Secchi di noccioline (noccioline)

Limite di tempo: 0.2 secondi
Limite di memoria: 64 MiB

La Pinats S.p.A. è uno dei principali esportatori galattici di deliziose noccioline. Il processo di produzione prevede che le noccioline vengano prelevate dal grande deposito, e poi inscatolate nei secchi di noccioline[1] destinate alla vendita. Ogni confezione prodotta contiene esattamente K preziose noccioline, tostate e salate con rarissimo sale dell'Himalaya.

Purtroppo la macchina che versa le noccioline nelle confezioni si è guastata, e N secchi sono stati riempiti con un numero casuale di noccioline. Per risolvere il problema è stata assunta una squadra di 150 ingegneri, che hanno prima di tutto contato, per ogni secchio, il numero di noccioline contenute. È stata installata poi una macchina robotizzata in grado di prelevare una singola nocciolina da un secchio e lasciarla delicatamente cadere in un altro secchio. A richiesta la macchina può anche prelevare o scaricare una nocciolina nel grande deposito. Tuttavia la macchina richiede di essere programmata da un tecnico, e si pone ora il problema di come risolvere il problema nel minor tempo possibile. Dato il numero di secchi e le quantità di noccioline in essi contenute, scrivi un programma che determini il numero minimo di operazioni che il robot deve compiere affinché alla fine tutti i secchi contengano K noccioline ciascuno.

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`noccioline.c`, `noccioline.cpp`, `noccioline.pas`) con un esempio di implementazione.

Se usi C/C++, dovrai implementare la funzione `ContaOperazioni` utilizzando il seguente prototipo:

```
int ContaOperazioni(int N, int K, int* secchi);
```

Se invece usi Pascal, dovrai usare il seguente prototipo:

```
function ContaOperazioni(N: longint; K: longint; var secchi: array of longint): longint;
```

N è il numero di secchi, mentre `secchi`$[i]$ contiene, per ogni $0 \leq i < N$, il numero di noccioline contenute nel secchio i. La funzione deve restituire il numero minimo di operazioni che la macchina dovrà effettuare affinché alla fine del processo tutti i secchi contengano K noccioline ciascuno.

Assunzioni

- $1 \leq N \leq 10\,000$
- $1 \leq K \leq 100\,000$
- Ogni secchio contiene inizialmente un numero di noccioline non superiore a $100\,000$
- Il grande deposito è in grado di fornire e ricevere un numero infinito di noccioline

[1] Un'indagine di mercato ha determinato che vendere le noccioline a secchi non solo beneficia i volumi delle vendite, ma riduce anche sensibilmente il numero di suicidi collegati al tragico e improvviso esaurimento di noccioline, spesso dovuto alle piccole dimensioni della confezione.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1 [0 punti]:** Casi d'esempio.
- **Subtask 2 [16 punti]:** $N \leq 10$.
- **Subtask 3 [24 punti]:** Tutti i secchi contengono K noccioline, tranne due.
- **Subtask 4 [30 punti]:** La somma di tutte le noccioline presenti nei secchi è esattamente $N \cdot K$.
- **Subtask 5 [30 punti]:** Nessuna limitazione specifica.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, a quel punto chiama la funzione `ContaOperazioni` che dovete implementare, e scrive il risultato restituito dalla vostra funzione sul file `output.txt`.

- Riga 1: contiene gli interi N e K.
- Riga 2: contiene N interi, di cui l'i-esimo rappresenta il numero di noccioline presenti nel secchio i.

Esempi di input/output

input.txt	output.txt
5 4 1 0 7 9 5	9
3 3 1 2 1	5

Spiegazione

Nel **primo caso d'esempio** una configurazione ottimale del robot prevede che le tre noccioline in eccesso nel terzo secchio vengano spostate (con tre viaggi) nel primo secchio. Successivamente 4 noccioline dal quarto secchio vengono spostate (con quattro viaggi) nel secondo secchio. Rimangono ora due secchi con 5 noccioline ciascuno. Con due viaggi si trasportano le due noccioline in eccesso nel grande deposito.

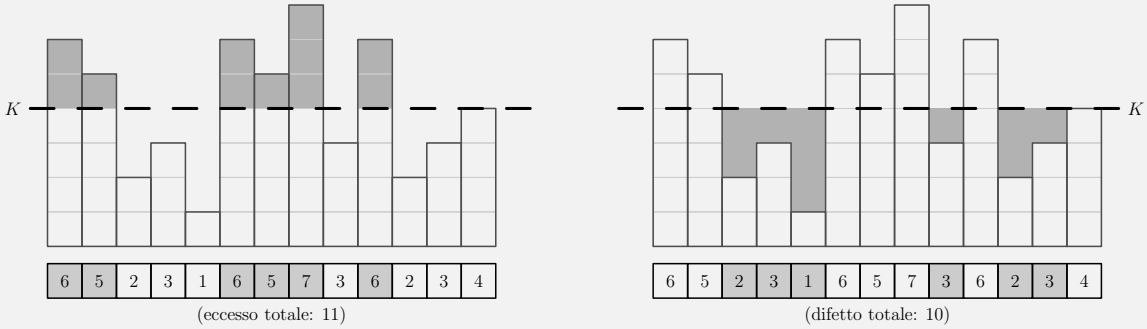
Nel **secondo caso d'esempio** è necessario trasportare 5 noccioline dal grande deposito ai vari secchi, per un totale di 5 operazioni.

Soluzione

Analizziamo i secchi: se un secchio contiene più di K noccioline diremo che ha un *eccesso di noccioline* pari alla differenza tra le noccioline che contiene e K, mentre se contiene meno di K noccioline diremo che ha un *difetto di noccioline* pari alla differenza tra K e la quantità di noccioline contenute.

In altre parole, se un secchio contiene più noccioline del dovuto allora il suo eccesso di noccioline è il numero di noccioline da asportare affinché alla fine contenga K noccioline, mentre se ne contiene meno del dovuto allora il suo difetto di noccioline è il numero di noccioline da versare affinché alla fine contenga K noccioline.

Chiamiamo *eccesso totale* la somma degli eventuali eccessi dei secchi, mentre *difetto totale* la somma degli eventuali difetti dei secchi. Consideriamo un piccolo esempio per fissare il concetto:



Dimostreremo che la risposta al problema è pari al massimo tra l'eccesso e il difetto totale iniziali dei secchi (nel caso dell'immagine sopra, la risposta sarebbe 11). Per dimostrare che effettivamente quel valore rappresenta il minimo numero di operazioni da compiere, dapprima mostreremo che è necessario compiere almeno quel numero di mosse e successivamente che quel numero di mosse è sufficiente.

Iniziamo a mostrare che qualunque sequenza di mosse che svolga il compito necessita di almeno quel numero di passaggi. Supponiamo per semplicità che l'eccesso totale inziale sia superiore o uguale al difetto totale iniziale. Analizziamo i secchi con un eccesso di noccioline: sicuramente tutte quelle noccioline in eccesso dovranno essere spostate prima o poi, in altri secchi o eventualmente nel grande deposito.

