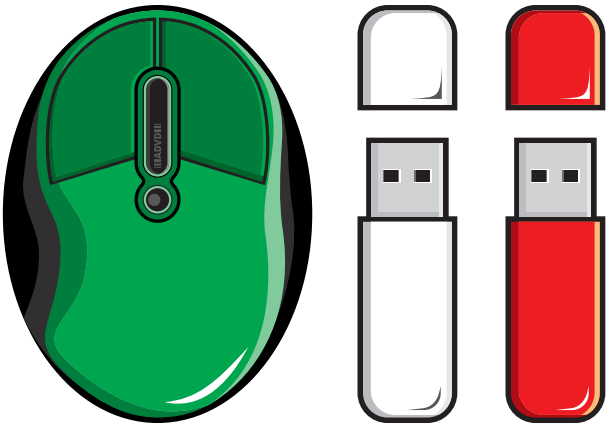
Ministero dell'Istruzione dell'Università e Ricerca

AICA
Associazione Italiana per l'Informatica ed il Calcolo Automatico



OLIMPIADI ITALIANE DI INFORMATICA

FISCIANO, 18 – 20 SETTEMBRE 2014

Finale nazionale

Testi e soluzioni ufficiali

Problemi a cura di

Luigi Laura

Coordinamento

Monica Gati

Testi dei problemi

Giorgio Audrito, Matteo Boscariol, Alice Cortinovis, Gabriele Farina, Giada Franz
Federico Glaudo, Roberto Grossi, Luigi Laura, Giovanni Paolini, Romeo Rizzi

Soluzioni dei problemi

Alice Cortinovis, William Di Luigi, Gabriele Farina

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Indice

Introduzione

Qui di seguito riportiamo la lettera scritta dalla Prof.ssa Genny Tortora, che ha presieduto l'organizzazione della finale nazionale delle Olimpiadi Italiane di Informatica, svoltesi per il secondo anno consecutivo nel campus di Fisciano, Università di Salerno. Da parte di tutto il comitato delle Olimpiadi, un enorme ringraziamento alla Prof.ssa Genny Tortora e a tutto il suo staff, per la splendida ospitalità che hanno riservato ai ragazzi e a tutti i partecipanti.

Sono terminate da poco le Olimpiadi Italiane di Informatica ospitate per il secondo anno all'interno del Campus di Fisciano ed organizzate dal Dipartimento di Studi e Ricerche Aziendali – Management & Information Technology dell'Università degli Studi di Salerno, in collaborazione con l'Istituto di Istruzione Superiore di Baronissi e con il supporto di alcune rilevanti aziende del settore informatico quali INDRA Company, BUSINESS SOLUTION, JOBIZ FORMAZIONE.

Le Olimpiadi costituiscono sicuramente un'occasione per accrescere l'interesse nei giovani verso l'Informatica e per valorizzare le loro attitudini al problem solving ed all'utilizzo delle nuove tecnologie; contemporaneamente sono un'opportunità per scoprire e valorizzare nuovi talenti. In considerazione della centralità che l'Informatica riveste nei processi d'innovazione, le Olimpiadi rappresentano anche un'opportunità per indirizzare i ragazzi verso una partecipazione più attiva al conseguimento degli ambiziosi obiettivi che l'Italia si è prefissata di raggiungere con la programmazione di Horizon 2020.

L'Ateneo salernitano è stato ben lieto di essere per due anni consecutivi al fianco del MIUR e di AICA nell'organizzazione di questo importante evento che lo scorso anno ha permesso all'Italia di essere protagonista delle Olimpiadi Internazionali con l'assegnazione di una medaglia d'argento. L'augurio è che in Kazakhstan, nel 2015, si possa conquistare il podio più alto.

Resta da parte nostra tutto l'impegno e la disponibilità a continuare ad affiancare questa iniziativa.

Genny Tortora
Professore Ordinario di Informatica

Olimpiadi Italiane di Informatica
Università degli Studi di Salerno

Accensione dei PC (accensione)

Limite di tempo: 1.0 secondi
Limite di memoria: 256 MiB

Alle OII, gli N computer dei partecipanti sono numerati da 1 a N. Al momento solo alcuni di essi sono accesi, e Gabriele deve accenderli tutti prima dell'inizio della gara!

Sfortunatamente, l'accensione e lo spegnimento dei computer può avvenire solamente attraverso un sofisticato sistema di pulsanti. Nella Sala Controllo vi sono infatti N pulsanti, anch'essi numerati da 1 a N. Il pulsante K ha, come effetto, quello di cambiare lo stato di tutti i computer il cui numero identificativo sia un divisore di K. Ad esempio, il pulsante 12 cambia lo stato dei computer 1, 2, 3, 4, 6, 12.

Ogni pulsante può essere premuto al massimo una volta, ed è necessario premere i pulsanti in ordine crescente (non si può premere il pulsante K_1 dopo aver premuto il pulsante K_2, se $K_1 < K_2$). Un computer può venire acceso e/o spento anche varie volte; l'importante è che alla fine tutti i computer risultino accesi. Sapendo quali sono i computer inizialmente accesi, dovete decidere quali pulsanti premere in modo da accenderli tutti.

Ad esempio, supponiamo che i computer siano $N = 6$. Ci sono quindi $N = 6$ pulsanti, che cambiano lo stato dei computer secondo la tabella seguente.

Pulsante	Effetto
1	Computer 1
2	Computer 1 e 2
3	Computer 1 e 3
4	Computer 1, 2 e 4
5	Computer 1 e 5
6	Computer 1, 2, 3 e 6

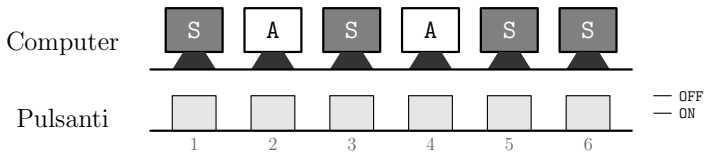
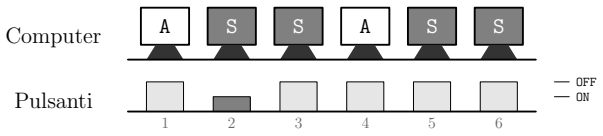


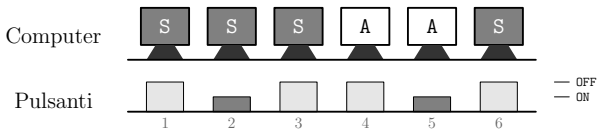
Figura 1: Configurazione iniziale dei computer.

Supponiamo che i computer siano inizialmente nello stato della Figura 1 (dove `S` significa spento e `A` significa acceso). In questo caso, per poterli accendere tutti, Gabriele deve premere in sequenza i seguenti pulsanti:

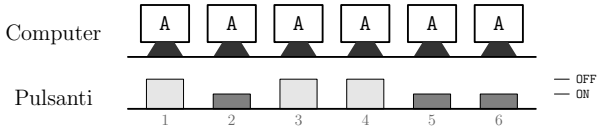
Pulsante 2



Pulsante 5



Pulsante 6



Aiuta Gabriele a determinare la sequenza di pulsanti da premere per poter accendere tutti i computer.

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** [**5 punti**]: Caso d'esempio.
- **Subtask 2** [**10 punti**]: $N \leq 10$.
- **Subtask 3** [**8 punti**]: I computer spenti all'inizio sono tutti e soli quelli contrassegnati da un numero primo (2, 3, 5, 7...).
- **Subtask 4** [**26 punti**]: $N \leq 1000$.
- **Subtask 5** [**28 punti**]: $N \leq 100\,000$.
- **Subtask 6** [**23 punti**]: Nessuna limitazione specifica (vedi la sezione **Assunzioni**).

Implementazione

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

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

Dovrai implementare la seguente funzione:

C/C++	`void Accendi(int N, int acceso[], int pulsante[]);`
Pascal	`procedure Accendi(N: longint; var acceso, pulsante: array of longint);`

dove:

- L'intero N rappresenta il numero di computer.
- L'array `acceso` descrive lo stato iniziale dei computer: se `acceso`$[i]$ vale 1, all'inizio l'i-esimo computer è acceso; se vale 0 è spento.
- La funzione dovrà riempire l'array `pulsante`, inserendo nella posizione i dell'array il valore 1 se è necessario premere l'i-esimo pulsante, 0 altrimenti. È garantito che inizialmente `pulsante`$[i]$ valga 0 per ogni i.

Attenzione: `acceso`[0] e `pulsante`[0] non contengono informazioni utili (in quanto i computer e i pulsanti sono numerati da 1 a N).

