verso destra e dall'alto verso il basso, dunque una riga di indice più grande si trova "sotto". Inseriamo le cifre del numero di partenza in celle consecutive di una riga i, le colonne con indice più piccolo conservano le cifre più significative del numero. In altri termini: inseriamo il numero partendo dalle cifre più significative e proseguendo da destra verso sinistra sulla stessa riga, con indici di colonna crescenti. Dato che la griglia contiene cifre di un numero in base 6 in ogni casella possiamo dire che l'automa opera con celle a sei stati, ciascuno dei quali è rappresentato da una cifra da 0 a 5. Il resto delle celle della griglia può inizialmente essere inizializzato col valore 0; si può aggiungere un settimo stato "nullo" per indicare la fine del numero, ma nell'implementazione che abbiamo adottato abbiamo mantenuto 6 stati e distinto inizio e fine del numero memorizzando in variabili ausiliarie un riferimento a tali celle.
Il numero così espresso funge da configurazione iniziale per $CA_{\times 3}$. Se indichiamo con "(i;j)" una casella alla riga i e colonna j, allora i suoi "vicini" saranno "(i-1;j)" e "(i-1;j+1)" (cioè la casella sovrastante e quella a destra di quest'ultima). Le regole dell'automa possono essere riassunte nella seguente tabella:

T_1:

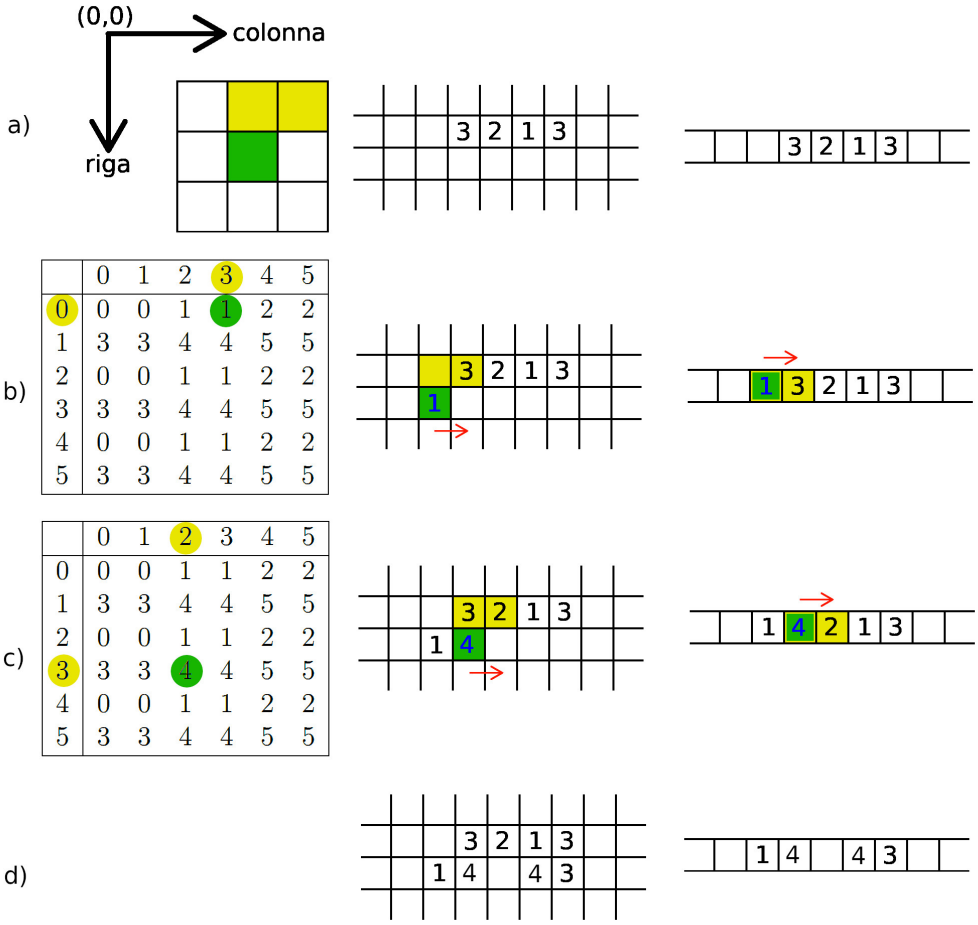
	0	1	2	3	4	5
0	0	0	1	1	2	2
1	3	3	4	4	5	5
2	0	0	1	1	2	2
3	3	3	4	4	5	5
4	0	0	1	1	2	2
5	3	3	4	4	5	5

$$(i;j) = T_1[(i-1;j)][(i-1;j+1)]$$

In altri termini: la casella (i;j) assume il valore che si trova nella tabella, scegliendo la riga pari al valore della cella sovrastante e la colonna pari al valore della cella in alto a destra. Ogni nuova generazione di $CA_{\times 3}$ inserisce una nuova riga di valori non-nulli sotto alle precedenti (che restano inalterate), l'ultima riga è il valore che il numero aveva alla precedente generazione triplicato. Assumendo che la griglia si estenda all'infinito in tutte le direzioni, dato che non sono effettuate modifiche alle righe precedenti si avrà sempre uno "storico" delle generazioni.
Anche se $CA_{\times 3}$ prevede l'utilizzo di una griglia bidimensionale è perfettamente possibile usare solamente due righe: una è usata per contenere la generazione attuale, l'altra è usata per memorizzare la nuova generazione; una volta finito il calcolo della nuova generazione si scambiano i ruoli e si ricomincia. Osserviamo però che i vicini della cella correntemente considerata sono sempre la cella corrispondente nella riga precedente e la sua vicina di destra. Questa caratteristica ci consente di utilizzare una singola riga per computare il CA, a patto di seguire un ordine specifico di esecuzione: occorre aggiornare le celle partendo da quella più significativa verso la meno significativa (indici crescenti).

```
per ogni x in 0...N:
   vettore[x]=T1[vettore[x]][vettore[x+1]]
```

L'implementazione di questo CA è triviale, ma lo spazio messo a disposizione nella memoria del computer sarà senza dubbio finito. Occorrono pertanto una serie di strategie che garantiscano che la computazione non esca dai limiti dello spazio.
In figura 6 è rappresentato un esempio pratico del funzionamento di $CA_{\times 3}$.
In che modo un CA che ha l'unico effetto di moltiplicare per tre può essere usato per calcolare gli iterati di Collatz? In primo luogo osserviamo che triplicando un numero in base 6 otteniamo sempre un numero che termina con 0 quando il numero di partenza era pari, un numero che termina con 3 se invece era dispari. In secondo luogo osserviamo che lo shift di un numero in base 6 a destra o sinistra equivale a dividere o moltiplicare quello stesso numero per 6. Sulla base di questi elementi possiamo sviluppare un semplice algoritmo che calcola $f(x)$ usando $CA_{\times 3}$:



	0	1	2	3	4	5
0	0	0	1	1	2	2
1	3	3	4	4	5	5
2	0	0	1	1	2	2
3	3	3	4	4	5	5
4	0	0	1	1	2	2
5	3	3	4	4	5	5

	0	1	2	3	4	5
0	0	0	1	1	2	2
1	3	3	4	4	5	5
2	0	0	1	1	2	2
3	3	3	4	4	5	5
4	0	0	1	1	2	2
5	3	3	4	4	5	5

Figura 6: In questo esempio viene triplicato il numero 729 (”3213” in base 6) usando $CA_{\times 3}$. Nei passaggi sono evidenziati in giallo i vicini della casella correntemente considerata (a sua volta colorata di verde). Nel punto *a)* è presentata la configurazione iniziale in uno spazio bidimensionale e la stessa configurazione nella variante monodimensionale. Applicando le regole in ordine di colonna crescente (richiesto dalla versione monodimensionale) in *b)* viene incontrata la prima casella della nuova generazione che assumerà valore non-nullo: si consulta la tabella alle coordinate indicate dalle celle vicine ”0” e ”3” portando al risultato ”1” che viene scritto nella casella corrente (indicato in blu). Similmente in *c)* viene assegnato ”4” alla successiva cella, dati i vicini ”3” e ”2”. La computazione prosegue fino al risultato finale, presentato in *d)*. La variante monodimensionale comporta la perdita di valori della generazione precedente, ma la sovrascrittura avviene solo quando tali valori non sono più necessari per la computazione.