Mostriamo ora che in effetti quel numero di mosse è anche sufficiente, oltre che necessario. Come prima, supponiamo senza perdita di generalità che l'eccesso totale iniziale sia maggiore o uguale al difetto totale iniziale. Vogliamo quindi dimostrare che con un numero di mosse pari all'eccesso totale iniziale è possibile portare tutti i secchi a contenere esattamente K noccioline.

Possiamo trasportare l'eccesso iniziale di noccioline nei secchi in difetto, colmandone la differenza. Poiché il difetto totale iniziale è per ipotesi minore o uguale all'eccesso totale iniziale, dopo un numero di mosse pari al difetto totale iniziale nessun secchio sarà più in difetto. A questo punto portiamo le noccioline ancora in eccesso, se presenti, nel grande deposito, raggiungendo la situazione desiderata. Il numero di mosse eseguite è pari a

$$\underbrace{(\text{difetto totale iniziale})}_{\text{per colmare i vasi in difetto}} + \underbrace{(\text{eccesso totale iniziale} - \text{difetto totale iniziale})}_{\text{da trasportare nel grande deposito}} = (\text{eccesso totale iniziale})$$

Abbiamo quindi dimostrato che nel caso in cui l'eccesso totale iniziale sie maggiore o uguale al difetto totale iniziale il minimo numero di operazioni da compiere è pari all'eccesso totale iniziale. Con un ragionamente del tutto analogo si dimostra che se il difetto totale iniziale è maggiore o uguale all'eccesso totale iniziale allora il numero minimo di mosse da compiere è pari al difetto totale iniziale dei secchi.

Esempio di codice C++

```cpp
int ContaOperazioni(int N, int K, int *secchi) {
    int eccesso_totale = 0;
    int difetto_totale = 0;

    for (int i = 0; i < N; i++) {
        if (secchi[i] > K)
            eccesso_totale += secchi[i] - K;
        else if (secchi[i] < K)
            difetto_totale += K - secchi[i];
    }

    if (difetto_totale >= eccesso_totale)
        return difetto_totale;
    else
        return eccesso_totale;
}
```

Skyline (`skyline`)

Limite di tempo:	1.0 secondi
Limite di memoria:	64 MiB

Nella città di Yendys ci sono N altissimi grattacieli.

Essi costituiscono la principale attrazione turistica del luogo: sono innumerevoli, ogni anno, i turisti che passano da Yendys per un preciso motivo: ammirare lo *skyline* della città. Però recentemente si assiste a un calo del turismo, a causa della crisi economica. Per risolvere il problema, siccome i turisti sono attratti dalle novità, si è deciso di rinnovare lo skyline, modificando l'altezza dei grattacieli. Quindi è stato indetto un concorso per permettere a chiunque di presentare il proprio progetto di rinnovamento.

Ogni progetto, per essere ammesso al concorso, deve rispettare i seguenti *requisiti di ammissione*:

- deve prevedere N grattacieli, per comodità numerati da 0 a $N-1$;
- per ogni grattacielo i, deve essere specificato il numero di piani, indicato con $\texttt{H}[i]$;
- ogni grattacielo deve avere almeno un piano, e meno di 2 000 000 piani;
- non ci possono essere due grattacieli con lo stesso numero di piani, altrimenti il panorama diventerebbe noioso.

Molti tra i più famosi architetti del mondo hanno presentato i loro progetti. Essi devono essere valutati, in modo che sia più facile scegliere il migliore. Per effettuare la valutazione si è deciso di assegnare ad ogni progetto un valore, detto *grado Ilazimi* in onore dell'inventore di questo metro di valutazione. Il grado Ilazimi di un progetto è uguale al numero di terne (A, B, C) di numeri interi che rispettano contemporaneamente le seguenti condizioni:

- $0 \leq A < B < C \leq N-1$
- $\texttt{H}[A] = \max\{\texttt{H}[0], \texttt{H}[1], \ldots, \texttt{H}[B-1], \texttt{H}[B]\}$
- $\texttt{H}[C] = \max\{\texttt{H}[B], \texttt{H}[B+1], \ldots, \texttt{H}[N-2], \texttt{H}[N-1]\}$

Inoltre, siccome verrà assegnato un sostanzioso premio al progetto migliore, anche tu hai deciso di partecipare al concorso. Voci di corridoio dicono che, per garantire un bell'aspetto allo skyline, è preferibile che il grado Ilazimi del progetto sia uguale a K.

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`skyline.c`, `skyline.cpp`, `skyline.pas`) con un esempio di implementazione.

Implementazione della funzione `valuta`

Se usi C/C++, dovrai implementare la funzione `valuta` utilizzando il seguente prototipo:

```
int valuta(int N, int *H);
```

Se invece usi Pascal, dovrai usare il seguente prototipo:

```
function valuta(N: longint; var H: array of longint): longint;
```

- N è il numero di grattacieli.
- `H`$[i]$ indica il numero di piani del grattacielo i. $(0 \leq i \leq N-1)$

La funzione `valuta` dovrà calcolare e restituire come valore di ritorno un intero: il grado Ilazimi relativo al progetto.

Implementazione della funzione `progetta`

Se usi C/C++, dovrai implementare la funzione `progetta` utilizzando il seguente prototipo:

```
void progetta(int N, int K, int *H);
```

Se invece usi Pascal, dovrai usare il seguente prototipo:

```
procedure progetta(N: longint; K: longint; var H: array of longint);
```

- N è il numero di grattacieli che deve avere il tuo progetto.
- K è il grado Ilazimi che deve avere il tuo progetto.

La procedura `progetta` dovrà riempire l'array `H` con le altezze dei grattacieli del tuo progetto.
In particolare bisogna memorizzare in `H`$[i]$ il numero di piani del grattacielo i. $(0 \leq i \leq N-1)$
Il tuo progetto deve soddisfare tutti i requisiti di ammissione e il suo grado Ilazimi deve essere esattamente uguale a K.

Assunzioni

Ogni input ufficiale verificherà le seguenti condizioni:

- $1 \leq N \leq 1\,000\,000$.
- Ogni progetto di skyline passato alla funzione `valuta` rispetta i requisiti di ammissione.
- $K \geq 0$. Inoltre ogni chiamata alla funzione `progetta` consente almeno una soluzione.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1 [0 punti]:** Casi d'esempio.
- **Subtask 2 [16 punti]:** $N \leq 300$. Solo la funzione `valuta` verrà chiamata.
- **Subtask 3 [25 punti]:** $N \leq 5\,000$. Solo la funzione `valuta` verrà chiamata.
- **Subtask 4 [28 punti]:** Solo la funzione `valuta` verrà chiamata.
- **Subtask 5 [31 punti]:** Solo la funzione `progetta` verrà chiamata.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, a quel punto chiama la funzione `valuta` o `progetta` che dovete implementare, e scrive il risultato restituito dalla vostra funzione sul file `output.txt`.

