UNIVERSITA' degli STUDI di ROMA
TOR VERGATA

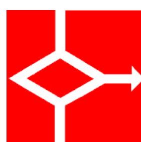
Gara di Allenamento TOR Vergata

(GATOR)

Terza Edizione, 2016

Testi e soluzioni ufficiali

Ministero dell'Istruzione dell'Università e Ricerca



AICA
Associazione Italiana per l'Informatica ed il Calcolo Automatico

Problemi a cura di
Vincenzo Bonifaci, Giuseppe F. Italiano, Luigi Laura

Coordinamento
Monica Gati

Logo della gara
Manuela Pattarini

Testi dei problemi
Gabriele Farina, William Di Luigi, Giuseppe F. Italiano, Luigi Laura, Luca Versari

Soluzioni dei problemi
William Di Luigi, Gabriele Farina, Luca Versari

Gestione gara online
William Di Luigi, Gabriele Farina, Luca Versari

Sistema di gara
Contest Management System (CMS)[1]

[1] http://cms-dev.github.io

Introduzione

La terza edizione della Gara di Allenamento TOR vergata (GATOR) si è svolta sabato 9 e domenica 10 aprile 2016 (http://people.uniroma2.it/giuseppe.italiano/gator/). La GATOR è una gara online di programmazione, con 4 problemi da svolgere in 5 ore. I problemi sono stati concepiti in modo da poter essere affrontati dagli studenti delle scuole secondarie superiori selezionati per la fase Territoriale delle Olimpiadi di Informatica. Ben 165 persone da tutta Italia si sono registrate alla terza edizione della GATOR; di queste, 64 hanno ottenuto punti su almeno uno dei quattro problemi proposti.

La GATOR è una iniziativa svolta nell'ambito della convenzione tra il Comitato Italiano delle Olimpiadi di Informatica e il Dipartimento di Ingegneria Civile e Ingegneria Informatica dell'Università di Roma "Tor Vergata. Le Olimpiadi di Informatica sono nate con l'intento di selezionare e formare, ogni anno, una squadra di atleti che rappresenti il nostro paese alle International Olympiad in Informatics (IOI), indette dall'UNESCO fin dal 1989. L'organizzazione delle Olimpiadi di Informatica è gestita dal Ministero dell'Istruzione, dell'Università e della Ricerca e dall'AICA, con l'obiettivo primario di stimolare l'interesse dei giovani verso la scienza dell'informazione e le tecnologie informatiche.

In questo documento trovate i testi dei quattro problemi proposti nella gara e una proposta di soluzione.

Ringraziamo Monica Gati, William Di Luigi, Gabriele Farina e Luca Versari per la loro collaborazione, senza la quale non sarebbe stato possibile organizzare la GATOR. Ringraziamo inoltre Manuela Pattarini, autrice del logo.

Giuseppe F. Italiano e Luigi Laura

Sequenza di Pollatz (pcollatz)

Limite di tempo:	1.0 secondi
Limite di memoria:	512 MiB
Difficoltà:	2

Consideriamo il seguente algoritmo, che prende in ingresso un intero positivo N:

1. Se N vale 1, l'algoritmo termina.
2. Se N è pari, dividi N per 2, altrimenti (se N è dispari) moltiplicalo per 3 e aggiungi 1.

Per esempio, applicato al valore $N = 6$, l'algoritmo produce la seguente sequenza (di lunghezza 9, contando anche il valore iniziale $N = 6$ e il valore finale 1):

$$6, 3, 10, 5, 16, 8, 4, 2, 1.$$

La congettura di Collatz, chiamata anche congettura $3N + 1$, afferma che l'algoritmo qui sopra termini sempre per qualsiasi valore N; in altri termini, se prendo un qualsiasi numero intero maggiore di 1 applicare la regola numero 2 conduce sempre al numero 1. È riferendosi a questa celebre congettura che il famoso matematico Erdős ha detto "*La matematica non è ancora pronta per problemi di questo tipo*".

Patrizio ha creato una versione alternativa della sequenza, secondo la seguente regola:

1. Se N vale 1, l'algoritmo termina.
2. Se N è pari, dividi N per 2, altrimenti (se N è dispari) moltiplicalo per **5** e aggiungi 1.