1. Applica $CA_{\times 3}$
   $x \Rightarrow 3x$

2. Controlla l'ultima cifra del numero:

   (a) Se termina con 0: shift a destra del numero
   $3x \Rightarrow 3x/6 \Rightarrow x/2$

   (b) Se termina con 3: imposta l'ultima cifra a 4
   $3x \Rightarrow 3x + 1$

Intuitivamente occorrono TST generazioni per raggiungere 1.
$CA_{\times 3}$ è reversibile: ogni configurazione di stati può essere ottenuta da una singola configurazione. Pertanto possiamo definire $CA_{\div 3}$ per eseguire la divisione per 3 in base 6.

T_2:

	0	1	2	3	4	5
0	0	2	4	0	2	4
1	0	2	4	0	2	4
2	0	2	4	0	2	4
3	1	3	5	1	3	5
4	1	3	5	1	3	5
5	1	3	5	1	3	5

$$(i; j) = T_2[(i-1; j)][(i-1; j-1)]$$

Osserviamo come in questo caso i vicini di (i;j) sono rispettivamente la casella sovrastante e quella a sinistra di quest'ultima. Con un ragionamento analogo al precedente, possiamo permetterci di usare una singola riga per computare $CA_{\div 3}$ se iniziamo sovrascrivendo le cifre meno significative (quindi partendo da colonne di indice maggiore e decrementando).

```
per ogni x in N...0:
   vettore[x]=T2[vettore[x]][vettore[x-1]]
```

In Figura 7 è rappresentato un esempio pratico del funzionamento di $CA_{\div 3}$.

Possiamo usare $CA_{\div 3}$ per costruire un semplice algoritmo per calcolare l'albero di Collatz, tenendo però presente che la funzione di Collatz non è invertibile: ad esempio il numero 16 dovrebbe produrre due diverse configurazioni 5 e 32. Ricordiamo un concetto dimostrato nel capitolo introduttivo, cioè che un nodo x dell'albero di Collatz è raggiunto da uno o due altri nodi:

- $2x, \qquad \forall x \in \mathbb{N}$
- $\frac{x-1}{3}, \qquad x \in \mathbb{N} \wedge x \bmod 6 = 4$

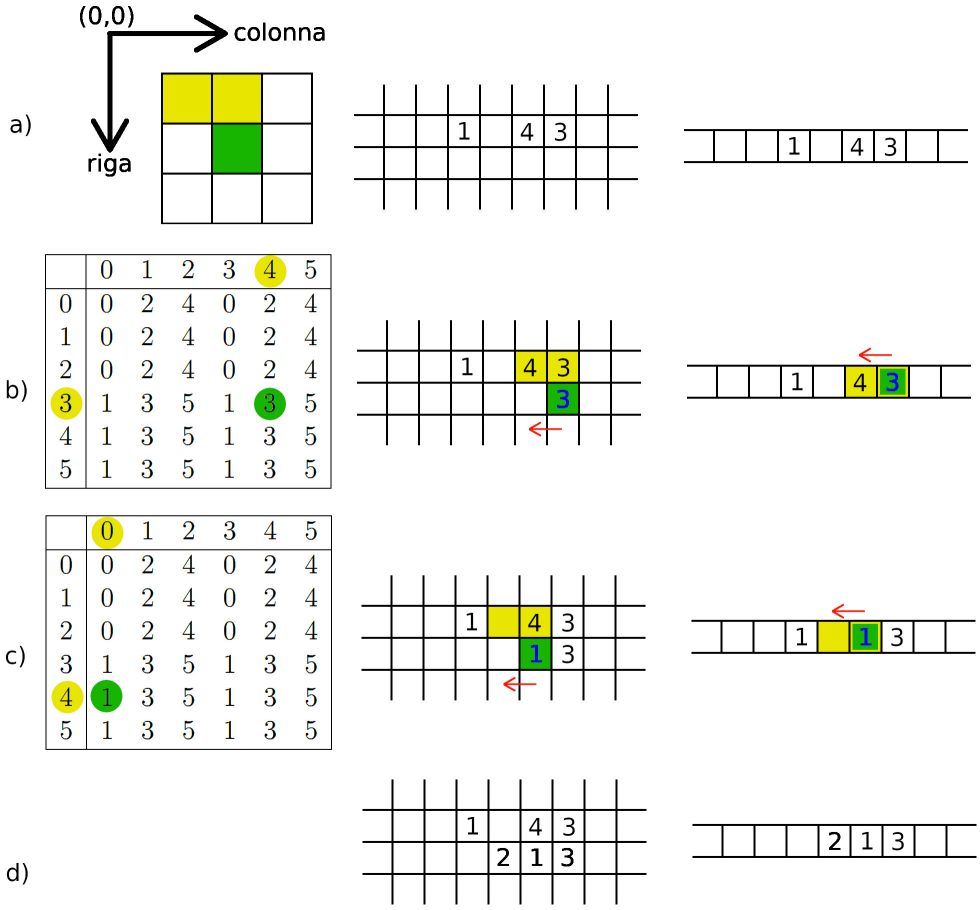


Figura 7: In questo esempio viene diviso per tre il numero 243 (”1043” in base 6) usando $CA_{\div 3}$. La nuova generazione viene calcolata in modo analogo a quello presentato in Figura 6, in questo caso però è necessario applicare le regole in ordine di colonna decrescente perché la variante monodimensionale non sovrascriva valori ancora necessari alla computazione.

Ne consegue che aver scelto di utilizzare la base 6 per esprimere i numeri rende triviale determinare se quel nodo avrà uno o due "nodi padre" nell'albero: si ha una duplicazione dei rami se e solo se l'ultima cifra è pari a 4.

1. Se il numero termina con la cifra 4:
   (a) Duplica il numero per il secondo ramo-padre
   $x \Rightarrow x'$
   (b) Imposta l'ultima cifra del nuovo numero a 3
   $x' \Rightarrow x' - 1$
   (c) Applica $CA_{\div 3}$
   $x' - 1 \Rightarrow \frac{x'-1}{3}$
2. Shift a sinistra del numero
   $x \Rightarrow 6x$
3. Applica $CA_{\div 3}$
   $6x \Rightarrow 6x/3 \Rightarrow 2x$

Riassumendo: per effettuare operazioni su numeri arbitrariamente grandi abbiamo deciso di utilizzare automi cellulari. Abbiamo scelto un automa cellulare in base 6 per semplificare l'individuazione dei nodi che sono immagine di due diversi valori in f. Per la parte del programma che utilizza il metodo diretto si sarebbe potuto utilizzare un diverso automa che consente una riduzione del numero di iterazioni, tuttavia abbiamo scelto di utilizzare anche in questo caso un automa in base 6 (meno efficiente) per agevolare il più possibile il passaggio di parametri tra le due parti dell'algoritmo.

Sviluppo dell'algoritmo

Punti chiave

Riepiloghiamo ora per chiarezza i punti chiave del problema presentati nelle precedenti sezioni, tali concetti saranno usati per progettare un algoritmo di verifica.

1. Un programma non può dimostrare che un numero è controesempio con traiettoria divergente a crescita infinita
2. Un programma potrebbe dimostrare che un numero è controesempio con traiettoria ciclica non-triviale.
3. Un programma che testa se un numero è verificato può dare falsi negativi, ma non falsi positivi[6].
4. Il metodo diretto è sempre più efficiente per verificare valori singoli.
5. Il metodo inverso potrebbe essere più efficiente per verficare intervalli di valori molto grandi, ma il suo costo cresce esponenzialmente.
6. Il TST di alcuni numeri ("picchi") è molto superiore al TST dei numeri ad essi vicini.
7. Il numero di nodi dell'albero cresce necessariamente con un fattore compreso tra 1 e 2. Sperimentalmente rileviamo un fattore di circa 1.264.
8. Sperimentalmente rileviamo che $TST(x) \approx k \log(x)$ per $x \gg 0$.