I test si dividono in due categorie: test di valutazione (che chiamano solo la funzione `valuta`) e test di progettazione (che chiamano solo la funzione `progetta`).

Nel caso vogliate generare un input per un test di valutazione, il file `input.txt` deve avere questo formato:

- La riga 1 contiene il numero 0.
- La riga 2 contiene il numero N.
- La riga 3 contiene N interi separati da uno spazio: $\texttt{H}[0], \texttt{H}[1], \ldots, \texttt{H}[N-2], \texttt{H}[N-1]$.

Il grader stamperà su `output.txt` il valore di ritorno della funzione `valuta`.

Nel caso vogliate generare un test di progettazione, il file `input.txt` deve avere questo formato:

- La riga 1 contiene il numero 1.
- La riga 2 contiene i numeri N e K, separati da uno spazio.

Il grader stamperà su `output.txt` il contenuto dell'array `H`, riempito dalla vostra funzione.

Esempi di input/output

input.txt	output.txt
0 7 8 3 10 18 2 6 11	3
1 7 3	10 5 1 18 9 3 7

Spiegazione

Il primo caso d'esempio è un test di valutazione. Le terne che verificano le condizioni espresse nel testo sono:

- $(0, 1, 3)$, grattacieli con rispettivamente $8, 3, 18$ piani.
- $(3, 4, 6)$, grattacieli con rispettivamente $18, 2, 11$ piani.
- $(3, 5, 6)$, grattacieli con rispettivamente $18, 6, 11$ piani.

Il secondo è un esempio di progettazione. Come si vede, l'output consiste in un array di 7 elementi, e il numero di terne che verificano le condizioni espresse nel testo sono esattamente 3. Anche i requisiti di ammissione sono tutti rispettati.

Note

Se esistono più soluzioni alla progettazione dello skyline, è sufficiente restituirne una qualsiasi.

Soluzione

■ Funzione di valutazione

Definiamo *buone* tutte le terne che soddisfano le condizioni scritte nel testo. La risposta al problema è uguale al numero di terne buone.

Per contare le terne buone fissiamo come elemento centrale della terna un certo B. Si può presentare solamente una di queste due possibilità:

- Non esiste alcuna terna di cui B sia elemento centrale.
- Esiste una terna buona (A, B, C). In questo caso la terna è anche unica: poiché A e C sono gli indici del grattacielo più alto rispettivamente a sinistra e a destra di B, questi sono determinati in maniera univoca, avendo i grattacieli altezze diverse.

Il problema si riduce quindi a determinare, per ogni B, se esiste una terna buona che ammette B come elemento centrale. Notiamo che questo accade solo quando il più alto tra i grattacieli posizionati *prima* di B ha un'altezza maggiore del grattacielo B e, contemporaneamente, il più alto tra i grattacieli posizionati *dopo* di B ha un'altezza maggiore del grattacielo B.

Per effettuare tale controllo conviene precalcolare, per ogni i compreso tra 0 e $N-1$, le quantità:

- $\texttt{SX}[i]$ = massima altezza tra i grattacieli da 0 a i (estremi inclusi)
- $\texttt{DX}[i]$ = massima altezza tra i grattacieli da i a $N-1$ (estremi inclusi)

In questo modo per controllare se esiste una terna buona che ammette B come elemento centrale è sufficiente controllare se $\texttt{SX}[B-1] > \texttt{H}[B]$ e, al contempo, $\texttt{DX}[B+1] > \texttt{H}[B]$.

Rimane il problema di come calcolare le grandezze descritte in modo efficiente. Prima di tutto assegniamo $\texttt{SX}[0] = \texttt{H}[0]$ e $\texttt{DX}[N-1] = \texttt{H}[N-1]$: quando è presente un solo grattacielo, questo è anche di altezza massima. Induttivamente, costruiamo le altre posizioni:

- $\texttt{SX}[i] = \max\{\texttt{SX}[i-1], \texttt{H}[i]\}$: l'altezza del più alto grattacielo nell'intervallo $[0, i]$ è pari al massimo tra l'altezza del più alto grattacielo nell'intervallo $[0, i-1]$, e l'altezza del grattacielo i.
- $\texttt{DX}[i] = \max\{\texttt{DX}[i+1], \texttt{H}[i]\}$: l'altezza del più alto grattacielo nell'intervallo $[i, N-1]$ è pari al massimo tra l'altezza del più alto grattacielo nell'intervallo $[i+1, N-1]$, e l'altezza del grattacielo i.

Con due passate dell'array delle altezze dei grattacieli (di cui una da sinistra a destra e una da destra a sinistra) è quindi possibile determinare gli array SX e DX.

■ Funzione di progettazione

A differenza della funzione di valutazione, la funzione di progettazione può ammettere più di una risposta valida per ogni caso di prova. Mostreremo uno dei tanti modi di assegnare le altezze dei grattacieli, che funziona per ogni coppia N, K in input.

Cominciamo a trattare il caso particolare $K = 0$. Notiamo che una sequenza di grattacieli di altezza crescente non ammette nessuna terna buona, e perciò possiede grado Ilazimi pari a 0. Nel caso di $K \neq 0$ invece è sufficiente usare la seguente costruzione:

$$N, 1, 2, 3, \ldots, K, N-1, N-2, N-3, \ldots, K+2, K+1$$

In questo caso il grado Ilazimi vale K, perché solo $1, 2, \ldots, K$ sono elementi centrali di qualche terna.

Tale costruzione funziona solo se $0 \leq K \leq N-2$. D'altra parte, è facile vedere che se tale condizione non dovesse valere allora nessuna configurazione dei grattacieli rispetterebbe le richieste: il grado Ilazimi di uno skyline formato da N grattacieli non può mai eccedere il valore $N-2$, in quanto il primo e l'ultimo grattacielo non possono essere elementi centrali di nessuna terna.

Esempio di codice C++

```cpp
const int MAXN = 1000000;

int max(int x, int y) {
    return x > y ? x : y;
}

int SX[MAXN];   // SX[i] è pari all'altezza del più alto grattacielo
                // con indice compreso tra 0 e i (estremi inclusi)
int DX[MAXN];   // DX[i] è pari all'altezza del più alto grattacielo
                // con indice compreso tra i e N-1 (estremi inclusi)

int valuta(int N, int *H) {
    // Inizializza SX e DX
    SX[0] = H[0];
    DX[N-1] = H[N-1];

    // E costruiscili induttivamente
    for (int i = 1; i < N; i++)
        SX[i] = max(SX[i-1], H[i]);
    for (int i = N-2; i >= 0; i--)
        DX[i] = max(DX[i+1], H[i]);

    // Il grado Ilazimi dello skyline è pari al numero di grattacieli
    // che ammettono sia alla propria sinistra che alla propria destra
    // un grattacielo più alto
    int risposta = 0;
    for (int i = 1; i < N-1; i++)
        if (H[i] < SX[i-1] && H[i] < DX[i+1])
            risposta++;

    return risposta;
}

void progetta(int N, int K, int *H) {
    if (K == 0) {
        // Lo skyline con grattacieli di altezza crescente possiede
        // grado Ilazimi pari a 0
        for (int i = 0; i < N; i++)
            H[i] = i+1;
    } else {
        // Il primo grattacielo è il più alto di tutti
        H[0] = N;
        // Seguito da una successione di altezze crescenti
        for (int i = 1; i < K+1; i++)
            H[i] = i;
        // E infine da una successione di altezze decrescenti
        for (int i = K+1; i < N; i++)
            H[i] = N - (i - K);
    }
}
```

Scontri equilibrati (scontri)

Limite di tempo: 1.0 secondi
Limite di memoria: 128 MiB

Ci sono delle frecce disposte in linea. Ogni freccia può puntare o a destra o a sinistra.