Patrizio ha chiamato la sua versione *sequenza di Pollatz*, e si è accorto che, per alcuni numeri la sua sequenza è più corta di quella originale; per esempio, applicata al numero 6, otteniamo:

$$6, 3, 16, 8, 4, 2, 1.$$

ovvero una sequenza di lunghezza 7. Allo stesso tempo, però, Patrizio si è anche accorto che per altri numeri **la sequenza non termina mai**. Il vostro compito è quello di aiutare Patrizio a calcolare quante volte la sequenza di Pollatz termina generando una sequenza di numeri strettamente più corta di quella di Collatz.

Dati di input

Il file `input.txt` è composto da una riga di testo, contentente i due interi A e B ($A \leq B$).

Dati di output

Il file `output.txt` è composto da una sola riga contenente un intero positivo: quanto sono gli N ($A \leq N \leq B$) per cui la lunghezza della sequenza di Pollatz calcolata a partire da N termina e ha lunghezza strettamente minore della corrispondente sequenza di Collatz.

Assunzioni

- $1 \leq A \leq B \leq 10\,000$.

Esempi di input/output

input.txt	output.txt
1 5	1
50 60	2

Spiegazione

Nel primo caso di esempio, le lunghezze delle sequenze di Pollatz e di Collatz coincidono per $N = 1, 2, 4$. Per $N = 3$ la sequenza di Pollatz $(3, 16, 8, 4, 2, 1)$ è strettamente più corta di quella di Collatz $(3, 10, 5, 16, 8, 4, 2, 1)$. Per $N = 5$ la sequenza di Pollatz non termina mai.

Soluzione

Per risolvere il problema è sufficiente implementare le semplici regole di evoluzione delle sequenze di Collatz e di Pollatz, come descritte nel testo. Per ogni elemento iniziale N compreso tra A e B provvederemo a simulare le sequenze, e controllare quale delle due termina prima.

Per ovviare al caso in cui la sequenza di Pollatz potrebbe non terminare, conviene partire dal calcolo della lunghezza C della sequenza di Collatz (che è garantito terminare per $N \leq 10\,000$), e solo dopo calcolare i termini della serie di Pollatz. Qualora la sequenza di Pollatz non fosse terminata entro C passi, si può concludere che la proprietà richiesta dal testo non vale, e passare al valore di N successivo.

Esempio di codice C++11

```
1  #include <iostream>
2  #include <cstdio>
3
4  int collatz(int n) {
5    int lunghezza = 0;
6    while (n > 1) {
7      ++lunghezza;
8      if (n % 2 == 0)
9        n /= 2;
10     else
11       n = 3 * n + 1;
12   }
13   return lunghezza;
14 }
15
16 int pollatz(int n, int limite_lunghezza) {
17   int lunghezza = 0;
18   while (n > 1 && lunghezza < limite_lunghezza) {
19     ++lunghezza;
20     if (n % 2 == 0)
21       n /= 2;
22     else
23       n = 5 * n + 1;
24   }
25   return lunghezza;
26 }
27
28 int main() {
29   freopen("input.txt", "r", stdin);
30   freopen("output.txt", "w", stdout);
31
32   int A, B;
33   std::cin >> A >> B;
34
35   int risposta = 0;
36   for (int N = A; N <= B; ++N) {
37     int lunghezza_collatz = collatz(N);
38     int lunghezza_pollatz = pollatz(N, lunghezza_collatz);
39
40     if (lunghezza_pollatz < lunghezza_collatz)
41       ++risposta;
42   }
43
44   std::cout << risposta << std::endl;
45 }
```

Somme costose (somme)

Limite di tempo:	1.0 secondi
Limite di memoria:	512 MiB
Difficoltà:	2

Gabriele sta studiando il seguente problema. Deve sommare dei numeri tra di loro, ma ogni volta può sommare solo una coppia di numeri, e il **costo** di questa operazione è pari al risultato della somma.