[6]Quando un algoritmo raggiunge l'obiettivo cercato (1 nel metodo diretto, il numero di input nel metodo inverso) è dimostrato inequivocabilmente che il numero passato come input è verificato. Se si ferma prima di aver raggiunto l'obiettivo potrebbe trattarsi di un controesempio, oppure di un numero che sarebbe stato verificato procedendo più a lungo.

Riteniamo importante sottolineare come parte di questi concetti siano derivati da ragionamenti logici o matematici dimostrabili, mentre gli ultimi derivano da considerazioni probabilistiche. Queste ultime portano a risultati assolutamente validi in certe condizioni (ad es. considerando un certo range di valori verificati sperimentalmente), ma non è da escludere la possibilità che esistano altre condizioni tali per cui le approssimazioni non siano più accurate. Pertanto ci assicureremo che le scelte compiute euristicamente alterino solo le prestazioni del programma e non i risultati: l'algoritmo dovrà fornire l'output corretto indipendentemente dalla validità delle approssimazioni.
Poter stimare anche solo approssimativamente il TST di un numero prima di averlo verificato ci permette di fare previsioni sui costi attesi. Tratteremo più formalmente dei costi di esecuzione in sezioni successive, per ora si ricordi semplicemente quanto precedentemente discusso sul tema.
Il metodo di verifica inverso richiede di percorrere i nodi dell'albero in ordine di altezza crescente tenendo traccia man mano di quali numeri sono incontrati, fino a raggiungere l'ultimo numero rimasto da verificare nell'intervallo $[A, ...B]$. Poiché la collocazione dei numeri nell'albero ha altezza equivalente al TST, un algoritmo che segue il metodo inverso dovrà salire fino al numero che nell'intervallo ha TST maggiore. Usando l'approssimazione derivante dai modelli probabilistici [10] possiamo dire che il caso peggiore in termine di costi vede B appartenente alla categoria dei "picchi", con $TST(B) \approx 32.33 * \log(B)$. Anche se non ci fosse alcun picco nell'intervallo l'elemento B sarebbe il più costoso, pari a $10.43 * \log(B)$, dunque proponiamo una variazione del metodo ibrido per migliorarne i tempi di esecuzione.
L'idea consiste nel procedere come per il metodo inverso dalla radice dell'albero segnando i nodi incontrati, ma si sceglie euristicamente di interrompere il calcolo esattamente all'altezza di $10.43 * \log(B)$. A quel punto, se le approssimazioni sono accurate nell'intervallo, la ricerca dovrebbe aver verificato tutti i valori dell'intervallo eccetto i picchi. Infine l'algoritmo dovrà prendere in esame i numeri rimasti e verificarli con il metodo diretto, poiché oltre ai picchi con TST alto ma finito potrebbero esserci anche controesempi per la congettura.
Questa scelta consente di ridurre il costo di esecuzione a una frazione di quanto sarebbe cercando di verificare l'intero intervallo, se l'approssimazione è accurata. Se invece l'approssimazione non dovesse essere accurata (per eccesso o difetto) il programma porterà comunque allo stesso risultato finale ma con performance inferiori.

Approssimazioni

Come spiegato nel precedente capitolo possiamo utilizzare un automa cellulare per eseguire semplici operazioni aritmetiche con numeri espressi in uno spazio di memoria arbitrariamente grande. Questo tipo di operazioni locali sono essenziali per calcolare gli iterati di Collatz quando non è possibile memorizzare l'intero numero in una variabile e compiere una operazione globale. Purtroppo l'algoritmo proposto finora richiede di conoscere il logaritmo in base 10 di B: calcolare tale valore usando solo delle operazioni locali su porzioni del numero non è un compito triviale.
Per risolvere questo problema sfruttiamo una proprietà dei logaritmi che consente il cambio di base:

$$\log_b(x) = \frac{\log_c(x)}{\log_c(b)} = \log_b(c)\log_c(x)$$

Sfruttando questa formula:

$$c * \log(x) = c * \log(6) * \log_6(x) \approx 0.778c * \log_6(x)$$

Effettuiamo ora un semplice ragionamento: sia x un numero intero composto da k cifre[7] espresso in base 6. Ne consegue che:

$$6^{k-1} \leq x \leq 6^k - 1$$

Ad esempio con 3 cifre in base 6 potrebbe essere compreso tra 555 e 100 (equivalenti a 215 e 36 in base 10). La funzione del logaritmo è strettamente crescente, dunque:

$$log_6\left(6^{k-1}\right) \leq \log_6(x) \leq \log_6\left(6^k - 1\right) < \log_6\left(6^k\right)$$
$$k - 1 \leq \log_6(x) < k$$

Dunque scegliendo di approssimare $\log_6(x) \approx k$ stiamo sovrastimando il valore effettivo con un errore massimo di una unità. Questo significa che tutti i punti del programma in cui applicheremo questa approssimazione useranno una stima leggermente più alta del valore reale: terremo a mente questo dettaglio per prevederne le conseguenze.
Una volta stabilito che (con k pari al numero di cifre di x):

$$c * \log(x) \approx 0.778ck$$

riscriviamo le stime del TST previste da Lagarias in [10] perché diventino in funzione del numero di cifre del numero in base 6.

[7] Qui e in seguito, si assume di non considerare nel conteggio delle cifre eventuali zero iniziali presenti nel CA.

TST [relativo a]	$g(x)$ $[\log(x)]$	$f(x)$ $[\log(x)]$	$f(x)[k]$
Caso comune	$6.95\log(x)$	$10.43\log(x)$	$8.11k$
Picco locale	$21.55\log(x)$	$32.33\log(x)$	$25.15k$
Massimo generale	$41.68\log(x)$	$62.52\log(x)$	$48.64k$

L'approssimazione del TST in caso comune è utilizzata nell'algoritmo per determinare l'altezza massima dell'albero da costruire. Sovrastimare k di una unità in questa situazione significa creare circa 8 livelli di albero aggiuntivo, nel peggiore dei casi. Se le approssimazioni sono accurate questo significa aumentare di un fattore 6.7 il numero totale di nodi da visitare[8]: sarà restituito lo stesso risultato ma in un tempo 6.7 volte superiore a quello necessario. Viceversa, se decidessimo di decrementare k di una unità per ridurre i costi, nel peggiore dei casi avremo sottostimato di 1 il valore effettivo. In questo caso il costo di costruire l'albero sarebbe 6.7 volte inferiore, non sappiamo però se tale altezza sia sufficiente per incontrare tutti i casi "comuni" e lasciare unicamente i "picchi" non verificati: nel peggiore dei casi fermarsi a una altezza sottostimata potrebbe non verificare nessun valore, portando alla verifica dell'intero range col metodo diretto (ciò significa rendere inutile l'intero costo di costruzione dell'albero).
L'approssimazione del TST in caso di picco locale potrebbe essere utilizzata nell'algoritmo per determinare un numero ragionevole di iterazioni da eseguire in metodo diretto prima di interrompere e segnalare all'utente il numero "sospetto". Tuttavia abbiamo scelto di usare per questo scopo il valore di massimo generale, in modo da avere un margine di sicurezza. Tale margine è utile in particolare quando si verificano valori relativamente piccoli: per tali valori l'approssimazione del TST è poco accurata e l'algoritmo rischia di segnalare falsi negativi[9]. Inoltre, se assumiamo che l'approssimazione sia tendenzialmente valida per valori B sufficientemente alti e che la Congettura sia vera, il numero di valori x in un qualunque intervallo tali che $TST(x) > 62.52\log(x)$ dovrebbe essere nullo (o tendenzialmente nullo). Gli unici casi segnalati dall'algoritmo con queste condizioni sarebbero una percentuale estremamente bassa di numeri nel range, tutti falsi negativi causati da errori di approssimazione. Pertanto l'aumento dei costi di esecuzione dovuto all'approssimazione per eccesso del valore k influirebbe unicamente

[8] Aggiungere 8.11 livelli comporta moltiplicare il numero di nodi totali di $1.264^{8.11} \approx 6.7$.