Si definisce uno *scontro* quando una freccia in posizione i punta verso destra mentre una freccia in posizione $i+1$ punta verso sinistra.

Se uno scontro avviene tra le frecce i e $i+1$, la sequenza contigua più lunga che comprende la freccia $i+1$ ed è composta solo da frecce con orientamento verso sinistra viene definita *parte destra* dello scontro. Analogamente la sequenza contigua più lunga che comprende la freccia i ed è composta solo da frecce con orientamento verso destra viene definita *parte sinistra* dello scontro.

Le frecce che compongono la parte sinistra e la parte destra di uno scontro si dicono che partecipano allo scontro. Uno scontro è *equilibrato* quando la parte sinistra e la parte destra hanno la stessa lunghezza.

Calcolare il numero minimo di frecce a cui bisogna cambiare orientamento in modo che:

- ogni freccia partecipi ad uno scontro
- ogni scontro sia equilibrato.

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`scontri.c`, `scontri.cpp`, `scontri.pas`) con un esempio di implementazione.

Se usi C/C++, dovrai implementare la funzione `Gira` utilizzando il seguente prototipo:

```
int Gira(int N, int *freccia);
```

Se invece usi Pascal, dovrai usare il seguente prototipo:

```
function Gira(N: longint; var freccia: array of longint): longint;
```

- N è il numero di frecce disposte in linea. Le frecce sono numerate da 0 a $N-1$.
- `freccia`$[i]$ indica l'orientamento della freccia numero i, e può assumere solo due valori: 0 per una freccia che punta a destra, 1 per una freccia che punta a sinistra.

Il valore di ritorno della funzione è un intero: la risposta al problema.

Assunzioni

Ogni test a cui verrà sottoposta la tua soluzione verificherà le seguenti condizioni:

- $1 \leq N \leq 10\,000$.
- N è pari.
- `freccia`$[i] = 0$ oppure `freccia`$[i] = 1$.

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1 [0 punti]:** Caso d'esempio.
- **Subtask 2 [11 punti]:** $N \leq 20$
- **Subtask 3 [27 punti]:** $N \leq 40$
- **Subtask 4 [39 punti]:** $N \leq 1\,000$
- **Subtask 5 [23 punti]:** nessuna limitazione specifica.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, a quel punto chiama la funzione `Gira` che dovete implementare, e scrive il risultato restituito dalla vostra funzione sul file `output.txt`.

Nel caso vogliate generare un input, il file `input.txt` deve avere questo formato:

- Prima riga: l'intero N.
- Seconda riga: N interi separati da spazi, ognuno uguale a 0 oppure 1, che indicano in ordine l'orientamento delle frecce.

Esempio di input/output

input.txt	output.txt
6 1 0 1 1 0 0	2

Spiegazione

Basta girare la prima e l'ultima freccia.

Soluzione

Questo problema può essere risolto usando la tecnica della *programmazione dinamica*, a patto di riconoscerne la sottostruttura ottima. Per alleggerire la soluzione definiamo *sequenza equilibrata* una qualsiasi sequenza di frecce che rispetti la condizione del testo, ovvero in cui ogni freccia partecipi ad uno scontro ed ogni scontro sia equilibrato. Il problema si traduce nel determinare, data una sequenza di N frecce, qual è il minimo numero di frecce a cui è necessario cambiare orientamento affinché la sequenza diventi equilibrata.

Dato che in una sequenza equilibrata ogni freccia deve partecipare ad uno scontro, sicuramente la prima freccia sarà rivolta verso destra, e farà parte della parte sinistra di uno scontro. Avendo questa informazione, è immediato notare che la definizione di sequenza equilibrata si presta ad una formulazione ricorsiva:

- la sequenza vuota è equilibrata;
- ogni sequenza equilibrata non vuota è composta da uno scontro equilibrato seguito da una sequenza equilibrata.

Notiamo inoltre che, date due sequenze di N frecce, il numero minimo di frecce da invertire per trasformare la prima nella seconda è pari al numero di posizioni i per cui la i-esima freccia della prima sequenza e la i-esima freccia della seconda sequenza sono discordi.

Analizziamo ora la sequenza iniziale. Per la definizione ricorsiva di sequenza equilibrata, sappiamo che al termine del processo di orientamento delle frecce esisterà un $k \leq N/2$ per cui le prime $2k$ frecce formeranno uno scontro, mentre le restanti frecce formeranno una sequenza equilibrata. Possiamo immaginare di fissare, uno alla volta, i diversi valori di k, orientare le prime $2k$ frecce e ricorrere sulle ultime $N - 2k$ posizioni: in fondo, anche le ultime frecce devono essere riorientate col minimo numero di operazioni.

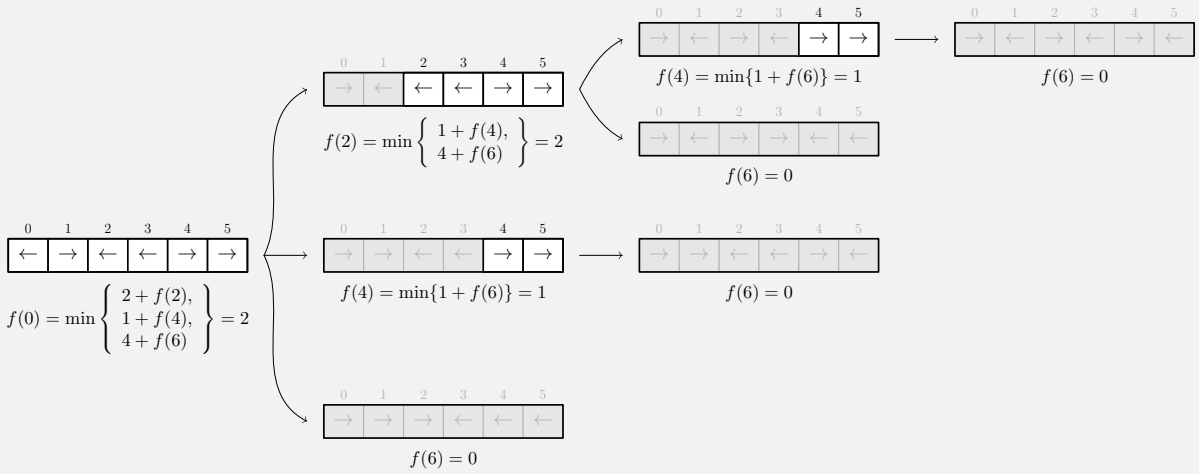


Figura 1: Esempio di scomposizione del problema.