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 `Accendi` che dovete implementare. Il grader scrive sul file `output.txt` quali pulsanti sono stati premuti dalla vostra funzione.

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

- Riga 1: contiene l'intero N, il numero di computer.

- Riga 2: contiene N valori; l'i-esimo di questi vale 1 se l'i-esimo computer è acceso, vale 0 se è spento.

Il file `output.txt` invece ha questo formato:

- Riga 1: contiene N valori; l'i-esimo di questi vale 1 se l'i-esimo pulsante è stato premuto, vale 0 altrimenti.

Assunzioni

- $1 \leq N \leq 1\,000\,000$.

Esempi di input/output

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

Questo caso di input corrisponde all'esempio spiegato nel testo.

Soluzione

Innanzi tutto notiamo che conta solo *quali* bottoni vengono premuti, e non anche l'ordine in cui questi vengono effettivamente premuti. Forti di questo, l'osservazione centrale per risolvere il problema consiste nell'accorgersi che, conoscendo la configurazione iniziale, la scelta sull'ultimo pulsante è forzata: se il computer N è acceso allora il pulsante N non deve essere premuto (nessun altro pulsante sarebbe in grado poi di riaccendere il computer N); se, al contrario, il computer N è spento allora sicuramente è necessario premere il pulsante N (nessun altro pulsante lo accende), cambiando come effetto collaterale lo stato dei computer identificati da un numero divisore di N. Dopo aver agito sul bottone N, e averlo eventualmente premuto, il computer N sarà acceso, e possiamo dimenticarcene, passando a considerare i primi $N-1$ computer, i cui stati potrebbero essere stati cambiati rispetto alla configurazione iniziale, reiterando il ragionamento.

In altre parole, possiamo accendere i computer uno alla volta partendo dal fondo: per ogni bottone i, in ordine da N a 1, valutiamo se il computer i è spento, e in caso premiamo il bottone i. Questo semplice metodo è in grado, partendo da una configurazione qualunque, di accendere tutti i computer. Proponiamo una simulazione dell'algoritmo in Figura 1.

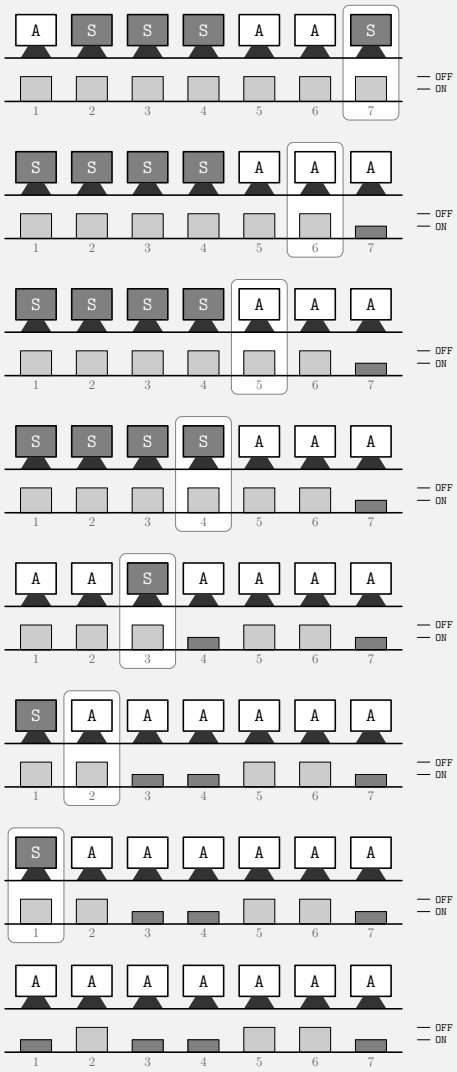


Figura 1: Simulazione dell'algoritmo.

■ Una soluzione $O(N^2)$

Il punto più delicato nell'algoritmo proposto è il momento in cui, dopo aver premuto il pulsante i, si rende necessario aggiornare lo stato dei computer interessati dal pulsante premuto. Possiamo immaginare di iterare su tutti i computer con indice compreso tra 1 e i, controllando di volta in volta se questi sono modificati dal pulsante i, e aggiornando lo stato dei computer con un numero divisore di i.

Questo approccio conduce ad un algoritmo di complessità quadratica, in grado di risolvere il Subtask 4.

■ Una soluzione $O(N\sqrt{N})$

Per rendere più efficiente l'algoritmo quadratico, possiamo accorgerci che i divisori di un numero n sono disposti "a coppie": ogni volta che d è un divisore di n, infatti, anche n/d è un divisore di n. Inoltre, almeno uno tra d e n/d deve essere minore o uguale a $\sqrt{n}$[1]. Per elencare tutti i divisori di n possiamo allora iterare su tutti i numeri da 1 a $\lfloor\sqrt{n}\rfloor$, non più su tutti i numeri da 1 a n. L'unico caso particolare è quello per cui n è un quadrato perfetto: in tal caso dobbiamo assicurarci di aggiornare una sola volta lo stato del computer $\sqrt{n}$.

Questo algoritmo, di complessità $O(N\sqrt{N})$, permetteva di risolvere il Subtask 5.

[1] Infatti, se sia d che n/d fossero maggiori di $\sqrt{n}$, il loro prodotto sarebbe $n = d \cdot (n/d) > \sqrt{n} \cdot \sqrt{n} = n$, assurdo.

■ Una soluzione $O(N \log N)$ poco efficiente nella pratica

Le due soluzioni che abbiamo analizzato finora sono simili: prima premiamo un pulsante k e successivamente iteriamo sui computer, per cambiare di stato i computer con numero divisore di k. In questa iterazione, tuttavia, consideriamo molti candidati che, non avendo come numero un divisore di k, non verranno cambiati di stato. Per evitare questo dispendio di tempo, questa volta operiamo in maniera leggermente diversa: prima ancora di cominciare a valutare i pulsanti, precalcoliamo per ogni pulsante la lista di computer che questi andranno a modificare. Attribuiamo ad ogni pulsante i la lista `divisori`$[i]$ di tutti i computer che vengono modificati da questo.

Le liste, inizialmente vuote, sono progressivamente riempite in questo modo:

- aggiungiamo alle liste di tutti i pulsanti il valore 1;
- aggiungiamo alle liste di tutti i pulsanti pari il valore 2;
- aggiungiamo alle liste di tutti i pulsanti multipli di 3 il valore 3;
- ...
- aggiungiamo alla lista del pulsante N il valore N.

In questo modo, quando verrà premuto il pulsante k, avremo già a disposizione la lista di tutti i computer da modificare, senza doverla calcolare.

Per determinare la complessità dell'algoritmo, notiamo che il numero di operazioni compiute è proporzionale alla somma delle dimensioni delle varie liste, e quindi bene approssimata dalla formula

$$N + \frac{N}{2} + \frac{N}{3} + \cdots + \frac{N}{N}.$$

Dimostreremo che tale somma è sicuramente inferiore a $N + N \log_2 N$.

APPROFONDIMENTO

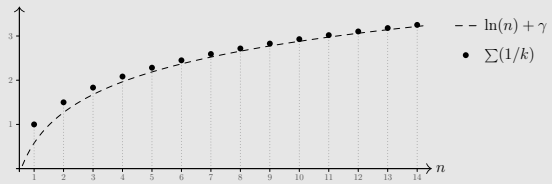
La complessità dell'algoritmo è strettamente legata all'andamento, in funzione di N, della somma del numero di divisori tra tutti i numeri da 1 a N. Come abbiamo già visto, questa grandezza è in effetti bene approssimata dalla somma

$$\frac{N}{1} + \frac{N}{2} + \cdots + \frac{N}{N} = N\left(1 + \frac{1}{2} + \cdots + \frac{1}{N}\right).$$

Abbiamo già dimostrato che la somma in parentesi è sempre inferiore a $1 + \log_2 N$. Esiste una stima più precisa: è stato dimostrato che

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln(n)\right),$$

dove la costante $\gamma \approx 0.577$ è nota come *costante di Eulero–Mascheroni*. Intuitivamente, questo significa che, a mano a mano che n diventa sempre più grande, la somma $1+1/2+\cdots+1/n$ approssima sempre meglio la funzione $\ln(n)+\gamma$.



Per farlo, raggruppiamo gli addendi in (circa) $1 + \log_2 N$ gruppi, in modo che ogni gruppo cominci con una frazione che ha al denominatore una potenza di 2:

$$N + \left(\frac{N}{2} + \frac{N}{3}\right) + \left(\frac{N}{4} + \frac{N}{5} + \frac{N}{6} + \frac{N}{7}\right) + \cdots + \left(\cdots + \frac{N}{N}\right)$$

In ogni gruppo la somma degli addendi è minore di N, perciò la somma totale non può che essere inferiore a $N(\log_2 N + 1) = N + N \log_2 N = O(N \log N)$.