[9] Sperimentalmente rileviamo che anche questa misura precauzionale non è sufficiente per valori di B ancora più piccoli, ad esempio $B = 27$ (43 se espresso in base 6, cioè 2 cifre). Dato che $TST(27) = 111$ e $48.64 * 2 \approx 97$ la computazione è interrotta prima del tempo e il numero è segnalato come possibile controesempio. Incrementare il margine di sicurezza ulteriormente consente di eliminare tali falsi negativi, ma abbiamo comunque scelto di usare $48.64k$ poiché lo strumento che intendiamo realizzare è pensato per valori di B molto grandi.

su una frazione estremamente ridotta sull'insieme complessivo di valori da verificare.

Nodi dell'albero in memoria

In questa sezione descriveremo come viene gestita la memoria all'interno del programma, tuttavia si cercherà di mantenere un livello di astrazione alto trascurando i dettagli più specifici.
Definiamo una struttura dati che ha il compito di conservare le informazioni di un nodo dell'albero e sulla quale sono effettuati i calcoli. Tale struttura dovrà ospitare una sezione di dimensione variabile destinata allo spazio di lavoro degli automi cellulari, pertanto dovrà avere capacità sufficiente per memorizzare il numero in base 6 (come minimo[10]). Stabiliamo all'interno del programma una serie di regole che espandono la memoria allocata quando occorre rappresentare un numero le cui dimensioni superavano il precedente limite massimo: al termine dell'esecuzione del programma quell'area di memoria sarà grande a sufficienza per contenere il numero più grande incontrato nella successione.
Nel caso del metodo diretto non possiamo prevedere quanto sarà grande il massimo iterato della sequenza, tuttavia sappiamo che ci occorre memorizzare un singolo "nodo" per eseguire questa parte dell'algoritmo. Ipotizziamo per assurdo che il numero da verificare col metodo diretto sia talmente grande da occupare da solo la maggior parte della memoria disponibile. Sappiamo che il numero potrebbe crescere ulteriormente, tanto da richiedere una estensione della memoria. Tuttavia il numero di cifre cresce logaritmicamente rispetto al numero stesso, ogni unità di spazio aggiuntivo destinato a questo scopo fa crescere esponenzialmente il massimo numero rappresentabile. Per questo motivo il metodo diretto non dovrebbe mai richiedere più memoria di quanta disponibile, a meno che il valore in input non parta già superiore o molto vicino al limite massimo.
Studiare il massimo spazio richiesto per archiviare un numero con il metodo inverso è curiosamente più semplice: l'albero è costruito a partire da 1 fino a una altezza di esattamente $10.43 \log(x)$. I valori incontrati nell'albero salgono e scendono in valore, tuttavia è certo che il valore più alto possibile a quella altezza si troverà nel ramo delle potenze del 2, ossia l'unico ramo che nel metodo inverso incrementa costantemente il valore del suo iterato. All'altezza massima abbiamo quindi un numero massimo di valore $2^{10.43 \log(x)}$,

[10]Nell'implementazione è allocato spazio per le cifre del numero, più altro spazio libero per consentire l'espansione del numero e il calcolo del CA. Per semplicità assumiamo in questa descrizione che serva spazio unicamente per conservare il numero attuale.

cioè circa $4.03 \log(x)$ cifre (approssimando come in precedenza). Come prima approssimiamo il logaritmo di x con il numero di cifre e concludiamo che per conservare il numero più grande che sarà incontrato nell'albero occorrono poco più del triplo del numero di cifre iniziali ($3.14k$). Se ipotizziamo che tutti i nodi siano cresciuti fino a quel valore (approssimazione per eccesso) possiamo stimare che al massimo il costo in memoria del metodo inverso sarà il numero di nodi contemporaneamente registrati in memoria moltiplicato per il triplo delle cifre dell'iterato iniziale.
Ci chiediamo ora quanti siano i nodi da tenere contemporaneamente in memoria per eseguire il metodo inverso. Definendo l'albero di Collatz abbiamo dichiarato che per costruire l'albero occorre fare crescere i vari rami alla stessa altezza in parallelo, dato che non si sa quanto a lungo occorre far crescere il grafo prima di incontrare tutti i numeri cercati.

```
inserisci "1" in lista "nodi"
altezzaCorrente=0;
finché altezzaCorrente<altezzaRichiesta:
   altezzaCorrente = altezzaCorrente + 1
   per ogni nodoCorrente in nodi:
      se nodoCorrente si biforca in due nodi:
         crea nuovonodo in fondo a "nodi"
         nuovonodo = (nodoCorrente - 1 ) / 3
      nodoCorrente = nodoCorrente * 2
```

Seguendo queste regole il numero di nodi in memoria aumenta all'aumentare del numero di nodi al livello corrente, possiamo dire che il massimo numero di nodi da tenere in memoria sarà pari al numero di nodi del livello dell'albero più alto. Sperimentalmente rileviamo che l'albero ha una crescita esponenziale, pertanto il numero di nodi all'ultimo livello crescerà esponenzialmente in funzione dell'altezza.
Possiamo però usare una strategia alternativa per costruire l'albero, in modo da ridurre il numero di nodi contemporaneamente conservati. Abbiamo scelto euristicamente di interrompere l'esecuzione a una altezza ben precisa, pertanto usiamo il seguente algoritmo:

```
inserisci "1" in lista "nodi"
finché la lista "nodi" ha elementi:
   nodoCorrente è l'elemento in cima a "nodi"
   finché nodoCorrente deve crescere in altezza:
      incrementa altezza di nodoCorrente
      se nodoCorrente si biforca in due nodi:
         crea nuovonodo in cima a "nodi"
         nuovonodo = (nodoCorrente - 1 ) / 3
         altezza nuovonodo = altezza nodoCorrente
      nodoCorrente = nodoCorrente * 2
   distruggi nodoCorrente
```

In questa versione dell'algoritmo ogni nodo tiene traccia individualmente della propria altezza e i vari rami non sono fatti crescere contemporaneamente. Si trasforma il nodo per farlo risalire nell'albero il più rapidamente possibile, tenendo in sospeso i rami secondari incontrati come in una sorta di pila. Non si tratta propriamente di una pila poiché è possibile rimuovere elementi che non si trovano in cima, tuttavia sono aggiunti elementi solamente alla cima e il successivo elemento preso in considerazione viene scelto guardando in cima alla pila. Quando il primo nodo giunge alla massima altezza viene "distrutto"[11] e il programma prende in considerazione il nodo che in quel momento è più in alto nella lista. Tale nodo deriverà dalla più recente biforcazione del ramo principale, pertanto parte già a una altezza molto vicina a quella definitiva (dunque costruirà un numero di rami extra molto ridotto, o nullo). L'algoritmo ripete l'esecuzione cercando di esplorare l'albero in profondità finché tutti i nodi non sono giunti al termine e la lista si è svuotata completamente.
Questa versione dell'algoritmo è sicuramente più complicata da implementare e da comprendere ma ha un vantaggio interessante. Il nodo-radice viene iterato un numero di volte pari all'altezza, e ad ogni iterazione è possibile creare un secondo ramo in aggiunta a quello correntemente considerato. Ricordiamo che l'albero di Collatz ha crescita inferiore a quella di un albero binario, ma per semplicità assumiamo temporaneamente che la crescita sia la stessa. In queste condizioni si crea sempre un nuovo ramo ad ogni iterazione, dunque all'altezza h sono stati creati altri h nodi. Quando si giunge per la prima volta alla massima altezza sono presenti in memoria $h+1$ nodi, il primo nodo ha esaurito le iterazioni e viene rimosso. Il nodo successivo è

[11] Nell'implementazione i nodi "distrutti" sono semplicemente spostati in una pila di "riserve", poiché sarebbe un inutile spreco di risorse allocare e deallocare continuamente aree di memoria. Quando un nodo deve essere "creato" viene estratto dalla pila di riserve e inizializzato.