Indichiamo con AggiustaScontro(i, j) il minimo numero di frecce da invertire affinché le frecce da i a j (estremi inclusi) della sequenza iniziale formino uno scontro equilibrato, e con $f(i)$ il numero minimo di frecce che è necessario invertire per rendere le ultime $N - i$ frecce della sequenza iniziale una sequenza equilibrata. Abbiamo:

$$f(i) = \begin{cases} 0, & \text{se } i = N \\ \min\limits_{k \leq (N-i)/2} \{\text{AggiustaScontro}(i, i + 2k - 1) + f(i + 2k)\}, & \text{altrimenti} \end{cases}$$

La risposta al problema è il valore di $f(0)$, che dipende a sua volta dai valori di $f(2), f(4), f(6), \ldots, f(N)$. Ipotizzando di implementare la funzione AggiustaScontro in modo che restituisca ogni risposta in tempo $O(N)$, abbiamo trovato un algoritmo di complessità cubica in N.

Per rendere più efficiente l'algoritmo possiamo notare che è possibile implementare la funzione AggiustaScontro in modo che risponda ad ogni domanda in tempo costante, rendendo quadratica la programmazione dinamica. Infatti, AggiustaScontro(i, j) è pari alla somma del numero di frecce che puntano a sinistra nell'intervallo $[i, (i + j + 1)/2)$ e del numero di frecce che puntano a destra nell'intervallo $[(i + j + 1)/2, j]$. Per ottenere velocemente questi due numeri possiamo precalcolare, per ogni posizione i della sequenza iniziale, quante sono le frecce che puntano a sinistra nell'intervallo $[0, i]$ di frecce.

Esempio di codice C++

```cpp
#include <algorithm>
using namespace std;

const int MAXN = 10005;

int sinistra[MAXN]; // sinistra[i] contiene il numero di frecce che
                    // puntano a sinistra nell'intervallo [0, i]
int f[MAXN];        // f[i] memorizza la risposta per la f(i) definita sopra

// Restituisce il minimo numero di frecce da girare affinché
// le frecce da i a j compresa formino uno scontro equilibrato
int AggiustaScontro(int i, int j) {
    int m = (i + j + 1) / 2;

    // Minimo numero di frecce da girare nella prima e nella
    // seconda metà dell'intervallo dato
    int prima_meta, seconda_meta;

    if (i > 0)
        prima_meta = sinistra[m - 1] - sinistra[i - 1];
    else
        prima_meta = sinistra[m - 1];
    seconda_meta = (j - m + 1) - (sinistra[j] - sinistra[m - 1]);

    return prima_meta + seconda_meta;
}

int Gira(int N, int *freccia){
    // Riempi il vettore sinistra
    for (int i = 0; i < N; i++) {
        if (i > 0)
            sinistra[i] = sinistra[i - 1];
        if (freccia[i] == 1)
            sinistra[i]++;
    }

    // Riempi il vettore della programmazione dinamica
    for (int i = N - 2; i >= 0; i -= 2) {
        f[i] = N;
        for (int k = 1; k <= (N - i) / 2; k++)
            f[i] = min(f[i], AggiustaScontro(i, i + 2*k - 1) + f[i + 2*k]);
    }

    return f[0];
}
```

Trasporti pericolosi (`trasporti`)

Limite di tempo:	1.0 secondi
Limite di memoria:	128 MiB

Una ditta organizza trasporti di valori tra città considerate alquanto pericolose.
Ogni trasporto avviene da una città di partenza fino ad una di arrivo percorrendo diverse strade. Ogni strada può essere percorsa in entrambi i sensi, inoltre prese due città della nazione esiste sempre un unico percorso di strade che le congiunge.

Quando un convoglio transita per una certa città i briganti del posto decidono di attaccarlo se e solo se si trovano in stretta maggioranza rispetto al numero di guardie armate che lo proteggono. Per evitare quindi che un trasporto venga derubato, la ditta dovrà impiegare tante guardie quanto è il massimo numero di briganti tra le città attraversate (incluse quelle di partenza e di arrivo).

Date le strade tra le città, il numero di briganti in ogni città e la lista dei trasporti da portare a termine, aiuta la ditta a trovare il minimo numero necessario di guardie per ogni trasporto al fine di evitare furti indesiderati.

Implementazione

Dovrai sottoporre esattamente un file con estensione `.c`, `.cpp` o `.pas`.

> ☞ Tra gli allegati a questo task troverai un template (`trasporti.c`, `trasporti.cpp`, `trasporti.pas`) con un esempio di implementazione.

Se usi C/C++, dovrai implementare la funzione `solve` utilizzando il seguente prototipo:

```
void solve(int N, int Q, int *briganti, int *A, int *B, int *start, int *end, int *sol);
```

Se invece usi Pascal, dovrai usare il seguente prototipo:

```
procedure solve(N, Q: longint; var briganti, A, B, start, stop, sol: array of longint);
```

- N è il numero di città. I numeri da 0 a $N-1$ identificano ognuno una città diversa.
- Q è il numero di viaggi che la ditta deve organizzare. Essi sono numerati da 0 a $Q-1$.
- `briganti`$[i]$ indica il numero di briganti nella città numero i ($0 \le i \le N-1$)
- `A`$[i]$ e `B`$[i]$ indicano le città agli estremi di una strada ($0 \le i \le N-2$)
- `start`$[i]$ e `end`$[i]$ indicano le città di partenza e di arrivo del trasporto numero i ($0 \le i \le Q-1$)

Durante l'esecuzione, la tua funzione deve riempire l'array `sol`. In particolare devi memorizzare in `sol`$[i]$ il numero di guardie che è necessario impiegare nel trasporto numero i ($0 \le i \le Q-1$).

Assunzioni

Ogni input ufficiale verificherà le seguenti condizioni:

- $1 \le N \le 100\,000$
- $1 \le Q \le 100\,000$
- $1 \le$ `briganti`$[i] \le 1\,000\,000$
- $0 \le$ `A`$[i]$, `B`$[i] \le N-1$
- $0 \le$ `start`$[i]$, `end`$[i] \le N-1$

Assegnazione del punteggio

Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1 [0 punti]:** Caso d'esempio
- **Subtask 2 [7 punti]:** Ogni città è collegata con la città 0
- **Subtask 3 [22 punti]:** Il percorso formato dalle strade è lineare (`0 - 1 - 2 - ...- N-1`)
- **Subtask 4 [13 punti]:** Tutti i trasporti partono dalla città 0
- **Subtask 5 [18 punti]:** In tutte le città, *tranne una*, è presente un solo brigante.
- **Subtask 6 [12 punti]:** $N \leq 500$ e $Q \leq 1\,000$
- **Subtask 7 [18 punti]:** $N \leq 2\,500$ e $Q \leq 2\,500$
- **Subtask 8 [10 punti]:** Nessuna limitazione specifica.