Ad esempio, volendo sommare 1, 2 e 3, ci sono tre possibilità:

1. Sommiamo $(1 + 2) + 3 = 3 + 3 = 6$: la prima operazione $(1 + 2)$ ci costa 3, la seconda $(3 + 3)$ ci costa 6, il totale per fare questa sequenza di somme è quindi 9.
2. Sommiamo $(1 + 3) + 2 = 4 + 2 = 6$: la prima operazione $(1 + 3)$ ci costa 4, la seconda $(4 + 2)$ ci costa 6, il totale per fare questa sequenza di somme è quindi 10.
3. Sommiamo $(2 + 3) + 1 = 5 + 1 = 6$: la prima operazione $(2 + 3)$ ci costa 5, la seconda $(5 + 1)$ ci costa 6, il totale per fare questa sequenza di somme è quindi 11.

Il vostro compito, dato un insieme di numeri da sommare, è quello di trovare il costo minimo per sommarli tutti tra di loro. Ad esempio, come visto sopra, se l'insieme dei numeri da sommare è 1, 2 e 3, il costo minimo per sommarli è 9.

Dati di input

Il file `input.txt` contiene due righe. La prima riga contiene l'intero N, il numero di elementi da sommare. La seconda riga contiene N interi $a_1, \ldots, a_N$ separati da uno spazio, corrispondenti alla lista dei numeri da sommare tra loro.

Dati di output

Sul file `output.txt` stampare il costo minimo necessario per sommare tra di loro tutti i numeri.

Assunzioni

- $2 \leq N \leq 100\,000$.
- $1 \leq a_i \leq 1000$ per ogni $i = 1, \ldots, N$.

Esempi di input/output

input.txt	output.txt
3 2 1 3	9
5 10 7 2 9 3	67

Soluzione

È possibile dimostrare che conviene sempre sommare tra di loro i numeri più piccoli a disposizione. Per determinare velocemente la coppia di numeri da sommare, usiamo una coda di priorità basata su heap.

Esempio di codice C++11

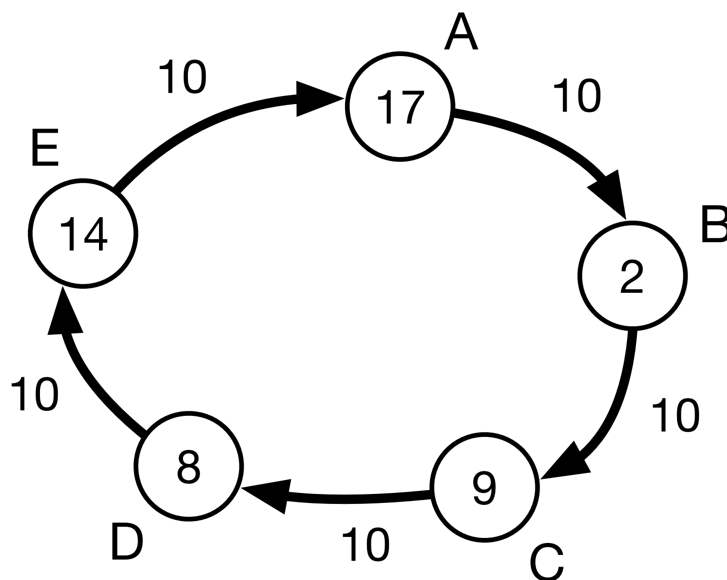
```
 1  #include <iostream>
 2  #include <cstdio>
 3  #include <queue>
 4
 5  int main() {
 6    freopen("input.txt", "r", stdin);
 7    freopen("output.txt", "w", stdout);
 8
 9    int N;
10    std::priority_queue<int, std::vector<int>, std::greater<int>> A;
11
12    std::cin >> N;
13    for (int i = 0; i < N; ++i) {
14      int valore;
15      std::cin >> valore;
16
17      A.push(valore);
18    }
19
20    int risposta = 0;
21
22    while (A.size() > 1) {
23      int primo_addendo = A.top();
24      A.pop();
25
26      int secondo_addendo = A.top();
27      A.pop();
28
29      risposta += secondo_addendo + primo_addendo;
30      A.push(secondo_addendo + primo_addendo);
31    }
32
33    std::cout << risposta << std::endl;
34  }
```

Tesla Motors (`tesla`)

Limite di tempo: 1.0 secondi
Limite di memoria: 512 MiB
Difficoltà: 3

Le macchine Tesla sono auto elettriche dalla linea sportiva. Francesco sa che a breve si diffonderanno anche in Italia e vuole costruire una rete di punti di ricarica a batteria, scollegati quindi dalla rete elettrica.