l'ultimo aggiunto nella pila, anch'esso esaurisce le iterazioni senza ulteriori diramazioni e viene rimosso. Con $h-1$ nodi rimanenti l'esecuzione considera il nodo successivo, che raggiunge l'ultimo livello creando un secondo nodo. L'algoritmo procede in questo modo sino alla fine, oscillando ma senza mai superare tale numero: concludiamo che per un albero binario il numero di nodi conservati contemporaneamente è al massimo $h+1$. Dato che l'albero di Collatz ha una crescita certamente inferiore, per poter visitare tutti i nodi entro altezza h non occorreranno mai più di h nodi conservati in memoria contemporaneamente. In questo modo il costo di esecuzione è rimasto pressoché uguale (visto che sono esplorati in ogni caso tutti i nodi dell'albero), ma il costo in memoria è passato da una crescita esponenziale a una crescita lineare in funzione dell'altezza.
Abbiamo scelto di usare una variabile intera da 4 byte per ogni cella del vettore usato come area di lavoro dell'automa cellulare: tale scelta è stata compiuta nel programma sviluppato per massimizzare la velocità di esecuzione. Poteva essere usato una variabile di dimensioni inferiore (come un carattere da un byte o porzioni di un singolo byte fino a un minimo di 3 bit), ma questo risparmio di memoria giungerebbe al prezzo di un deterioramento delle prestazioni. Effettuiamo una stima molto approssimativa dell'ordine di grandezza del costo in memoria, dato un termine massimo dell'intervallo B[12].
Parte del costo in memoria di esecuzione del programma è necessario per allocare alcune variabili "fisse", cioè sempre presenti e di dimensioni costanti indipendentemente da quanto B sia grande o piccolo. La parte rimanente del costo in memoria è rappresentata dai nodi dell'albero, cioè h nodi ciascuno composto da altre variabili di dimensioni fisse e un'area di lavoro per i CA. L'area di lavoro ha dimensione variabile nell'arco del programma, assumiamo come upper bound che al termine dell'esecuzione tutte le aree di lavoro abbiano dimensione massima: come visto sopra questo equivale a triplicare il numero di cifre inizialmente necessarie per B. Se consideriamo come trascurabili le variabili di costo fisso[13] possiamo riassumere che il costo finale in memoria sia circa equivalente al numero di nodi per le dimensioni massime dello spazio di lavoro:

$$h * 4.03 \log(B) * 4 \text{ byte} = 10.43 \log(B) * 4.03 \log(B) * 4 \text{ byte}$$
$$\approx 168.13 \log^2(B) \text{ byte}$$

[12]Il termine A di inizio dell'intervallo non influisce sul costo in termini di tempo di esecuzione o memoria per quanto concerne la creazione dell'albero. "A" interviene unicamente nella parte di algoritmo finale, ossia quando sono verificati tutti gli elementi dell'intervallo AB non ancora incontrati.

[13]Importante: il costo in memoria di tali variabili non è trascurabile in termini assoluti, potrebbe anzi superare il costo delle aree di lavoro in certe condizioni. Tuttavia tale costo resta fisso indipendentemente da B, dunque al crescere dell'intervallo sarà una percentuale sul totale via via inferiore, tendente a zero con B estremamente grande.

Se ad esempio $B = 10^{10}$ (cioè 13 cifre in base 6), allora il costo in memoria dovrebbe aggirarsi intorno ai 16.5 kilobyte. Anche se elevato al quadrato, il logaritmo rallenta notevolmente la crescita dei costi anche per valori di B molto grandi. Come vedremo nel capitolo successivo il costo di esecuzione in termini di tempo cresce in modo molto più rapido del costo in memoria: immaginando di far crescere progressivamente B e misurare tempi di esecuzione e costo in memoria, il tempo di esecuzione diventerà "inaccettabile" (troppo lungo per essere adatto all'uso pratico) ben prima del punto in cui la memoria si rivelerà insufficiente. Pertanto sacrificare efficienza nel consumo di memoria (scegliendo un intero per ogni cella dello spazio di lavoro) è una scelta giustificabile e anzi nettamente preferibile a sacrificare velocità di calcolo.

Numeri verificati in memoria

Nella precedente stima abbiamo trascurato di considerare un altro aspetto del programma che richiede spazio in memoria: memorizzare quali numeri sono stati visitati con l'albero per dedurre quali dovranno essere verificati con il metodo diretto (o in alternativa memorizzare i numeri non ancora verificati).
Possiamo fare una premessa generale che potremo sfruttare in ogni successiva ipotesi: è possibile confrontare in modo molto semplice due valori in base 6 espressi come sequenza di cifre in celle intere, innanzitutto confrontando il numero di cifre. Se questo confronto non porta a un risultato (numero di cifre equivalente) si confrontano le coppie di cifre corrispondenti a partire dalle più significative, nella prima occasione in cui le due non sono uguali abbiamo individuato quale numero è maggiore tra i due. Se lo scorrimento giunge al termine senza risultato abbiamo invece scoperto che i due numeri sono uguali: ne consegue che confrontare due numeri non ha un costo in memoria aggiuntivo (a patto che i due valori considerati siano già espressi in tale forma in memoria) e il costo in termini di tempo è generalmente costante o variabile fino a un massimo di $log(n)$ in ordine di grandezza. Pertanto possiamo verificare molto semplicemente se l'iterato corrente è compreso nell'intervallo con costo al massimo $log(n)$, avendo l'accortezza di memorizzare inizialmente i due valori A e B come vettore di cifre intere in base 6.
Riflettiamo ora sulle opzioni a nostra disposizione: sono possibili varie strategie per memorizzare i numeri verificati. Chiaramente il nostro obiettivo è individuare quali tra questi metodi possa fornire il migliore rapporto di costi tra tempi di esecuzione e consumo di memoria, ma anche assicurarci che questa scelta non influisca negativamente sul range di valori che possono

essere effettivamente verificati dato l'hardware a disposizione. Per chiarezza: in questa sezione intendiamo con "chiave" una sequenza di cifre nello spazio di lavoro che rappresentano un numero in base 6; l'obiettivo che cerchiamo di raggiungere è utilizzare la chiave per riconoscere il prima possibile se quel numero era stato verificato, ad esempio raggiungendo un "valore" associato alla chiave (uno stato "verificato" o "non verificato").
Una prima idea potrebbe consistere nel conservare una lista di chiavi, riempita gradualmente dai numeri incontrati dal metodo inverso. Il compito di trasferire una chiave da o verso gli spazi di lavoro è triviale, poiché occorre semplicemente copiare le $\log(x)$ cifre scorrendole di seguito. Tuttavia sarebbe molto costoso cercare se un elemento è presente o meno nella lista e non sarebbe un uso parsimonioso della memoria. Consideriamo ad esempio un caso limite $A = B - 1 \wedge B \gg 0$: anche se il numero di elementi compresi nell'intervallo è minimo (2 soli valori) occorrerebbero molti byte per memorizzare ogni cifra del numero. Sono possibili alcune variazioni di questa idea (come scegliere di ordinare la lista una volta completata, partire da una lista di tutti i numeri nel range e rimuovere quelli incontrati...) ma nel complesso non appare una soluzione convincente.
Una seconda strategia potrebbe consistere nel conservare un vettore di dimensioni $B - A + 1$ elementi: l'i-esimo elemento è 0 o 1 a seconda dello stato di verifica del numero $A + i$. Il costo in memoria è estremamente più basso di quello della prima soluzione proposta, un singolo byte sarebbe sufficiente per ogni numero dell'intervallo[14] dunque basterebbero due byte per l'esempio precedente. Tuttavia in questa situazione è piuttosto complesso calcolare l'indice del vettore dato il numero espresso nello spazio di lavoro: occorre implementare un CA per effettuare la differenza tra il valore considerato e A (o un altro metodo per trattare la differenza come sequenza di operazioni locali tra singole celle dell'area di lavoro), poi inserire il risultato di tale differenza in una variabile da utilizzare come indice del vettore. Questa operazione non avrebbe senso se questo valore superasse il massimo valore archiviabile dalla variabile usata, ma avere un overflow in una variabile *size_t* sarebbe possibile solo se lo stesso $B - A$ fosse superiore al massimo numero di celle indicizzabili (e in questo caso non sarebbe stato possibile allocare un vettore tanto grande in primo luogo). Anche in questo caso sono possibili varianti della prima idea: si potrebbe ad esempio decidere di ignorare il termine A dell'intervallo e creare una tabella tra 0 e B. In questo caso si eviterebbe la necessità di calcolare la differenza tra il numero corrente e A,