Nonostante la complessità computazionale sia inferiore, nella pratica questa soluzione si rivela spesso paragonabile a quella precedente, di complessità $O(N\sqrt{N})$, perché intervengono altri fattori, come il tempo di allocazione della memoria, che incidono sulle prestazioni dell'algoritmo.

Allo stato delle cose[2], per fare in modo che questa soluzione non ecceda il tempo limite di 1 secondo su un processore di fascia media è necessario implementare con particolare attenzione l'algoritmo, avendo cura di minimizzare qualunque tipo di overhead, soprattutto quello dovuto all'allocazione dinamica della memoria.

■ Una soluzione $O(N \log N)$ efficiente

Pur restando all'interno della stessa classe di complessità, l'algoritmo può essere ulteriormente migliorato. In effetti, possiamo evitare di ricalcolare lo stato dei computer dopo aver premuto un pulsante: è facile ricostruire lo stato del computer k sapendo se i pulsanti $k+1, k+2, \ldots, N$ sono stati premuti. Infatti, gli unici pulsanti che possono aver cambiato lo stato del computer k sono $2k, 3k, 4k, \ldots$. Dunque è sufficiente controllare quanti di questi sono stati premuti per determinare lo stato del computer k e decidere se è necessario premere il pulsante k di conseguenza.

Come prima, la complessità dell'algoritmo è $O(N \log N)$; non dovendo più allocare una grande quantità di memoria per memorizzare la lista dei divisori di ogni numero, tuttavia, l'implementazione in codice di questo algoritmo risulta considerevolmente più veloce della precedente.

Esempio di codice C++11

Proponiamo, nelle prossime pagine, un'implementazione funzionante per ognuna delle soluzioni proposte, fatta eccezione per la soluzione più lenta, di complessità $O(N^2)$, che viene lasciata come facile esercizio per i lettori più meticolosi.

■ Soluzione $O(N\sqrt{N})$

```cpp
void Accendi(int N, int acceso[], int pulsante[]) {
    for (int i = N; i >= 1; i--) {
        // Se il computer i è spento allora premiqmo il pulsante i,
        // e cambiamo lo stato dei computer di conseguenza
        if (!acceso[i]) {
            pulsante[i] = 1;

            // Cerchiamo tutti i j che dividono i, ma solo
            // fino a j <= sqrt(i)
            for (int j = 1; j * j <= i; j++) {

                // Se j è un divisore di i, allora anche i/j è un divisore di i
                if (i % j == 0) {
                    // Cambia lo stato del computer j
                    acceso[j] ^= 1;

                    // Cambia lo stato del computer i/j, ma solo se questo non
                    // coincide con j. Questo è per evitare ad esempio il caso
                    // i = 4, j = 2, in cui altrimenti lo stato del computer 2
                    // verrebbe cambiato due volte invece di una sola
                    if (j * j != i)
                        acceso[i/j] ^= 1;
                }
            }
        }
    }
}
```

[2] Questa frase è stata scritta nel settembre del 2014.

■ Soluzione $O(N \log N)$ poco efficiente

Proponiamo una versione già ottimizzata della soluzione presentata. Si è optato per una soluzione che privilegiasse la facilità di lettura rispetto all'eleganza del codice.

```cpp
#include <cstddef>

const int MAXN = 1000000;
const int MAXL = 15000000;

// Per minimizzare l'overhead dovuto all'allocazione dinamica della memoria, allochiamo staticamente MAXL
// nodi di lista, e gestiamoli manualmente. Il numero MAXL è sufficientemente grande da garantire che
// non sarà necessario allocare ulteriori nodi
struct node_t {
    int value;
    node_t* next = NULL;
};
// pool rappresenta la memoria allocata staticamente, ovvero i MAXL nodi di lista
node_t pool[MAXL];
// pool_ptr punta al primo nodo non ancora usato
node_t *pool_ptr;
// end[i] punta all'ultimo nodo della lista dei divisori del numero i
node_t *end[MAXN + 1];

void Accendi(int N, int acceso[], int pulsante[]) {
    // Inseriamo il divisore 1 nelle liste dei divisori dei numeri da 1 a N
    for (int i = 1; i <= N; i++) {
        pool[i].value = 1;
        end[i] = pool + i;
    }
    pool_ptr = pool + N + 1;

    // Inseriamo ora nelle varie liste tutti i divisori dal 2 in poi
    for (int i = 2; i <= N; i++) {
        for (int j = i; j <= N; j += i) {
            pool_ptr->value = i;
            end[j]->next = pool_ptr;
            end[j] = pool_ptr++;
        }
    }

    // Dopo aver costruito le liste, procediamo come nella soluzione precedente, col vantaggio di conoscere
    // in anticipo quali computer vengono modificati dal pulsante i
    for (int i = N; i >= 1; i--) {
        if (!acceso[i]) {
            pulsante[i] = true;

            for (node_t* j = pool + i; j != end[i]; j = j->next)
                acceso[j->value] ^= 1;
        }
    }
}
```

■ Soluzione $O(N \log N)$ efficiente

```cpp
void Accendi(int N, int acceso[], int pulsante[]) {
    for (int i = N; i >= 1; i--) {
        int stato_del_pc_i = acceso[i];

        // Controllo tutti i pulsanti con numero multiplo di i
        for (int j = 2 * i; j <= N; j += i) {
            // Se il pulsante j è stato premuto, lo stato del computer
            // i è stato invertito
            if (pulsante[j] == 1)
                stato_del_pc_i ^= 1;
        }

        if (stato_del_pc_i == 0)
            pulsante[i] = 1;
    }
}
```

Collo di bottiglia (`bottleneck`)

Limite di tempo: 2.0 secondi
Limite di memoria: 256 MiB

Luca e William stanno valutando una nuova e apparentemente vantaggiosa offerta della Fraudolent, la più diffusa compagnia telefonica dell'arcipelago delle isole NoWhere. Il contratto garantisce ai suoi utilizzatori che ogni comunicazione online tra due computer seguirà sempre il percorso più breve (in termini di nodi intermedi) all'interno della rete internet dell'arcipelago.

Dati i trascorsi della società con la giustizia, tuttavia, i due ragazzi sono sospettosi, e temono che quello della Fraudolent sia solo un torbido tentativo di ingannare gli utenti più incauti: non a caso, infatti, il contratto parla di connessione *più breve*, e mai di più veloce. Per fugare ogni dubbio, Luca e William decidono di valutare, nel caso in cui dovessero sottoscrivere la promozione, la velocità di trasmissione dei dati tra i propri computer.

Per misurare la minima velocità di trasmissione garantita dal contratto, i ragazzi hanno mappato l'intera rete internet dell'arcipelago. Questa è rappresentata da un grafo in cui i vertici costituiscono i vari nodi della rete (tra cui i computer di William e di Luca) e gli archi identificano i collegamenti tra questi. Gli archi riportano anche la propria *capacità*, cioè il massimo numero di megabit che ogni secondo possono fluire attraverso di essi. La *velocità di trasmissione* in un percorso è pari alla minima capacità degli archi che lo compongono.

William e Luca sanno che la Fraudolent, pur essendo vincolata da contratto a garantire una trasmissione che passi per il minimo numero possibile di nodi intermedi, sceglierà sempre, tra tutti, il percorso più lento per collegare due computer, per risparmiare sull'utilizzo della rete.

Aiuta Luca e William a determinare, data la mappa della rete dell'arcipelago, quale sarebbe la velocità di trasmissione dati tra i propri computer. Per esempio, consideriamo la rete in figura 2: il computer di William corrisponde al nodo 2, colorato di blu, mentre quello di Luca al nodo 8, colorato di rosso; i numeri sui collegamenti rappresentano la capacità degli archi, in megabit al secondo.