Francesco ha costruito diversi impianti. Gli impianti sono collegati tra di loro in un circuito: da un impianto al successivo ci sono 10 km da percorrere, come nell'esempio mostrato qui sotto con $N = 5$ stazioni. Purtroppo, per un errore nella consegna, la somma delle energie presenti in tutte le stazioni è esattamente pari alla distanza necessaria per percorrere il circuito. In figura è mostrato un esempio in cui in ogni stazione è mostrata la quantità di energia presente (e la somma di tutte le energie è pari a 50): ad esempio, partendo dalla stazione A si ha energia per percorrere 17 km, però si arriva alla stazione B con energia per percorrere ancora 7 km, che unita alle 2 unità di energia nella stazione B non consentono di arrivare alla stazione C.



Nell'esempio mostrato in figura qui sopra, partendo dalla stazione E è possibile fare un circolo completo (si arriva in A con 4 unità residue, in B con 11 unità residue, in C con tre unità residue, in D con due unità residue e infine si torna in E.

Il vostro compito è quello di scrivere un programma che, ricevuta la rappresentazione di un circuito con N stazioni, vi dica qual è stazione da cui è possibile partire per fare il giro completo.

Dati di input

Il file `input.txt` è composto da due righe. La prima riga contiene il numero N di impianti. La seconda riga contiene N interi $a_1, \ldots, a_N$ separati da uno spazio, rappresentanti le quantità di energia disponibili nelle stazioni $1, \ldots, N$ rispettivamente. È sempre garantito che gli a_i sommano al valore $10N$.

Dati di output

Il file `output.txt` è composto da una sola riga contenente l'indice di una stazione da cui è possibile partire per completare il circuito. Nel caso in cui esista più di una risposta corretta, stamparne una qualsiasi.

Assunzioni

- $10 \leq N \leq 100\,000$.
- $0 \leq a_i \leq N$ per ogni $i = 1, \ldots, N$.
- Le stazioni sono numerate con i numeri da 1 a N.
- È sempre garantito che gli a_i sommano al valore $10N$.

Esempi di input/output

input.txt	output.txt
5 17 2 9 8 14	5
6 17 4 5 3 26 5	5

Soluzione

Per risolvere il problema è sufficiente provare a partire da tutte le possibili stazioni e vedere se una di queste soddisfa la condizione.

Questo porta a un algoritmo quadratico nel numero delle stazioni, ovvero un algoritmo che impiega un numero di passi proporzionale al quadrato del numero delle stazioni: per ogni stazione infatti proviamo a verificare se questa possa andare bene come punto di partenza, e visitiamo tutte le stazioni; se abbiamo N stazioni facciamo circa N passi per ognuna di esse.

Una soluzione del genere va benissimo a livello di prova territoriale delle olimpiadi, mentre qui di seguito forniamo una soluzione lineare, ovvero che impiega un numero proporzionale al numero di stazioni (una soluzione del genere va bene a livello di finale nazionale).

Soluzione lineare

Iniziamo fornendo una dimostrazione costruttiva del fatto che è sempre possibile trovare una stazione da cui partire per completare con successo un ciclo completo.

A tal fine, introduciamo

$$l(i) \triangleq \sum_{j=1}^{i}(a_j - 10),$$

ovvero la quantità di energia rimasta nella macchina prima di arrivare alla stazione $i + 1$, supponendo di partire dalla stazione 1. Notiamo che $l(N) = 0$.