[14]Si potrebbe ridurre le dimensioni di ogni elemento a un singolo bit, poiché essenzialmente si tratta di una *flag* vera o falsa. Come sopra però tale riduzione è possibile a costo di velocità di esecuzione e semplicità nel codice sorgente.

poiché il numero stesso sarebbe un indice accettabile del vettore. È però evidente che questa semplificazione significa potenzialmente utilizzare molta più memoria di quanto effettivamente necessaria e si riduce il range massimo di valori verificabili: se anche è possibile allocare un vettore di $B - A$ elementi non necessariamente è possibile allocare e indirizzare B elementi. E questa semplificazione renderebbe inutile il principio stesso di utilizzare CA per calcolare numeri arbitrariamente grandi: se ad ogni iterazione il numero deve essere convertito in una singola variabile unsigned sarebbe stato possibile anche iterare direttamente con tale variabile. Si potrebbe pensare di usare una strategia diversa programmaticamente a seconda dei valori di input (strategia veloce ma dispendiosa in memoria per B molto "piccolo", strategia più costosa ma leggera in memoria per intervallo molto vasto e B "grande") oppure sfruttare la memoria di massa del computer (caricando e scaricando parti della ram all'occorrenza per rendere virtualmente illimitato lo spazio di archiviazione, ma al prezzo di un rallentamento estremamente alto delle operazioni), ma non necessariamente porterebbe a benefici tanto grandi da giustificare la complessità aggiuntiva del codice.
Cercando una soluzione che ci consenta di mappare più facilmente i numeri in un vettore abbiamo preso in considerazione l'uso di "Hash Tables". Questa strategia prevede l'uso di una "funzione di hash", un algoritmo o funzione matematica pensata per mappare un certo input in un output differente in modo non invertibile (ad esempio una stringa di dimensioni arbitrarie in una stringa di dimensioni fisse, senza possibilità di tornare alla prima stringa partendo dall'output). L'output della funzione di hash è usato come indice per accedere rapidamente a una cella della tabella: idealmente ogni input diverso dovrebbe essere indirizzato a celle della tabella diverse, ma a seconda della funzione e dei valori in input potrebbe capitare che un certo output sia il risultato di due input diversi. Questo scenario si chiama "collisione" e occorre stabilire una strategia per prevenirle o reagire alla loro comparsa. Se si usa una funzione adatta e la tabella hash è ben dimensionata (di dimensioni sufficienti a ridurre il rischio di collisioni) il costo di cercare la presenza o assenza di un certo numero è estremamente ridotto (pressoché costante, indipendente dal numero di dati inseriti). In letteratura ci sono numerose funzioni di Hash adatte per uso generico, ma non è certo che portino alla massima efficienza nel nostro caso e costruire una funzione di hash specifica sarebbe un problema complesso.
Invece di approfondire questa possibilità ci siamo indirizzati sull'idea di utilizzare un albero per accedere rapidamente al valore cercato senza bisogno di una funzione di Hash: come nel precedente esempio costruiamo un vettore di valori booleani inizializzati a 0, ma questa volta sarà possibile accedere alle celle esplorando una sequenza di nodi dell'albero equivalente al numero

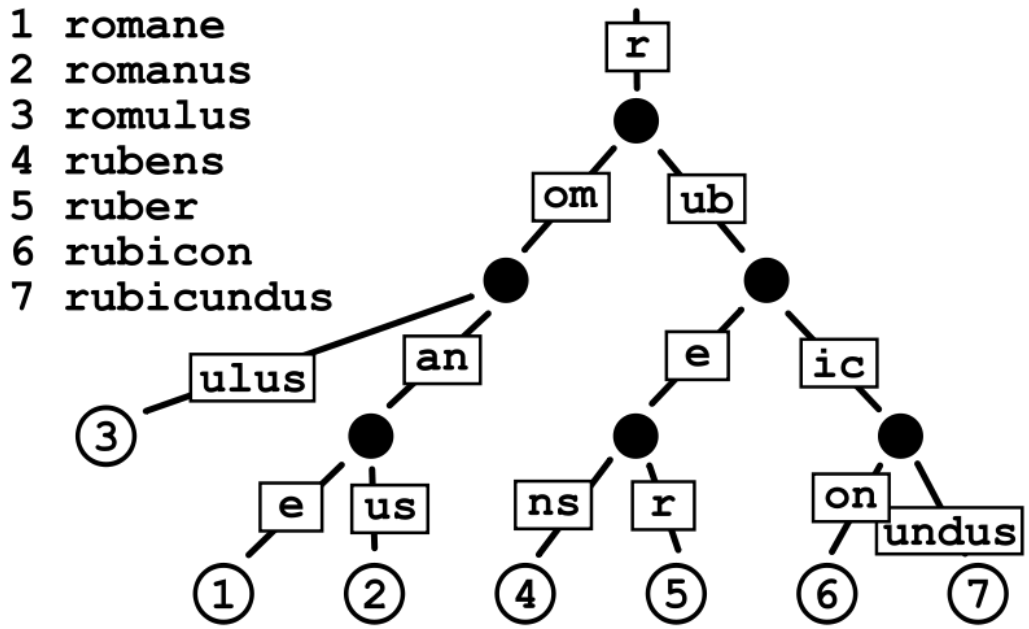


Figura 8: Esempio di Radix Tree usato per mappare un valore numerico a una stringa (chiave). Per raggiungere il valore associato alla chiave "rubicon" si scorre dalla radice lungo i percorsi che formano la chiave usata per la ricerca. Raggiungiamo in questo modo il valore 6 in 4 passaggi: nel caso peggiore occorre un numero di passi equivalente alla lunghezza della chiave.

di cifre della chiave. In questo scenario ogni nodo dell'albero di ricerca consente di raggiungere altri sei nodi, a loro volta altri sei e si prosegue fino agli ultimi nodi, che puntano alle specifiche celle del vettore di flag. In pratica la chiave è processata gradualmente, ogni cifra è usata come indirizzo per scegliere il percorso che condurrà al nodo successivo fino all'ultima cifra che indica il valore cercato (costo complessivo equivalente a scorrere le cifre del numero una volta). Per completare l'albero devono essere costruiti circa B nodi (un numero piuttosto alto come abbiamo visto se $B \gg 0$). Esiste però una struttura dati con simile principio ma più efficiente: il "Radix Tree". In questo tipo di albero (di cui è presentato un esempio in Figura 8) la chiave di ricerca (che è una serie di caratteri) è processata man mano che si esplora l'albero, ma invece di usare un singolo carattere come indirizzo per saltare da un nodo all'altro se ne usa un numero variabile per comprimere insieme i nodi che hanno lo stesso prefisso nella chiave (e ridurre le dimensioni dell'albero, che comunque è costruito man mano che sono aggiunti i numeri e non fin dall'inizio). Nella fase in cui occorre verificare se un certo numero è stato

verificato questo approccio è più rapido, poiché seguendo nell'albero la chiave di un numero mai aggiunto nell'insieme si può giungere a un "vicolo cieco" prima di completare la scansione della chiave: nel caso peggiore si avrà una risposta con un numero di operazioni equivalente alla lunghezza della chiave e l'albero occuperà solo lo spazio strettamente necessario.
Nel programma abbiamo utilizzato una libreria preesistente, chiamata "Judy Arrays" ([1]), che utilizza una versione altamente ottimizzata di un Radix Tree. Sfruttando il più possibile la cache della CPU offre prestazioni pari (o talvolta superiori) a quelli delle tabelle hash, occupando un numero di byte in memoria estremamente ridotto.

Analisi dei costi e conclusioni

In questo capitolo discuteremo più formalmente i costi di esecuzione dell'algoritmo creato (che chiamiamo "metodo ibrido"), finora appena accennati, alla luce delle informazioni dei precedenti capitoli. Inoltre effettueremo un confronto tra i vari metodi e proporremo alcune strategie che potrebbero essere usate per migliorarli ulteriormente.
Come base di partenza assumiamo che le approssimazioni derivanti da modelli probabilistici siano accurate per $B \gg 0$, anche se indirettamente significa assumere che non esistano controesempi per la congettura. Se non considerassimo attendibili le approssimazioni il valore $TST(x)$ sarebbe imprevedibile, ostacolando il confronto dei costi.