Grader di prova

Nella directory relativa a questo problema è presente una versione semplificata del grader usato durante la correzione, che potete usare per testare le vostre soluzioni in locale. Il grader di esempio legge i dati di input dal file `input.txt`, a quel punto chiama la funzione `solve` che dovete implementare, e scrive il risultato restituito dalla vostra funzione sul file `output.txt`.

Nel caso vogliate generare un input, il file `input.txt` deve avere questo formato:

- Riga 1: i numeri N e Q.
- Riga 2: `briganti`$[0]$ `briganti`$[1]$ `briganti`$[2] \ldots$ `briganti`$[N-1]$ (separati da spazi).
- Riga $3+i$: `A`$[i]$ `B`$[i]$ (con $0 \leq i \leq N-2$).
- Riga $N+2+i$: `start`$[i]$ `end`$[i]$ (con $0 \leq i \leq Q-1$).

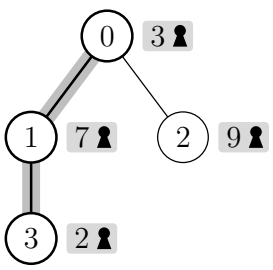
Il grader stamperà su `output.txt` il contenuto dell'array `sol`, riempito dalla vostra funzione.

Esempi di input/output

input.txt	output.txt
4 3 3 7 9 2 0 1 0 2 1 3 0 3 2 3 2 1	7 9 9

Spiegazione

La rete di città in input corrisponde a quella in figura: ogni città è rappresentata da un cerchio contenente il numero della città e ogni strada con un segmento nero; nel rettangolo a fianco di ogni città è specificato il numero di briganti.
Il percorso evidenziato corrisponde a quello che la ditta deve compiere per primo, cioè dalla città 0 alla città 3. Il numero di briganti in ciascuna delle città attraversate è 3, 7, 2, perciò a `sol`$[0]$ deve essere assegnato il valore 7.



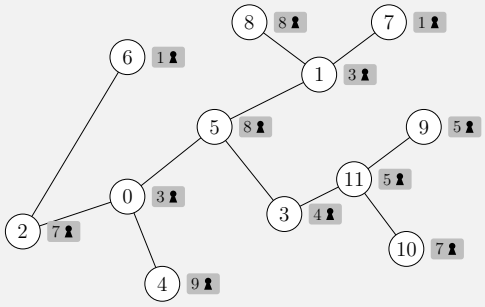
Soluzione

Per ogni subtask fino al quinto è possibile applicare idee diverse a seconda delle assunzioni che sono riportate nel testo. Non esporremo tali strategie, ma mostreremo direttamente un ragionamento generale che, inizialmente, ci permetterà di risolvere i subtask 6 e 7; infine, spiegheremo come adattare tale strategia per risolvere completamente il problema.

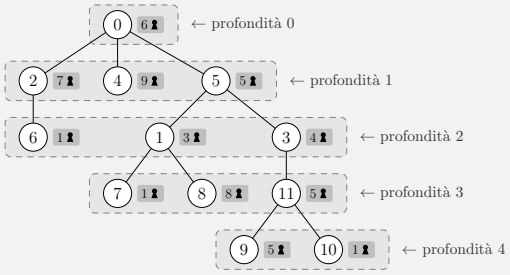
■ Introduzione e nozioni generali

Costruiamo la soluzione attorno all'esempio di mappa in figura 1a. In questo problema abbiamo a che fare con un grafo, i cui nodi sono le città e i cui archi sono le strade. La prima cosa da notare è che il grafo in questione è un albero, vale a dire un grafo connesso e senza cicli.

L'albero, così come ci viene dato in input, *non è radicato*, cioè non esiste nessun nodo particolare che rivesta il ruolo di radice; i rapporti padre–figlio tra i nodi non sono definiti. Nulla ci vieta, tuttavia, di radicare l'albero di nostra iniziativa, scegliendo come radice un qualsiasi nodo. Supporremo da qui in poi di aver scelto a tale fine il nodo 0, come in figura 1b. In questo modo possiamo visualizzare l'albero come una sorta di "grappolo", in cui ogni nodo (fatta eccezione per la radice) è sorretto dal proprio genitore e a sua volta sorregge i propri figli. Inoltre è ora lecito parlare di *profondità* di un nodo: alla radice è assegnata una profondità pari a 0, ai suoi figli una profondità pari a 1, ai figli dei figli una profondità pari a 2 e così via. Per definire tutti i rapporti padre–figlio e la profondità di ogni nodo, è sufficiente eseguire una *visita in profondità* partendo dal nodo radice.



(1a): Albero non radicato delle città.



(1b): Albero radicato nel nodo 0.

Aver radicato l'albero ci permette di considerare ogni percorso da A a B come diviso in due parti: la prima consiste nel risalire l'albero partendo da A, passando di genitore in genitore fino ad incontrare il primo nodo C antenato sia del nodo A che del nodo B; la seconda parte consiste nella risalita da B verso C. Indichiamo il nodo in cui effettuiamo il "cambio di rotta" con $C = \text{lca}(A, B)$ ("lca" è l'acronimo *lowest common ancestor*). Per fissare meglio il concetto, consideriamo, all'interno dell'albero in figura, il percorso tra i nodi $A = 10$ e $B = 8$: la figura 2 mostra il percorso e la sua scomposizione.

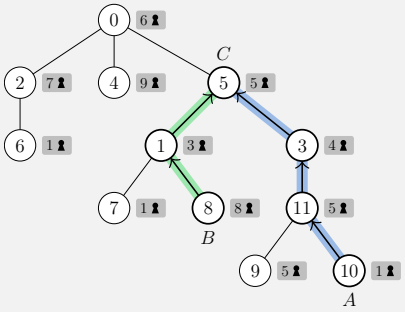


Figura 2: Percorso $A - B$ scomposto nei tratti $A \to C$ e $B \to C$.

Una volta spezzato il percorso nelle sue due parti, possiamo procedere a determinare il massimo numero di briganti nel percorso da A a C e da B a C, e considerare il massimo tra le due quantità.

■ Una soluzione $O(N \cdot Q)$

Immaginiamo due formiche, poste inizialmente una sul nodo A e una sul nodo B. Se fossimo in grado di simulare i movimenti delle formiche fino al momento in cui queste si incontrano nel nodo C, avremmo trovato un semplice algoritmo capace di rispondere ad ogni singola query in un tempo proporzionale al numero di livelli dell'albero (che, nel caso peggiore, è uguale al numero di nodi).