Introduciamo anche l'indice

$$S \triangleq \arg\min_{1 \leq i \leq N} l(i),$$

dove assumiamo di rompere le parità scegliendo il minimo S con la proprietà indicata. Dividiamo ora l'analisi in due casi:

- Se $S = N$, allora partendo dalla stazione 1 riusciamo a completare il giro. Infatti, per la definizione di S deve essere $l(i) \geq l(S) = 0$ per ogni indice di stazione $i = 1, \ldots, N$. Ma per definizione della funzione l questo significhe che la quantità di energia rimasta nella macchina appena prima di giugngere ad ogni stazione è sempre non-negativa, ed è possibile completare il giro con successo.
- Se $S < N$, allora partendo dalla stazione $S + 1$ riusciamo a completare il giro. Infatti, partendo da $S + 1$ la quantità di energia residua appena prima di giungere alla stazione i ($S + 1 < i \leq N$) è pari a

  $$q_a(i) \triangleq \sum_{j=S+1}^{i}(a_j - 10) = l(i) - l(S).$$

  Questo numero è non-negativo per definizione di S.

  Analogamente, la quantità residua prima di giungere alla stazione i ($1 \leq i < S + 1$) è

  $$q_b(i) \triangleq \sum_{j=S+1}^{N}(a_j - 10) + \sum_{j=1}^{i}(a_j - 10) = \sum_{j=1}^{N}(a_j - 10) - \sum_{j=1}^{S}(a_j - 10) + \sum_{j=1}^{i}(a_j - 10) = l(N) + l(i) - l(S).$$

  D'altra parte, $l(N) = 0$ e per definizione di S si conclude che $q_b(i)$ è non-negativo.

La dimostrazione costruttiva appena esposta può essere convertita direttamente in codice, ottenendo come già anticipato un algoritmo di complessità lineare nella dimensione dell'input.

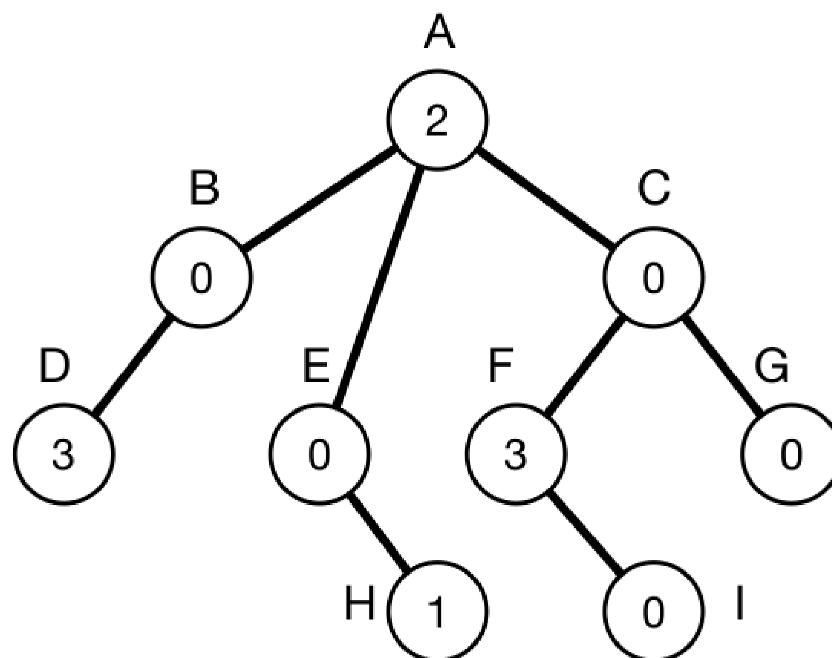
Esempio di codice C++11

```
#include <iostream>
#include <cstdio>
#include <vector>

int main() {
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);

  int N;
  std::vector<int> A;

  std::cin >> N;
  for (int i = 0; i < N; ++i) {
    int a;
    std::cin >> a;
    A.push_back(a);
  }

  int S = N - 1, l_di_S = 0, l_di_i = 0;
  for (int i = 0; i < N; ++i) {
    l_di_i += A[i] - 10;
    if (l_di_i < l_di_S) {
      S = i;
      l_di_S = l_di_i;
    }
  }

  std::cout << ((S + 1) % N + 1) << std::endl;
}
```

Monete a posto (monete)

Limite di tempo: 1.0 secondi
Limite di memoria: 512 MiB
Difficoltà: 3

William colleziona monete, e le tiene in un espositore come quello mostrato in figura, in cui ci sono N contenitori numerati da 1 a N appesi uno all'altro tramite una barra di metallo. In particolare, ogni contenitore tranne quello più in alto (il contenitore numero 1) è appeso a esattamente un altro contenitore. Sua sorella ha giocato con le monete, che prima erano una in ogni posizione, e le ha lasciate nella situazione mostrata qui sotto. William ha però una regola per rimetterle a posto: può spostare una sola moneta per volta, da un contenitore a uno collegato. Quante operazioni sono sufficienti a William per fare in modo che in ogni contenitore ci sia una moneta?