Costo del metodo diretto

Non ha senso utilizzare metodo inverso e ibrido per singoli valori, dunque per ottenere risultati confrontabili consideriamo il costo di verificare un range di valori col metodo diretto. Abbiamo già visto che il costo del metodo diretto per un singolo numero equivale al TST del numero di partenza, e conseguentemente il costo per un intervallo è

$$\sum_{x=a}^{b} TST(x)$$

Le approssimazioni suggeriscono che i numeri si suddividano in una categoria "semplice" di costo $10.43\log(x)$ e una categoria molto più rara dal costo di $32.33\log(x)$. Sia k il rapporto tra il numero di "picchi" e il numero totale di elementi nel range, cioè

$$k \in \mathbb{R} | 0 \leq k \leq 1$$

Abbiamo quindi $k(B - A + 1)$ picchi e $(1 - k)(B - A + 1)$ casi comuni.
Intuitivamente il costo complessivo di verificare un intervallo col metodo diretto equivale al costo di verificare tutti i casi semplici di quell'intervallo

col metodo diretto, più il costo di verificare tutti i picchi di quell'intervallo sempre col metodo diretto. Considerando che

$$\forall x \in [A, B], \log(x) \leq \log(B)$$

scomponiamo la precedente sommatoria nelle due categorie di casi:

$$\sum_{x=a}^{b} TST(x) < (1-k)(B-A+1)10.43\log(B) + k(B-A+1)32.33\log(B)$$

Si tratta di una disuguaglianza perché abbiamo approssimato i logaritmi dei vari elementi nel logaritmo del solo B. Completando i calcoli:

$$\sum_{x=a}^{b} TST(x) < (10.43 - k10.43 + k32.33)(B-A+1)\log(B)$$

$$\sum_{x=a}^{b} TST(x) < (10.43 + k21.9)(B-A+1)\log(B)$$

Possiamo considerare k una costante e stiamo assumendo che $B \gg 0$. Non sappiamo come si collochi A rispetto a B se non per il semplice $A < B$, distinguiamo i due casi limite:

$$A \approx B \Rightarrow \sum_{x=a}^{b} TST(x) < c\log(B)$$

$$A \ll B \Rightarrow \sum_{x=a}^{b} TST(x) < cB\log(B)$$

Dunque il costo di verifica di un intervallo col metodo diretto è:

$$C_{DIRETTO} = \begin{cases} O(\log n) & \text{se } A \approx B \\ O(n \log n) & \text{se } A \ll B \end{cases}$$

Intuitivamente il caso limite $A \approx B$ ha un intervallo tendente al singolo elemento, non è sorprendente che l'ordine di grandezza del costo sia analogo a quello di verificare un singolo elemento col metodo diretto (anch'esso proporzionale a $\log n$). Sapevamo già che per intervalli estremamente piccoli di numeri da verificare il metodo diretto è da preferirsi, nelle seguenti analisi eviteremo l'ipotesi $A \approx B$.

Costo del metodo inverso

Abbiamo già visto che verificare un intervallo [A,B] col metodo inverso ha un costo del tutto equivalente a verificare il singolo valore di TST più alto, poiché è costruito un albero di pari altezza. Sia M tale valore:

$$M \in [A, B] | TST(M) = max(TST(A), ..., TST(B))$$

Sappiamo che per incontrare quel valore col metodo inverso dobbiamo costruire l'albero fino all'altezza $TST(M)$, sfruttando le approssimazioni calcoliamo che il numero di nodi a quel livello (e conseguentemente il costo del metodo inverso) è circa:

$$1.264^{TST(M)} = \begin{cases} 1.264^{32.33 \log(M)} & \text{se esiste un picco in [A,B]} \\ 1.264^{10.43 \log(M)} & \text{altrimenti} \end{cases}$$

Se $A \approx B$ l'intervallo ha un numero di elementi tendente a 1, dunque è certamente più conveniente utilizzare il metodo diretto. Stiamo dunque considerando il caso di un intervallo molto vasto con $B \gg 0$, in queste condizioni è pressoché certo esista almeno un picco nell'intervallo. Assumendo che $TST(M) \approx TST(B)$, concludiamo che

$$1.264^{TST(M)} \leq 1.264^{32.33 \log(B)}$$

Sfruttiamo come in precedenza le proprietà dei logaritmi:
$1.264^{32.33 \log(B)} = (1.264^{\log_{10} 1.264 * \log_{1.264} B})^{32.33} = (1.264^{\log_{1.264} B})^{32.33 * \log_{10} 1.264} = B^{32.33 * \log 1.264} < B^{3.29}$
Infatti l'albero cresce esponenzialmente rispetto all'altezza, ma essendo l'altezza un logaritmo concludiamo che il metodo inverso ha un costo superiore al cubico (fortunatamente inferiore al costo esponenziale, per quanto sia comunque una crescita molto rapida).

$$C_{INVERSO} = O(n^{3.29})$$

Costo del metodo ibrido

L'algoritmo da noi progettato verifica col metodo inverso tutti i casi semplici, per poi verificare col metodo diretto i picchi. Usiamo ragionamenti analoghi a quelli delle due sezioni precedenti e scomponiamo il costo complessivo in due parti:

$$1.264^{10.43 \log(B)} + k(B - A + 1)32.33 * \log(B)$$

Chiaramente il primo termine della somma rappresenta il costo dell'albero all'altezza "comune", il secondo termine rappresenta il costo del metodo diretto per k "picchi". Calcoliamo il primo termine come in precedenza:

$$1.264^{10.43\log(B)} < B^{1.1}$$

Con la condizione $A \ll B$ approssimiamo il secondo termine:

$$k(B - A + 1)32.33 * \log(B) \approx c * B\log(B)$$

Il primo termine cresce con ordine di grandezza $n^{1.1\cdot}$, più rapidamente del secondo di ordine $n \log n$: con valori B molto grandi supererà sempre quest'ultimo anche se c fosse molto alta. Concludiamo che:

$$C_{IBRIDO} = O(n^{1.1})$$

Confronto tempi di esecuzione

Abbiamo determinato a livello teorico come crescono i costi di esecuzione in funzione degli input, proveremo ora a confrontare i tempi di esecuzione che si ottengono empiricamente. Occorre sottolineare che questa misura è soggetta a errori: in genere due diverse misurazioni portano a risultati diversi. Cause tipiche di questo fenomeno sono:

- funzioni di misurazione imprecise (amplificazione degli errori causata da operazioni tra numeri con cifre decimali a precisione finita, limitazioni nella massima precisione delle funzioni usate per calcolare la differenza tra due istanti di tempo, approssimazioni)
- differenze di implementazione (un certo algoritmo implementato in modi diversi potrebbe fornire prestazioni diverse da quelle stimate ad alto livello)
- differenze di hardware (uno stesso codice C eseguito su due macchine diverse porterà probabilmente a tempi di esecuzione diversi)
- altri processi nel sistema operativo (in una stessa macchina sono generalmente in esecuzione multipli processi senza correlazione con quello preso in esame)

I tempi di esecuzione sono però un modo estremamente semplice di presentare questo genere di dati e renderli confrontabili a "colpo d'occhio" (pur con una certa approssimazione), senza richiedere all'osservatore conoscenze

specifiche sul tema. Pertanto presentiamo i tempi di esecuzione del programma "ibrido" prodotto in questo lavoro di tesi, confrontandoli con i tempi di esecuzione del metodo "diretto" nello stesso intervallo[15] (i valori in input sono da considerarsi espressi in base 10). La prima tabella illustra i tempi di esecuzione nel caso in cui l'intervallo è "ampio", tale per cui $[A, B]$ tende all'intervallo massimo $[1, B]$.

	$A = 1 \Rightarrow B - A \approx B$			
	$B = 10^3$	$B = 10^4$	$B = 10^5$	$B = 10^6$
Diretto	8ms	0.105s	1.345s	17.240s
Ibrido	9ms	0.147s	1.547s	17.781s

La seconda tabella mostra i tempi di esecuzione nel caso di un intervallo più ristretto, tendente al singolo elemento B.