Cominciamo col considerare una situazione particolarmente favorevole: supponiamo che A e B si trovino alla stessa profondità. È sufficiente far salire le formiche un livello alla volta, in maniera sincronizzata: quando si troveranno per la prima volta nello stesso nodo, esso sarà necessariamente $C = \text{lca}(A, B)$.

Passando al caso generale, in cui non è garantito che A e B si trovino a uguale profondità, basta seguire la seguente procedura: inizialmente si fa risalire la formica che si trova a profondità maggiore, in modo da farle raggiungere lo stesso livello di profondità dell'altra formica. Una volta che le due formiche si trovano alla stessa profondità, possono manifestarsi due casi: queste si trovano nello stesso nodo, oppure no. Nel primo caso questo nodo è sicuramente il nodo C e non dobbiamo procedere oltre; altrimenti ci siamo ridotti al caso già analizzato e procediamo come sopra, muovendo le formiche in modo sincronizzato.

Ad ogni modo, mano a mano che le formiche risalgono l'albero teniamo traccia di quale è stato il massimo numero di furfanti tra quelli presenti nei nodi che esse hanno visitato. Quando le formiche si incontreranno, basterà valutare il massimo tra i due valori ricordati per ottenere la risposta alla query.

Questo algoritmo è sufficiente per risolvere, entro i limiti di tempo, il sesto e il settimo subtask.

■ Una soluzione $O((N + Q) \log N)$

Per risolvere anche l'ultimo subtask è necessario rendere più veloce la strategia appena descritta: invece di considerare due formiche, consideriamo due cavallette; queste sono più potenti delle formiche in quanto sono capaci di compiere dei salti e non solo di camminare. In particolare, queste cavallette possono compiere solo salti la cui lunghezza sia potenza di 2, e permettono di risalire $1, 2, 4, 8, 16, \ldots$ livelli di profondità in un colpo solo. La figura 3 mostra alcuni dei salti possibili.

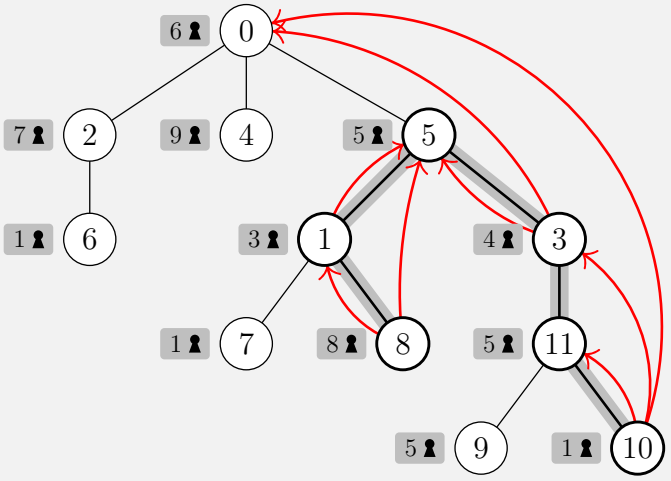


Figura 3: Alcuni dei salti possibili.

Una cavalletta può risalire x livelli di profondità in modo efficiente, secondo questo algoritmo:

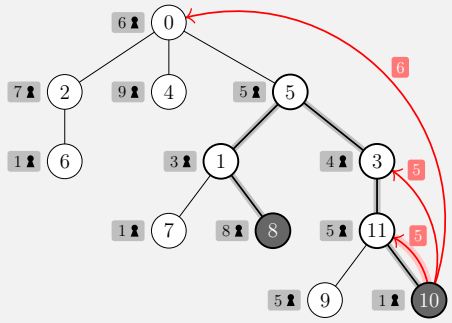
- trova la più grande potenza di 2 che non sia maggiore di x, e fa un salto di quella lunghezza;
- se la cavalletta non è ancora arrivata a destinazione, ripeti il passo precedente, considerando che il valore di x è cambiato.

Non è difficile dimostrare che in questo modo la cavalletta arriverà a destinazione con al più $\log_2 x$ salti.

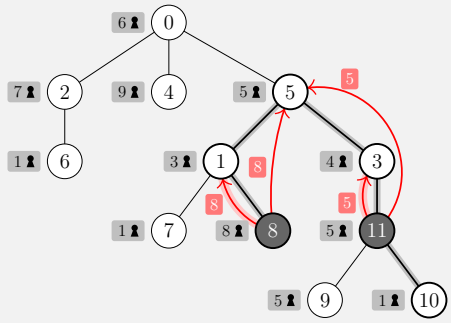
Come prima, per far incontrare le due cavallette, prima di tutto conviene portarle allo stesso livello. Per fare questo, basta far risalire la cavalletta che si trova più in basso di un numero di livelli uguale alla differenza di livello di partenza tra le due cavallette, usando l'algoritmo appena descritto. Poi, in modo simile a come facevamo per le formiche, si procede facendole muovere in sincrono:

- Se le cavallette si trovano sullo stesso nodo, termina.
- Altrimenti, trova il più lungo salto che, se eseguito contemporaneamente dalle due cavallette, *non* le fa incontrare nello stesso nodo.
- Se tale salto non esiste, fai avanzare entrambe le cavallette di 1, altrimenti esegui quel salto.
- Ritorna al primo punto.

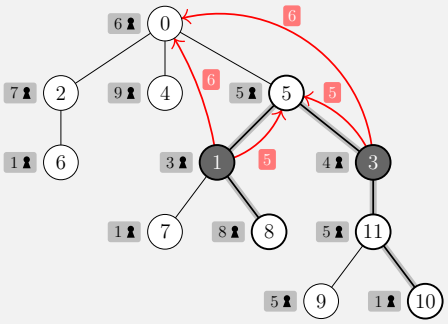
Ancora una volta, si dimostra facilmente che ogni cavalletta compirà al più $\log_2 N$ salti. Le figure 4a, 4b, 4c e 4d mostrano l'esecuzione dell'algoritmo sul percorso di figura 2.



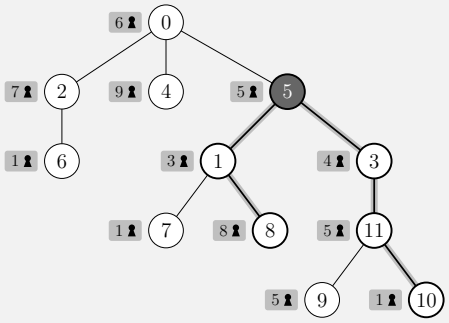
(4a): La cavalletta in 10 raggiunge quella in 8, risalendo l'albero.



(4b): Le cavallette si muovono in modo sincrono fino al proprio lca.



(4c): Non esistono salti che non facciano incontrare le cavallette: allora si muovono di 1.



(4d): Le cavallette si sono incontrate.