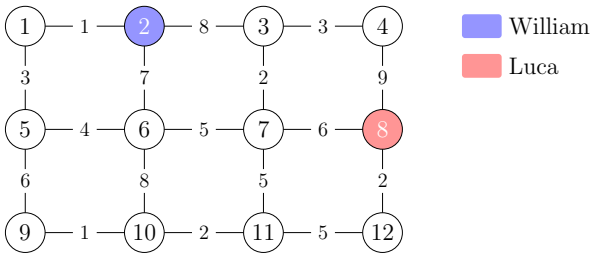
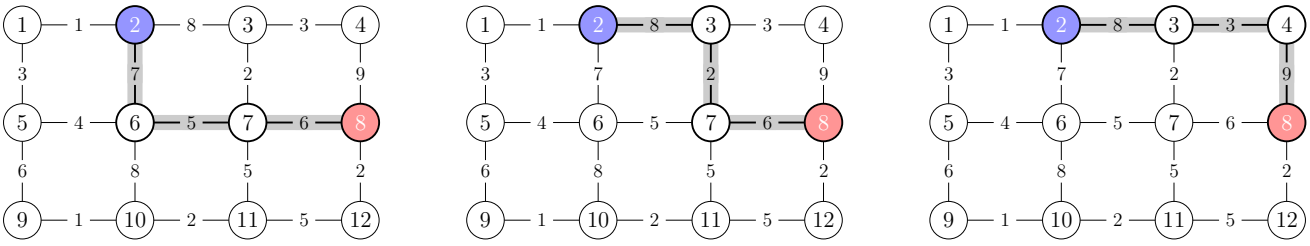


Figura 2: La rete dell'arcipelago.

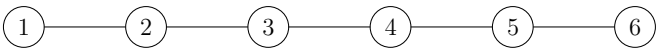
I tre percorsi di lunghezza minima tra i due computer sono mostrati qui sotto. La velocità della connessione in questo caso sarebbe pari a 2 megabit al secondo, corrispondente alla velocità di trasmissione del secondo percorso.



Assegnazione del punteggio

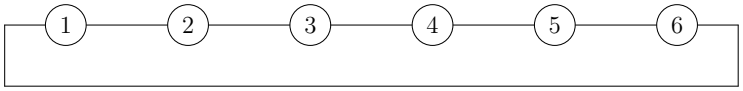
Il tuo programma verrà testato su diversi test case raggruppati in subtask. Per ottenere il punteggio relativo ad un subtask, è necessario risolvere correttamente tutti i test relativi ad esso.

- **Subtask 1 [5 punti]:** Caso d'esempio
- **Subtask 2 [7 punti]:** I nodi di rete sono collegati in sequenza, come in figura



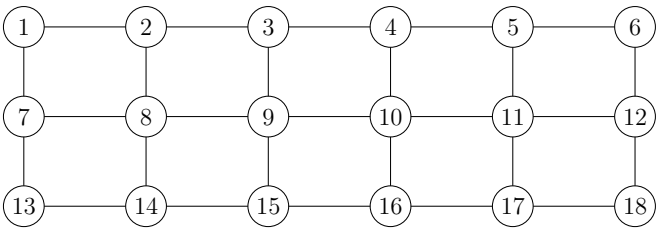
  È sempre garantito che il primo nodo della sequenza è il nodo 1, e che il nodo k segue sempre il nodo $k-1$, per ogni $k \geq 2$.

- **Subtask 3 [10 punti]:** I nodi di rete sono collegati ad anello, come in figura



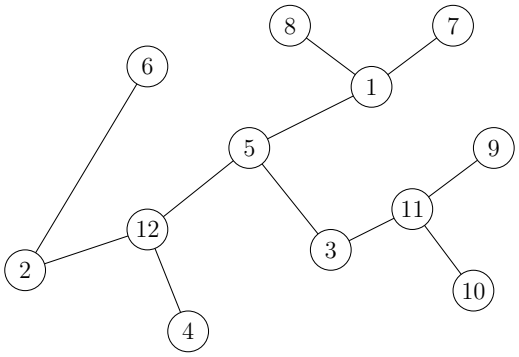
  È sempre garantito che il nodo N è collegato al nodo 1, e che il nodo k segue il nodo $k-1$, per ogni $2 \leq k \leq N$.

- **Subtask 4 [11 punti]:** I nodi di rete sono collegati a griglia, come in figura



  È sempre garantito che i nodi seguiranno una numerazione per righe.

- **Subtask 5 [17 punti]:** Tra ogni coppia di nodi di rete esiste un unico percorso che li collega, come in figura



- **Subtask 6 [22 punti]:** $N, M \leq 1000$.
- **Subtask 7 [28 punti]:** Nessuna limitazione specifica (vedi la sezione **Assunzioni**).

Implementazione

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

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

Dovrai implementare la seguente funzione:

C/C++	`int Analizza(int N, int M, int W, int L,` `  int arco_da[], int arco_a[], int capacita[], int R, int C);`

Pascal	`function Analizza(N, M, W, L: longint;`
	`  var arco_da, arco_a, capacita: array of longint;`
	`  R, C: longint): longint;`

dove:

- N rappresenta il numero di nodi di rete.
- M rappresenta il numero di collegamenti.
- W e L sono rispettivamente il computer di William e quello di Luca.
- `arco_da` e `arco_a` sono due array di dimensione M che rappresentano i collegamenti. L'i-esimo collegamento di rete connette (in modo bidirezionale) i nodi `arco_da`$[i]$ e `arco_a`$[i]$. È garantito che lo stesso collegamento non venga mai ripetuto.
- `capacita` è un array di dimensione M. L'intero `capacita`$[i]$ rappresenta la capacità dell'i-esimo collegamento, in megabit al secondo.
- R e C sono parametri speciali che di norma valgono -1. L'unica eccezione è il caso della topologia a griglia (vedi **Subtask 4**), in cui R e C rappresentano rispettivamente il numero di righe e di colonne della griglia.

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 `Analizza` che dovete implementare. Il grader scrive sul file `output.txt` la risposta fornita dalla funzione `Analizza`.

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

- Riga 1: contiene l'intero `N`, che rappresenta il numero di nodi di rete, l'intero `M`, che rappresenta il numero di collegamenti, gli interi W e L, che sono i computer di William e di Luca rispettivamente, gli interi R e C, che di norma valgono -1 e nel caso della topologia a griglia rappresentano rispettivamente il numero di righe e di colonne della griglia.
- Righe $2, \ldots, M+1$: l'i-esima riga contiene tre interi x_i, y_i, z_i con $1 \leq x_i \leq N$ e $1 \leq y_i \leq N$, che rappresentano un collegamento tra i nodi x_i e y_i con capacità z_i.

Il file `output.txt` invece ha questo formato:

- Riga 1: contiene il valore restituito dalla funzione `Analizza`.

Assunzioni

- $2 \leq N \leq 100\,000$.
- $1 \leq M \leq 1\,000\,000$.
- Le capacità dei collegamenti sono interi compresi tra 0 e $1\,000\,000\,000$.
- Quando R e C non valgono entrambi -1, vale che $1 \leq R, C \leq 300$.
- Il grafo della rete è connesso, cioè ogni nodo è raggiungibile da tutti gli altri.
- Nessun collegamento connette un nodo con se stesso.
- Per ogni coppia di nodi c'è al più un collegamento che li connette.

Esempi di input/output

input.txt	output.txt
12 17 2 8 3 4 1 2 1 2 3 8 3 4 3 1 5 3 2 6 7 3 7 2 4 8 9 5 6 4 5 9 6 6 7 5 7 8 6 9 10 1 10 11 2 11 12 5 10 6 8 11 7 5 8 12 2	2

Questo caso di input corrisponde all'esempio spiegato nel testo.

Soluzione

Questo problema ammette diverse soluzioni "specializzate", in grado di risolvere solo alcuni dei subtask proposti. Non ci occuperemo di tali soluzioni parziali in questa sede; mostreremo invece due approcci, diversi ma equivalenti dal punto di vista della correttezza, per risolvere completamente il problema.

■ Primo approccio

La prima delle due soluzioni che proponiamo consiste in una versione modificata dell'algoritmo di *visita in ampiezza* (BFS) di un grafo. Intuitivamente, partiamo dal PC di William, e visitiamo tutti i nodi a distanza 1, 2, ... fino a raggiungere il computer di Luca. Mano a mano che ci espandiamo nel grafo, manteniamo per ogni nodo v la minima capacità del percorso tra il computer di William e v stesso.