Ad esempio, nella situazione mostrata qui sopra, William impiega:

- Una operazione per spostare una moneta da A a E.
- Una operazione per spostare una moneta da F a I.
- Una operazione per spostare una moneta da F a C.
- Una operazione per spostare una moneta da D a B.
- Quattro operazioni per spostare una moneta da D a G (passando per B, A e infine C).

Dall'esempio qui sopra si vede che servono otto operazioni in totale per rimettere a posto le monete, con una moneta per ogni contenitore.

Dati di input

Il file `input.txt` contiene tre righe. La prima riga contiene l'intero N, il numero di contenitori (e di monete) che ha William. La seconda riga contiene i numeri interi $a_1, \ldots, a_N$, che rappresentano il numero di monete contenute nel rispettivo contenitore dopo che queste sono state disordinate. Infine la terza riga contiene $N - 1$ interi $b_2, \ldots, b_N$ che indicano che il contenitore j è appeso al contenitore b_j.

Dati di output

Sul file `output.txt` stampare il numero minimo di spostamenti necessari a William per rimettere a posto tutte le monete.

Assunzioni

- $1 \leq N \leq 100\,000$.
- $0 \leq a_i \leq N$ per ogni $i = 1, \ldots, N$.
- $a_1 + \cdots + a_N = N$.
- $1 \leq b_i < i$ per ogni $i = 1, \ldots, N$.

Esempi di input/output

input.txt	output.txt
9 2 0 0 3 0 3 0 1 0 1 1 2 1 3 3 5 6	8
10 0 0 1 2 1 1 2 0 3 0 1 2 2 2 4 3 5 1 8	11

Soluzione

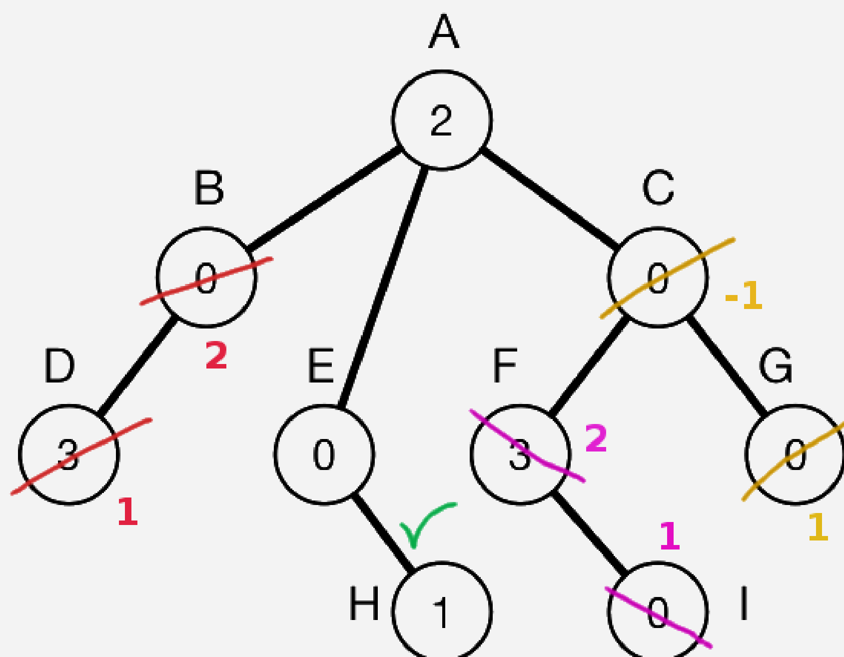
☞ Questo problema ha una difficoltà superiore a quella dei problemi che vengono dati alle prove territoriali; sicuramente è più adatto a una finale nazionale.