Se una cavalletta fosse in grado di ricordarsi il massimo peso tra quelli dei nodi coperti, *percorsi o saltati*, avremmo la risposta alla query come succedeva per le formiche, ma questa volta troveremmo la risposta ad ogni query in tempo $O(\log N)$. A dirla tutta, non è ancora chiaro come è possibile che una cavalletta sia in grado di fare salti di diversa lunghezza in modo efficiente.

Conviene precalcolarsi, a questo fine, delle informazioni per ogni nodo v e per ogni $0 \leq k \leq \log_2 N$:

- `antenato`$[k][v]$: il nodo a cui arriva una cavalletta se, partendo da v, esegue un salto lungo 2^k;
- `massimo`$[k][v]$: il massimo peso tra i nodi nel percorso da v a `antenato`$[k][v]$.

Tali informazioni si possono calcolare eseguendo un preprocessing sul grafo in tempo $O(N \log N)$. Per quanto riguarda `antenato`, notiamo che per ogni nodo v esclusa la radice, vale

$$\texttt{antenato}[0][v] = \texttt{genitore}[v]$$

Per salti più lunghi, cioè per $k \geq 1$, vale invece la seguente ricorsione:

$$\texttt{antenato}[k][v] = \texttt{antenato}[k-1][\texttt{antenato}[k-1][v]].$$

In modo del tutto analogo, notiamo che, per ogni nodo v esclusa la radice, vale

$$\texttt{massimo}[0][v] = \max\{\texttt{briganti}[v], \texttt{briganti}[\texttt{genitore}[v]]\}$$

Per salti più lunghi, cioè per $k \geq 1$, invece:

$$\texttt{massimo}[k][v] = \max\{\texttt{massimo}[k-1][v], \texttt{massimo}[k-1][\texttt{antenato}[k-1][v]]\}.$$

Esempio di codice C++

Proponiamo qui un'implementazione dell'algoritmo efficiente, valutata a punteggio pieno dal correttore.

```cpp
#include <vector>
#include <algorithm>
using namespace std;

const int MAXN = 100010;
const int LOGN = 20;

int *briganti;          // briganti[i] = numero di briganti nella città i

vector<int> adj[MAXN];  // adj[i] è un vettore che contiene i nodi che si possono
                        // raggiungere dal nodo i percorrendo una strada
int genitore[MAXN];
int profondita[MAXN];
int antenato[LOGN][MAXN];
int massimo[LOGN][MAXN];

// Esegue la visita in profondità
void dfs(int v, int p, int d) {
    // v = nodo che sto visitando in questo momento
    // p = nodo dal quale provengo
    // d = profondità all'interno dell'albero

    genitore[v] = p;
    profondita[v] = d;

    for (int i = 0; i < (int)adj[v].size(); i++) // Per ogni nodo vicino al nodo v
        if (adj[v][i] != p) // Se non è quello da cui provengo...
            dfs(adj[v][i], v, d+1); // ...Visitalo, impostando v come nodo di provenienza
                                    // e d+1 come profondità (perché sto scendendo di un livello)
}

// Ritorna il massimo numero di briganti che si incontra nel percorso dal nodo A al nodo B
int query(int A, int B) {
    // Da ora in poi consideriamo il contenuto delle variabili A e B come
    // gli indici dei nodi in cui si trovano le cavallette.
    // L'algoritmo terminerà quando A sarà uguale a B

    // impostiamo la risposta "provvisoria" al massimo tra i due nodi occupati dalle cavallette
    int risposta = max(briganti[A], briganti[B]);

    // Per comodità, facciamo in modo che sia sempre la cavalletta in A ad essere più in profondità
    if (profondita[A] < profondita[B]) swap(A,B);

    int k = LOGN - 1; // All'esecuzione di un salto, esso sarà di lunghezza 2^k
    // Abbiamo appena impostato k come per eseguire il salto più lungo possibile.

```

```
47      // Finché c'è differenza di profondità tra le due cavallette
48      while (profondita[A] - profondita[B] > 0) {
49          // Decrementa k, per evitare che la cavalletta in A "superi" la cavalletta in B
50          while (profondita[A] - profondita[B] < (1 << k)) k--;
51          // (Per chi non lo sapesse, l'operazione (1 << k) produce come risultato 2^k)
52
53          // Aggiorna la risposta provvisoria, considerando i nodi saltati dalla cavalletta A
54          risposta = max(risposta, massimo[k][A]);
55
56          A = antenato[k][A]; // Esegui il "salto", spostando la cavalletta
57      }
58
59      // Ora le cavallette si trovano alla stessa profondità
60
61      k = LOGN-1;
62
63      // Finché le due cavallette non si incontrano
64      while (A != B) {
65          // Decrementa k, per evitare che le cavallette vadano più in su del
66          // necessario, saltando il loro LCA
67          while(k > 0 && antenato[k][A] == antenato[k][B]) k--;
68
69          // Aggiorna la risposta temporanea e fai saltare
70          // la cavalletta A
71          risposta = max(risposta, massimo[k][A]);
72          A = antenato[k][A];
73          // e la cavalletta B
74          risposta = max(risposta, massimo[k][B]);
75          B = antenato[k][B];
76      }
77
78      return risposta;
79  }
80
81
82  void solve(int N, int Q, int *briganti, int *A, int *B, int *start, int *end, int *sol){
83      ::briganti = briganti;
84
85      // Per ogni strada, aggiorna i vettori dei vicini di ogni nodo
86      for (int i = 0; i < N - 1; i++) {
87          adj[A[i]].push_back(B[i]);
88          adj[B[i]].push_back(A[i]);
89      }
90
91      // Esegui la visita in profondità, assumendo 0 come nodo radice,
92      // impostando convenzionalmente -1 come nodo di provenienza
93      // e impostando la profondità a 0
94      dfs(0, -1, 0);
95
96      // Casi base per il nodo radice
97      antenato[0][0] = -1;    // Convenzionalmente, l'antenato è -1
98                              // se il salto fa "uscire fuori" dall'albero.
99      massimo[0][0] = briganti[0];
100
101     // I casi base per gli altri nodi
102     for (int i = 1; i < N; i++){
103         antenato[0][i] = genitore[i];
104         massimo[0][i] = max(briganti[i], briganti[genitore[i]]);
105     }
106
107     // In ordine crescente di "lunghezza del salto" applica le formule ricorsive
108     for (int k = 1; k < LOGN; k++)
109         for (int i = 0; i < N; i++) {
110             // Se il salto (k-1) già faceva uscire fuori dall'albero
111             if (antenato[k-1][i] < 0){
112                 antenato[k][i] = antenato[k-1][i];
113                 massimo[k][i] = massimo[k-1][i];
114             } else {
115                 antenato[k][i] = antenato[k-1][antenato[k-1][i]];
116                 massimo[k][i] = max(massimo[k-1][i], massimo[k-1][antenato[k-1][i]]);
117             }
118         }
119
120     // Calcola le risposte e riempi l'array sol[]
121     for (int i = 0; i < Q; i++) sol[i] = query(start[i], end[i]);
122 }
```