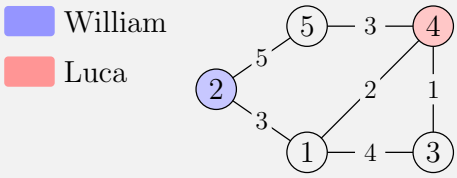
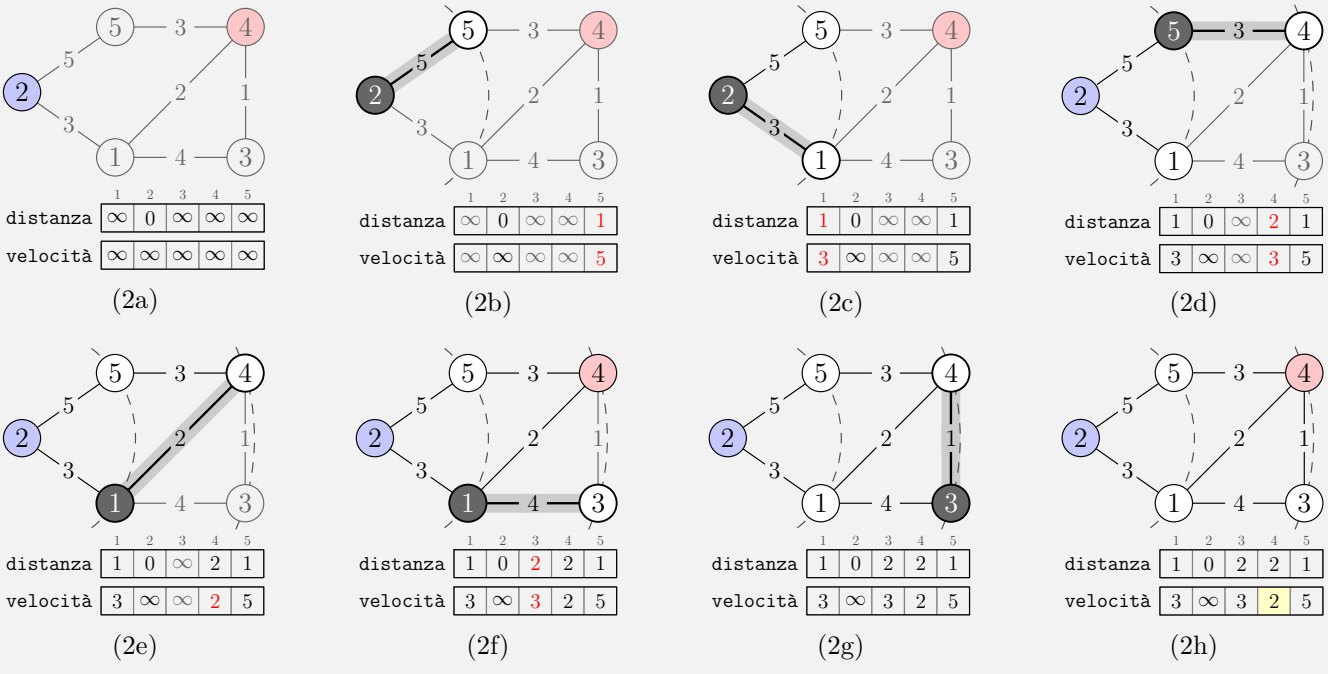


Figura 1: Mappa d'esempio.

Illustriamo i dettagli dell'algoritmo attraverso un esempio: supponiamo che la mappa della rete coincida con quella di Figura 1, dove il nodo blu ancora una volta corrisponde al PC di William (d'ora in poi indicato con W) e il nodo rosso a quello di Luca (d'ora in poi indicato con L).

1. Come prima cosa, per ogni nodo v memorizziamo due valori: `distanza`$[v]$ indica la distanza tra il nodo v e il nodo W, ovvero il minimo numero di archi necessari per raggiungere v a partire da W, mentre `velocità`$[v]$ indica la minima velocità di trasmissione tra tutti i percorsi di lunghezza minima da W a v. Inizialmente segniamo che `distanza`$[v]$ = `velocità`$[v] = \infty$ per ogni nodo $v \neq W$. Per quanto riguarda il nodo W, invece, segniamo `distanza`$[W] = 0$ e `velocità`$[W] = \infty$. Al termine dell'esecuzione dell'algoritmo tutti i nodi avranno i valori `distanza` e `velocità` corretti. La risposta al problema sarà quindi `velocità`$[L]$.
2. Consideriamo i nodi *in ordine di distanza da* W, mano a mano che ci espandiamo. Supponiamo di star considerando il nodo v. Per ogni vicino w di v sia c la capacità dell'arco $v - w$; se risulta che `distanza`$[w] >$ `distanza`$[v]$ eseguiamo il doppio assegnamento `distanza`$[w] =$ `distanza`$[v] + 1$, `velocità`$[w] = \min\{$`velocità`$[w]$, `velocità`$[v], c\}$.



Le figure (2a) – (2h) mostrano alcuni passi dell'algoritmo (sono state omesse alcune coppie di nodi per cui non valeva la disuguaglianza `distanza`$[w] >$ `distanza`$[v]$). Per dimostrare che l'algoritmo è a tutti gli effetti corretto, cominciamo col notare che non appena il valore `distanza` di un nodo risulta diverso da ∞, esso rappresenta effettivamente la distanza, in termini di archi, tra il nodo W e il nodo v. Infatti, se immaginiamo di omettere completamente l'informazione `velocità`, l'algoritmo coincide con quello della *visita in ampiezza* di un grafo, la cui correttezza viene qui data per assodata. Notiamo inoltre che dal momento che il grafo è connesso, al termine dell'esecuzione dell'algoritmo ogni nodo è stato processato.

Rimane da dimostrare che al termine dell'algoritmo le informazioni `velocità` risultano corrette per ogni nodo[3]. A tal fine, possiamo procedere per induzione sulla frontiera dei nodi: supponiamo di aver già processato tutti i nodi a distanza $< k$ da W, e che l'informazione `velocità` trovate per tali nodi sia corretta: la tesi vale per la base di induzione $k = 1$, ovvero per il singolo nodo W. Vogliamo ora processare tutti i nodi a distanza esattamente k, e dimostrare che l'algoritmo attribuisce a tutti questi un valore `velocità` effettivamente corretto.

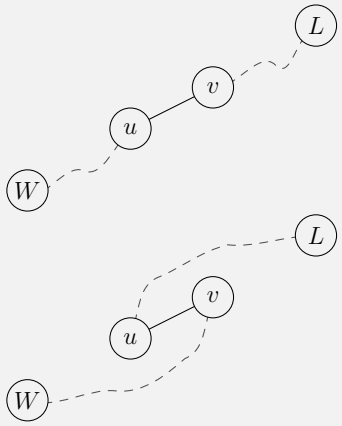
Sia v uno qualunque dei nodi a distanza k da W: dimostriamo che al termine dell'esecuzione dell'algoritmo `velocità`$[v]$ contiene effettivamente la minima velocità di trasmissione tra tutti i percorsi composti da k archi tra il nodo W e il nodo v. Notiamo che ogni percorso composto da k archi tra i nodi W e v necessariamente prevede come penultimo nodo un nodo u a distanza $k-1$ da W. Possiamo allora immaginare di scegliere in tutti i modi possibili l'ultimo arco $u-v$ del percorso, con u nodo a distanza $k-1$ da W, e cercare il percorso composto da $k-1$ archi con la velocità di trasmissione minima da W a u. Poiché per ipotesi induttiva i valori `velocità` di tutti i nodi a distanza $k-1$ da W contengono informazioni corrette, possiamo concludere che

$$\texttt{velocità}[v] = \min_u\{\texttt{velocità}[u], \text{capacità dell'arco } u-v\},$$

dove il minimo va inteso tra tutti i nodi u adiacenti a v e posti a distanza $k-1$ da W. In effetti è facile convincersi che questo è esattamente il calcolo eseguito dall'algoritmo.

■ Secondo approccio

La seconda soluzione nasce dall'osservazione che è possibile determinare in tempo costante se un particolare arco appartiene ad un percorso di lunghezza minima (in termini di archi) tra il PC di William e quello di Luca. Per ogni nodo v del grafo memorizziamo due valori `distanza_da_W` e `distanza_da_L`: essi rappresentano rispettivamente la distanza tra il computer W di William e il computer L di Luca, ovvero il minimo numero di archi che separano v dal computer indicato. Possiamo precalcolare questi valori tramite due *visite in ampiezza del grafo* (BFS), una a partire da W e una a partire da L.



Indichiamo con d la distanza tra i nodi W e L: d coincide col valore `distanza_da_W`$[L]$, ovvero col valore `distanza_da_L`$[W]$. È facile convincersi del fatto che l'arco $u-v$ appartiene ad un percorso di lunghezza minima tra L e W se e solo se vale almeno una delle due condizioni:

- `distanza_da_L`$[u]$ + `distanza_da_W`$[v] + 1 = d$, oppure
- `distanza_da_L`$[v]$ + `distanza_da_W`$[u] + 1 = d$.

Possiamo dunque iterare su tutti gli archi: la risposta al problema è pari alla capacità minima tra le capacità degli archi che appartengono ad almeno un percorso di lunghezza minima tra i nodi W e L.

[3] Si assume corretto per W il valore fittizio `velocità`$[W] = \infty$.