Un modo per arrivare alla soluzione è partire dalle foglie (ovvero i nodi nell'albero che non hanno "figli"). Se una foglia contiene 2 o più monete, è facile convincersi che la cosa migliore da fare è spostare tutte le monete "di troppo" un livello verso l'alto (per poi essere eventualmente spostate di nuovo, se necessario). Se una foglia contiene esattamente una moneta allora quella foglia è a posto così, conviene ignorarla.

Fin qui sembrerebbe che per risolvere il problema basti spostare sui "padri" le monete di troppo delle foglie, facendo attenzione alle foglie che hanno esattamente una moneta. Ma cosa succede se una foglia contiene 0 monete? In quel caso la foglia dovrà prendere una moneta dal padre, il quale però potrebbe a sua volta avere 0 monete!

Per aggirare questa complicazione possiamo introdurre una specie di "debito": permetteremo che i nodi abbiano un valore **negativo** di monete. In questo modo, se una foglia ha 0 monete, potrà facilmente ottenerne una dal padre anche qualora il padre avesse 0 monete: infatti, basterebbe portare a -1 il numero di monete del padre.

Consideriamo le foglie presenti nella figura del testo del problema, ovvero i nodi: D, H, I, G. La prima foglia contiene 3 monete, quindi darà 2 monete al padre (nodo B). La seconda foglia contiene una sola moneta, quindi la saltiamo. La terza foglia contiene 0 monete, ma basta prenderne una dal padre (nodo F) che ne ha 3. L'ultima foglia ricade nel caso discusso prima: ha 0 monete, e suo padre ha anch'esso 0 monete. Come spiegato, basta prendere una moneta dal padre, che (indebitandosi) passa a -1 monete.



Una volta eseguiti gli scambi in figura, tutte le foglie saranno "sistemate" (e nel minor numero di passaggi). Sarà quindi sufficiente rimuoverle dall'albero e ripetere gli scambi con le nuove foglie che verranno generate da tale rimozione (i nodi: B, E, F) e così via fino a che non rimane solo la radice.

Questa soluzione bastava per prendere 85 punti su 100, mentre negli ultimi casi risultava troppo lenta.

Infatti, rimuovendo di volta in volta le foglie dell'albero e ripetendo la procedura, in certi casi di input potrebbe capitare di ripetere troppe volte (ad esempio se ogni volta viene rimossa una sola foglia per N volte) andando fuori tempo limite.

Per implementare la soluzione in modo più efficiente, possiamo gestire in modo furbo la ricerca del "prossimo" insieme di foglie (dopo la rimozione di quelle vecchie). Per fare ciò, basta fare una visita in profondità dell'albero e (man mano che si risalgono i nodi) applicare gli scambi al nodo corrente (che sarà una foglia). La differenza principale rispetto all'approccio "lento" è l'ordine in cui visitiamo le foglie: con la visita in profondità vengono visitate prima tutte le foglie di un sottoalbero e poi tutte quelle degli altri sottoalberi. È facile convincersi però che l'ordine non è importante.

Esempio di codice C++11

```
1  #include <iostream>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <vector>
5
6  std::vector<int> monete;
7  std::vector<std::vector<int>> figli;
8
9  int dfs(int u, int parent) {
10     int res = 0;
11
12     // visita (e "rimuovi") eventuali figli
13     for (int v: figli[u])
14         res += dfs(v, u);
15
16     // ora sono su una foglia - è necessario fare spostamenti di monete?
17     if (monete[u] != 1) {
18         // prendi dal padre (anche a costo di indebitarlo)
19         monete[parent] += monete[u] - 1;
20         // registra la spesa nel totale
21         res += std::abs(monete[u] - 1);
22     }
23     return res;
24 }
25
26 int main() {
27     freopen("input.txt", "r", stdin);
28     freopen("output.txt", "w", stdout);
29
30     int N;
31     std::cin >> N;
32     monete.resize(N);
33     figli.resize(N);
34     for (int i=0; i<N; i++)
35         std::cin >> monete[i];
36
37     for (int i=1; i<N; i++) {
38         int p;
39         std::cin >> p;
40         figli[p - 1].push_back(i);
41     }
42
43     std::cout << dfs(0, 0) << std::endl;
44 }
```