	$A = B - 1 \Rightarrow B - A \approx 0$			
	$B = 10^3$	$B = 10^4$	$B = 10^5$	$B = 10^6$
Diretto	$<$ 1ms	$<$ 1ms	$<$ 1ms	$<$ 1ms
Ibrido	1ms	0.067s	0.457s	3.233s

Interpretiamo ora i risultati. Osservando la seconda tabella rileviamo che, come previsto, per intervalli estremamente ristretti il metodo diretto fornisce prestazioni nettamente superiori. Non sorprende che anche incrementando B di un fattore 10^3 il tempo di esecuzione appaia costante: data la crescita logaritmica del costo sarebbe necessario incrementarlo di un fattore ben superiore prima di iniziare a percepire una differenza nei valori.
Studiando invece i valori corrispondenti nella prima tabella possiamo rilevare che le prestazioni del metodo ibrido sono molto vicine a quelle del metodo diretto quando l'intervallo è molto vasto (pur rimanendo sempre più lento). Tuttavia notiamo una caratteristica dei tempi di esecuzione che sembra discostarsi da quanto previsto teoricamente. Abbiamo precedentemente discusso di come il costo di esecuzione del metodo inverso non dipenda dal termine "A" di input, ma dal solo termine B. Appare dunque curioso come il metodo ibrido (molto simile al metodo inverso) abbia tempi di esecuzione chiaramente diversi tra la prima e la seconda tabella, pur mantenendo invariato il termine B. Possiamo però spiegare molto semplicemente il fenomeno: se la prima fase di costruzione dell'albero è identica nei due casi (e pertanto con

[15] Anche per la verifica dell'intervallo con metodo diretto è stato impiegato lo stesso programma, ma verificando individualmente ogni valore dell'intervallo e sommando infine i rispettivi tempi di esecuzione. Come già descritto in precedenza, questa implementazione del metodo diretto non usa tecniche per convergere a 1 in un numero di passi inferiore a TST.

tempi pressoché equivalenti), la seconda fase di verifica è molto diversa. Nel caso dell'intervallo di ampiezza minima non occorre molto tempo per scorrere l'intero intervallo in cerca di numeri da verificare, e similmente saranno molti di meno i numeri che sarà necessario verificare col metodo diretto (trattandosi di un sottoinsieme dell'intervallo più ampio). Dunque il termine in input A influenza i tempi di esecuzione della seconda fase del metodo ibrido, mentre il termine B influenza l'intero programma.
Possiamo soffermarci brevemente su un ultimo punto di riflessione: considerato che i valori di B con cui è stato testato il programma nelle due tabelle non sono abbastanza grandi da rendere accurate le stime probabilistiche, è molto probabile che l'albero costruito nella prima fase non abbia incontrato tutti i numeri appartenenti alla classificazione "casi semplici". Ne consegue che tali numeri saranno stati verificati con il metodo diretto durante la seconda fase, incrementando ulteriormente la differenza di tempo nel metodo ibrido tra prima e seconda tabella. Incrementando B ulteriormente (a un livello sufficiente da rendere accurate le stime) ci aspettiamo che tutti i "casi semplici" siano verificati in prima fase, lasciando solamente i "picchi" ed eventuali controesempi nella seconda fase.
In ogni caso i tempi di esecuzione del metodo ibrido con intervallo ampio saranno sempre più alti dei corrispondenti casi con $A \approx B$.

Conclusioni

Allo stato attuale[16] sono stati verificati tutti i numeri fino a 2^{68} ([12]) e possiamo aspettarci che questo intervallo continuerà ad essere esteso in futuro. Basandoci su questo punto di partenza possiamo dare per scontato che qualunque algoritmo di verifica deve puntare all'efficienza con valori di B estremamente grandi perché sia utile alla ricerca. Abbiamo però visto che (contrariamente a quanto prevedevamo inizialmente) il metodo inverso e le sue variazioni non sono più efficienti del metodo diretto per grandi intervalli, e anzi tali metodi sono via via meno efficienti al crescere di B. La via dell'albero inverso non porterà a risultati migliori di implementazioni del metodo diretto, anche in considerazione del fatto che entrambe le tipologie di algoritmo possono essere migliorate in efficienza con il parallelismo e che il metodo inverso ha un consumo di memoria notevolmente superiore a quello diretto (pertanto, avendo a disposizione un tempo di calcolo infinito, il metodo diretto raggiungerà il massimo numero verificabile entro i limiti di memoria molto più tardi del metodo inverso).

[16] Aggiornato a settembre 2020

Parallelismo e ottimizzazioni

Concludiamo la presente trattazione fornendo alcuni possibili spunti di ricerche future. In un capitolo precedente abbiamo trattato di come l'uso di una certa base nel calcolo abbia un'influenza notevole sul numero di iterazioni necessarie per la convergenza a 1. Procedendo unicamente col metodo diretto non è più un problema "saltare" passaggi intermedi, pertanto diventa assolutamente conveniente ridurre al minimo il numero di iterazioni rispetto al teorico TST di $f(x)$. Gli Automi Cellulari che operano in base 2 sembrano un ottimo punto di partenza data la possibilità di concatenare in una singola iterazione qualunque numero di divisioni per due consecutive, sarebbe però da valutare quanto influisca negativamente dover operare su un numero di cifre più grande per ogni iterazione. Anche se non si desidera utilizzare un automa cellulare è necessario studiare accuratamente il modo più efficiente per memorizzare gli iterati, garantendo al contempo alte prestazioni e basso consumo di memoria (per rendere accettabile il calcolo di numeri sempre più grandi).
Risulta inoltre interessante studiare il problema assumendo fin da principio che l'algoritmo sia eseguito in parallelo da più processi o macchine diverse. Prendiamo per esempio l'algoritmo ibrido sviluppato: sarebbe possibile adattarlo per l'esecuzione concorrente suddividendo il compito della creazione dell'albero su più processi, ciascuno dei quali parte da uno dei primi rami creati dalla radice. Una volta che tutti i processi sono giunti al termine dell'esecuzione vengono raccolti in una unica lista i numeri del range già verificati, e similmente il compito di verificare i rimanenti è suddiviso tra più processi (uno o più numeri per ciascuno). Sarebbe possibile introdurre parallelismo nelle stesse operazioni di verifica di un numero, suddividendo su più processi il compito di iterare porzioni dello spazio dell'automa cellulare (ipotizzando che si stia iterando un numero con un numero di cifre estremamente grande), ma sorgerebbe il problema di come gestire le porzioni dello spazio al confine tra due partizioni. Viceversa si potrebbe inserire più di un numero in uno stesso spazio (separandoli da un certo numero di celle nulle) ed applicare le regole per la successiva generazione su tutti i numeri allo stesso tempo, ma in questo caso occorrerebbe prevenire la collisione tra due traiettorie distinte.
Sebbene lo studio di algoritmi basati sul metodo inverso non potrà avere ricadute pratiche oltre a uno studio puramente accademico, abbiamo comunque esplorato varie possibilità e variazioni dell'approccio portando a una grande riduzione del costo computazionale iniziale.

Bibliografia

[1] Judy arrays. http://judy.sourceforge.net/.

[2] Sequenza oeis a005186. https://oeis.org/A005186.

[3] Sequenza oeis a006577. https://oeis.org/A006577.

[4] M. Bruschi. Two cellular automata for the 3x+1 map, 2005.

[5] Sitan Chen. Cellular automata to more efficiently compute the collatz map, 2013.

[6] Harold Scott MacDonald Coxeter. Cyclic sequences and frieze patterns (the fourth felix behrend memorial lecture), 1971.

[7] C. J. Everett. Iteration of the number theoretic function f(2n) = n, f(2n + 1) = 3n + 2, 1977.

[8] Helmut Hasse. Unsolved problems in elementary number theory, 1975. Lezioni presso Università del Maine (Orono), appunti ciclostilati.

[9] Brian Hayes. Computer recreations: The ups and downs of hailstone numbers. 01 1984.

[10] Jeffrey Lagarias. The 3x+1 problem and its generalizations. 1985.

[11] Jeffrey Lagarias. The 3x+1 problem: An overview. https://www.researchgate.net/publication/228386392_The_3x_1_Problem_An_Overview, 2010.

[12] Eric Roosendaal. On the 3x+1 problem. http://www.ericr.nl/wondrous/.

[13] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.