Esempio di codice C++11

Proponiamo qui un'implementazione per entrambe le soluzioni.

■ Primo approccio

```cpp
#include <queue>
#include <vector>
#include <limits>
#include <algorithm>

const int MAXN = 100000;
// Per nostra convenienza definiamo infinito un valore che non possa essere confuso
// con nessuna distanza né velocità associate ad un nodo
const int INF = std::numeric_limits<int>::max();

// adj[i] contiene tutti gli archi che ammettono i come primo estremo. Ogni arco è rappresentato
// da una struttura arco_t che memorizza il secondo estremo e la capacità dell'arco
struct arco_t {
    int estremo, capacita;

    arco_t(int estremo, int capacita): estremo(estremo), capacita(capacita) {}
};

std::vector<arco_t> adj[MAXN + 1];
int distanza[MAXN + 1], velocita[MAXN + 1];

int Analizza(int N, int M, int W, int L, int arco_da[], int arco_a[], int capacita[], int, int) {
    // Costruiamo i vettori di adiacenza di tutti i nodi
    for (int i = 0; i < M; i++) {
        adj[arco_da[i]].emplace_back(arco_a[i], capacita[i]);
        adj[arco_a[i]].emplace_back(arco_da[i], capacita[i]);
    }

    // Passo 1: inizializziamo a INF gli array distanza e velocità, ad eccezione di
    // distanza[W], che invece vale 0
    std::fill(distanza, distanza + N + 1, INF);
    std::fill(velocita, velocita + N + 1, INF);
    distanza[W] = 0;

    // Passo 2: Processa i nodi in ordine di distanza da W
    std::queue<int> Q;
    Q.push(W);
    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();

        // Iteriamo su tutti i vicini w del nodo v
        for (const auto& arco: adj[v]) {
            int w = arco.estremo;
            int c = arco.capacita;

            if (distanza[w] > distanza[v]) {
                // Se è la prima volta che incontriamo il nodo w, mettiamolo nella coda
                // dei nodi da processare
                if (distanza[w] == INF)
                    Q.push(w);

                // Aggiorna distanza e velocità del nodo w.
                distanza[w] = distanza[v] + 1;
                if (velocita[w] > std::min(velocita[v], c))
                    velocita[w] = std::min(velocita[v], c);
            }
        }
    }

    return velocita[L];
}
```

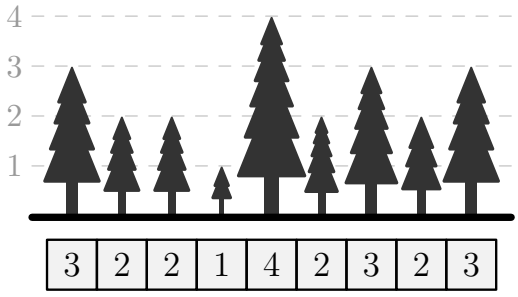
Secondo approccio

```cpp
#include <queue>
#include <vector>
#include <limits>
#include <algorithm>

const int MAXN = 100000;
// Per nostra convenienza definiamo infinito un valore che non possa essere confuso
// con nessuna distanza né velocità associate ad un nodo
const int INF = std::numeric_limits<int>::max();

// adj[i] contiene tutti tutti i vicini del nodo i
std::vector<int> adj[MAXN + 1];
int distanza_da_W[MAXN + 1], distanza_da_L[MAXN + 1];

// Esegue una visita in ampiezza a partire dal nodo sorgente. Scrive le distanza ottenute nel
// vettore puntato da distanza
void BFS(int sorgente, int distanza[]) {
    std::fill(distanza, distanza + MAXN + 1, INF);
    distanza[sorgente] = 0;

    std::queue<int> Q;
    Q.push(sorgente);

    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();

        for (const int w: adj[v]) {
            if (distanza[w] == INF) {
                distanza[w] = distanza[v] + 1;
                Q.push(w);
            }
        }
    }
}

int Analizza(int N, int M, int W, int L, int arco_da[], int arco_a[], int capacita[], int, int) {
    // Costruiamo i vettori di adiacenza di tutti i nodi
    for (int i = 0; i < M; i++) {
        adj[arco_da[i]].push_back(arco_a[i]);
        adj[arco_a[i]].push_back(arco_da[i]);
    }

    // Eseguiamo le visite in ampiezza
    BFS(W, distanza_da_W);
    BFS(L, distanza_da_L);
    int d = distanza_da_W[L];

    // Infine iteriamo sugli archi
    int risposta = INF;
    for (int i = 0; i < M; i++) {
        int u = arco_da[i], v = arco_a[i];

        if (distanza_da_W[u] + distanza_da_L[v] + 1 == d ||
            distanza_da_W[v] + distanza_da_L[u] + 1 == d)
        {
            risposta = std::min(risposta, capacita[i]);
        }
    }

    return risposta;
}
```

Taglialegna (`taglialegna`)

Limite di tempo:	1.0 secondi
Limite di memoria:	256 MiB

La Abbatti S.p.A. è una grossa azienda che lavora nel settore del disboscamento. In particolare, nel tempo si è specializzata nel taglio degli *alberi cortecciosi*, una tipologia di alberi estremamente alti, robusti e ostinati. Si tratta di una specie molto ordinata: i boschi formati da questi alberi consistono in una lunghissima fila di tronchi disposti lungo una fila orizzontale a esattamente un decametro l'uno dall'altro. Ogni albero ha una altezza, espressa da un numero (positivo) di decametri.



(i valori rappresentano le altezze in decametri)

Il taglio di un albero corteccioso è un compito delicato e, nonostante l'uso delle più avanzate tecnologie di abbattimento, richiede comunque molto tempo, data la loro cortecciosità. Gli operai sono in grado di segare i tronchi in modo che l'albero cada a destra o a sinistra, secondo la loro scelta.

Quando un albero corteccioso viene tagliato e cade, si abbatte sugli eventuali alberi non ancora tagliati che si trovano nella traiettoria della caduta, ovvero tutti quegli alberi non ancora tagliati che si trovano ad una distanza strettamente minore dell'altezza dell'albero appena segato, nella direzione della caduta. Data la mole degli alberi cortecciosi, gli alberi colpiti dalla caduta vengono a lora volta spezzati alla base, cadendo nella direzione dell'urto, innescando un effetto domino.

Per assicurarsi il primato nel settore, la Abbatti S.p.A. ha deciso di installare un sistema in grado di analizzare il bosco, determinando quali alberi gli operai dovranno segare, nonchè la direzione della loro caduta, affinchè tutti gli alberi cortecciosi risultino abbattuti alla fine del processo. Naturalmente, il numero di alberi da far tagliare agli operai deve essere il minore possibile, per contenere i costi. In quanto consulente informatico della società, sei incaricato di implementare il sistema.

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** [**5 punti**]: Casi d'esempio.
- **Subtask 2** [**9 punti**]: Gli alberi possono essere alti solo 1 o 2 decametri.
- **Subtask 3** [**20 punti**]: $N \leq 50$.
- **Subtask 4** [**19 punti**]: $N \leq 400$.
- **Subtask 5** [**22 punti**]: $N \leq 5000$.
- **Subtask 6** [**14 punti**]: $N \leq 100\,000$.
- **Subtask 7** [**11 punti**]: Nessuna limitazione specifica (vedi la sezione **Assunzioni**).

Implementazione

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

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

Dovrai implementare la seguente funzione:

C/C++	`void Pianifica(int N, int altezza[]);`
Pascal	`procedure Pianifica(N: longint; var altezza: array of longint);`

N è il numero di alberi cortecciosi nel bosco, mentre `altezza[i]` contiene, per ogni $0 \leq i < N$, l'altezza, in decametri, dell'i-esimo albero corteccioso a partire da sinistra. La funzione dovrà chiamare la routine già implementata

C/C++	`void Abbatti(int indice, int direzione);`
Pascal	`procedure Abbatti(indice: longint; direzione: longint);`

dove `indice` è l'indice (da 0 a $N-1$) dell'albero da abbattere, e `direzione` è un intero che vale 0 se l'albero deve cadere a sinistra e 1 se invece deve cadere a destra.

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 `Pianifica` che dovete implementare. Il grader scrive sul file `output.txt` il resoconto delle chiamate ad `Abbatti`.

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

- Riga 1: contiene l'intero N, il numero di alberi cortecciosi nel bosco (consigliamo di non superare il valore 50 data l'inefficienza del grader fornito).
- Riga 2: contiene N interi, di cui l'i-esimo rappresenta l'altezza in decametri dell'albero di indice i.

Il file `output.txt` invece ha questo formato:

- Righe dalla 1 in poi: La i-esima di queste righe contiene i due parametri passati alla funzione `Abbatti`, cioè l'indice dell'albero tagliato e la direzione della caduta (0 indica sinistra e 1 indica destra), nell'ordine delle chiamate.

Assunzioni

- $1 \leq N \leq 2\,000\,000$.
- L'altezza di ogni albero è un numero intero di decametri compreso tra 1 e $1\,000\,000$.
- Un'esecuzione del programma viene considerata errata se:
  - Al termine della chiamata a `Pianifica` tutti gli alberi sono caduti, ma il numero di alberi segati dagli operai non è il minimo possibile.

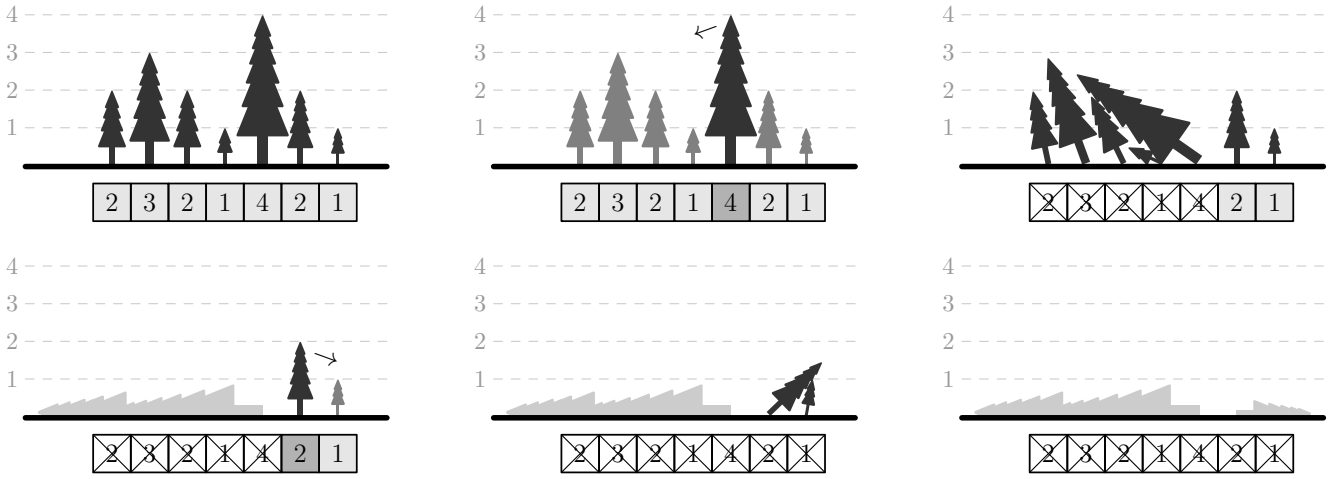
– Al termine della chiamata a `Pianifica` non tutti gli alberi sono caduti.
– Viene fatta una chiamata ad `Abbatti` con un indice o una direzione non validi.
– Viene fatta una chiamata ad `Abbatti` con l'indice di un albero già caduto, direttamente ad opera degli operai o indirettamente a seguito dell'urto con un altro albero.

Esempi di input/output

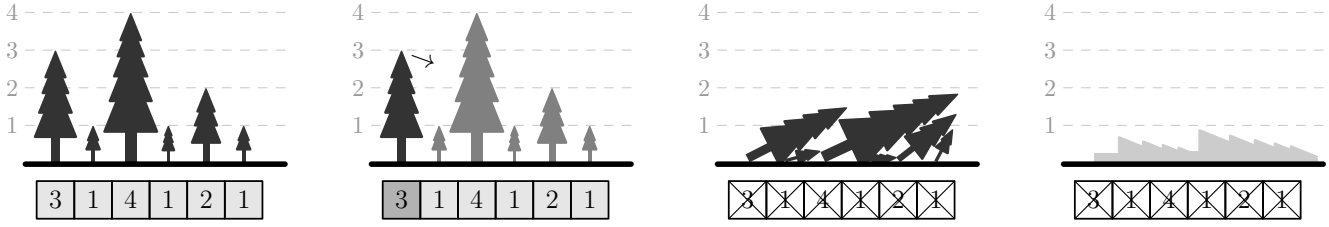
input.txt	output.txt
7 2 3 2 1 4 2 1	4 0 5 1
6 3 1 4 1 2 1	0 1

Spiegazione

Nel **primo caso d'esempio** è possibile abbattere tutti gli alberi segando il quinto albero (alto 4 decametri) facendolo cadere a sinistra, e il sesto albero (alto 2 decametri) facendolo cadere a destra. Il primo albero tagliato innesca un effetto domino che abbatte tutti gli alberi alla sua sinistra, mentre il secondo abbatte l'ultimo albero nella caduta.



Nel **secondo caso d'esempio** tagliando il primo albero in modo che cada verso destra vengono abbattuti anche tutti gli alberi rimanenti.



Soluzione

■ Introduzione e concetti generali

Risolveremo questo problema mediante la tecnica della *programmazione dinamica*. Presenteremo inizialmente una soluzione quadratica nel numero di alberi, e successivamente mostreremo come rendere la stessa soluzione più efficiente.

Entrambe le soluzioni affondano le proprie radici nella stessa osservazione fondamentale, cioè il fatto che possiamo supporre senza perdita di generalità che il primo taglio effettuato dagli operai abbia come effetto quello di abbattere il primo albero. A tal fine, gli operai hanno due opzioni:

- tagliare, facendolo cadere a destra, l'albero 1, oppure
- tagliare, facendolo cadere a sinistra, un albero la cui caduta provochi l'abbattimento dell'albero 1.

In entrambi gli scenari, dopo il primo taglio saranno rimasti intatti solo gli alberi da un certo indice in poi, e ci saremo ridotti a dover abbattere, col minor numero possibile di tagli, un numero minore di alberi rispetto al caso iniziale.

■ Una soluzione $O(N^2)$

Definiamo alcuni concetti che torneranno più volte utili nel corso della spiegazione di entrambe le soluzioni. Chiamiamo `rep` *dell'albero* i, indicato con `rep`$[i]$, l'indice dell'albero più a destra che viene abbattuto dalla caduta dell'albero i, quando questo cade verso destra; analogamente, chiamiamo `lep` *dell'albero* i, indicato con `lep`$[i]$, l'indice dell'albero più a sinistra che viene abbattuto dalla caduta dell'albero i, quando questo cade verso sinistra. Per rendere più chiaro il significato di `lep` e `rep` osserviamo l'esempio in figura 1.

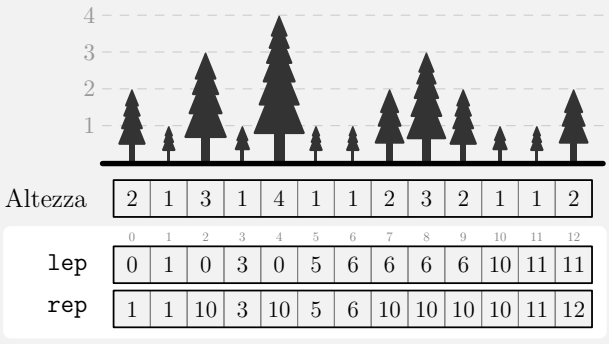


Figura 1

Nell'esempio il `rep` dell'albero 7 è 10, perché l'albero 7, una volta tagliato e lasciato cadere a destra, abbatte l'albero 8, che cade a sua volta abbattendo gli alberi 9 e 10; il `lep` dell'albero 6 invece è 6, perché l'albero in questione è alto 1 decametro e come tale non è in grado di abbattere nessun altro albero.

Calcolare `rep` e `lep` di ogni albero è semplice: il `rep` dell'albero i è pari al maggiore tra i `rep` degli alberi che i abbatte quando cade verso destra, o, nel caso in cui i non abbatta alcun albero nella caduta, a i stesso. Analogamente, il `lep` dell'albero i è pari al minore tra i `lep` degli alberi che i abbatte quando cade verso sinistra, o, nel caso in cui i non abbatta alcun albero nella caduta, a i stesso.

```
1  // Costruzione di rep
2  for (int i = N - 1; i >= 0; i--) {
3      rep[i] = i;
4      for (int j = i; j < i + H[i] && j < N; j++)
5          rep[i] = max(rep[i], rep[j]);
6  }
```

```
1  // Costruzione di lep
2  for (int i = 0; i < N; i++) {
3      lep[i] = i;
4      for (int j = i; j > i - H[i] && j >= 0; j--)
5          lep[i] = min(lep[i], lep[j]);
6  }
```

A questo punto possiamo implementare l'osservazione dell'introduzione, giungendo alla formulazione top-down dell'algoritmo della prossima pagina. Volendo privilegiare la trasparenza nella spiegazione, abbiamo volutamente ignorato la parte di memoizzazione (*memoization*), che non può invece essere trascurata in una implementazione reale.

```cpp
const int INF = numeric_limits<int>::max();
enum direzione_t {SINISTRA, DESTRA};

// Struttura info_t. I significati dei vari membri sono spiegati poco più sotto.
struct info_t {
    int numero_tagli = INF;
    int primo_albero;
    direzione_t direzione;
};

// risolvi(i) ritorna un oggetto info_t, che contiene
//   - numero_tagli: il minimo numero di tagli da effettuare per abbattere tutti gli alberi da
//                   i a N-1 inclusi.
//   - primo_albero: l'indice del primo albero da tagliare
//   - direzione:    la direzione della caduta del primo albero
info_t risolvi(int i) {
    info_t risposta;

    if (i == N) {
        // Se non ci sono alberi da tagliare, numero_tagli = 0
        risposta.numero_tagli = 0;
    } else {
        // Primo caso: abbatti i a destra
        risposta.numero_tagli = risolvi(rep[i] + 1).numero_tagli + 1;
        risposta.primo_albero = i;
        risposta.direzione = DESTRA;

        // Secondo caso: abbatti a sinistra un albero alla destra di i che, nella caduta, abbatta anche i
        for (int j = i; j < N; j++) {
            if (lep[j] <= i) { // Controlla che l'albero j abbatta i cadendo a sinistra
                // Valuta se tagliando l'albero j troviamo una soluzione migliore di quella che conosciamo
                if (risolvi(j + 1).numero_tagli + 1 < risposta.numero_tagli) {
                    risposta.numero_tagli = risolvi(j + 1).numero_tagli + 1;
                    risposta.primo_albero = j;
                    risposta.direzione = SINISTRA;
                }
            }
        }
    }

    return risposta;
}
```

Sfruttando le informazioni calcolate da `risolvi`, è facile ricostruire la sequenza completa dei tagli e risolvere il problema. La soluzione del problema coincide con la chiamata `ricostruisci_tagli(0)` nel codice qui sotto.

```cpp
// ricostruisci_tagli(i) si occupa di tagliare, attraverso opportune chiamate alla funzione Abbatti, il
// minimo numero di alberi affinchè tutti gli alberi da i a N-1 inclusi risultino abbattuti alla fine del
// processo. Riusa internamente le informazioni calcolate dalla funzione risolvi illustrata poco prima
void ricostruisci_tagli(int i) {
    if (i == N)
        return;

    int primo_albero = risolvi(i).primo_albero;
    direzione_t direzione = risolvi(i).direzione;

    Abbatti(primo_albero, direzione);

    if (direzione == SINISTRA)
        ricostruisci_tagli(primo_albero + 1);
    else
        ricostruisci_tagli(rep[i] + 1);
}
```

Analizziamo ora la complessità computazionale dell'algoritmo proposto. Il calcolo dei valori `lep` e `rep` richiede, per ogni albero, al più $O(N)$ operazioni, dunque il numero di operazioni necessarie per calcolare questi valori per tutti gli N alberi è proporzionale N^2. Analogamente, per calcolare `risolvi`(i) è necessario, al caso peggiore, considerare tutti gli alberi alla destra di i, rendendo il calcolo degli N valori

di `risolvi`(i) quadratico nel numero di alberi. Infine, il numero di operazioni svolte per la ricostruzione della sequenza di tagli è proporzionale al numero di alberi tagliati, dunque la complessità di questa ultima fase è pari a $O(N)$. In totale, quindi, l'intero algoritmo ha complessità $O(N^2)$.

■ Una soluzione $O(N)$

Prima di illustrare la soluzione lineare, introduciamo un altro paio di concetti importanti.

Definiamo *abbattitore di un albero* i, indicato con `abbattitore`$[i]$, l'indice del primo albero a destra di i, se esiste, che abbatte i quando cade verso sinistra; nel caso in cui l'abbattitore di i non esista, assegneremo convenzionalmente `abbattitore`$[i] = \infty$.

Definiamo inoltre *catena di abbattitori dell'albero* i la sequenza formata da i, dall'abbattitore di i, dall'abbattitore dell'abbattitore di i, e così via:

$$\text{catena di abbattitori di } i = i \to \texttt{abbattitore}[i] \to \texttt{abbattitore}[\texttt{abbattitore}[i]] \to \cdots,$$

dove l'iterazione viene troncata nel momento in cui giungiamo ad un albero che non ammette abbattitore. La catena di abbattitori di i non è mai vuota, perché contiene sempre almeno un elemento, cioè i stesso. Per fissare il concetto, consideriamo la figura 2.

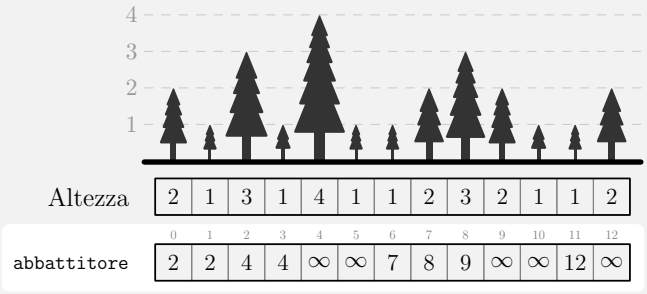


Figura 2

Nel caso in figura, ad esempio, l'abbattitore dell'albero 6 è l'albero 7, perché tra tutti gli alberi che lo abbattono se lasciati cadere a sinistra, 7 è il primo. La catena di abbattitori dell'albero 0 è $0 \to 2 \to 4$; la catena di abbattitori dell'albero 5 invece consiste del solo albero 5.

L'osservazione che permette di rendere la soluzione precedente più efficiente consiste nel notare che la catena di abbattitori dell'albero i è formata da tutti quegli alberi che sono in grado di abbattere i quando vengono lasciati cadere a sinistra. In altre parole, abbiamo appena constatato il fatto che gli alberi che abbattono i cadendo a sinistra sono disposti in maniera molto ordinata, e si raggiungono a partire dall'albero i passando di volta in volta da un albero al suo `abbattitore`.

Mostriamo ora come è possibile costruire velocemente le informazioni `lep`, `rep` e `abbattitore` per ogni albero:

```cpp
// Costruiamo rep in tempo lineare
for (int i = N - 1; i >= 0; i--) {
    rep[i] = i;
    while (rep[i] + 1 < N && rep[i] + 1 < i + H[i])
        rep[i] = rep[rep[i] + 1];
}
```

```cpp
// Costruiamo lep e abbattitore in tempo lineare
for (int i = 0; i < N; i++) {
    lep[i] = i, abbattitore[i] = INF;
    while (lep[i] - 1 >= 0 && lep[i] - 1 > i - H[i]) {
        abbattitore[lep[i] - 1] = i;
        lep[i] = lep[lep[i] - 1];
    }
}
```

L'idea alla base del calcolo è, a tutti gli effetti, quella di simulare l'*effetto domino*. Consideriamo ad esempio il calcolo di `rep` per l'albero i:

- inizialmente viene controllato se i, cadendo, abbatte l'albero $i + 1$;
- se sì, allora tutti gli alberi da $i + 1$ a `rep`$[i + 1]$ sono abbattuti dalla caduta di i, e `rep`$[i]$ viene temporaneamente impostato a `rep`$[i + 1]$;
- successivamente viene controllato se i, continuando la caduta, abbatte anche il primo albero non ancora caduto alla destra di i, cioè `rep`$[i + 1] + 1$;

- se sì, allora tutti gli alberi da `rep`$[i+1]+1$ a `rep`[`rep`$[i+1]+1$] sono abbattuti dalla caduta di i, e `rep`$[i]$ viene temporaneamente impostato a `rep`[`rep`$[i+1]+1$];
- ...

Dimostrare che questo metodo è in effetti capace di calcolare `lep`, `rep` e `abbattitore` di tutti gli alberi in tempo lineare è un semplice esercizio di analisi ammortizzata[4], ma non ce ne occuperemo qui per non appesantire la discussione.

Vediamo ora come è possibile velocizzare il calcolo di `risolvi`(i). L'implementazione che abbiamo visto prima era quadratica a causa del fatto che per ogni albero i è necessario considerare tutti gli alberi alla destra di i in grado di abbatterlo, una quantità di alberi ogni volta potenzialmente dell'ordine di N. Vedremo tra un attimo come evitare questa ricerca inutile, usando le informazioni sull'abbattitore di i.

Definiamo *migliore albero della catena di abbattitori di* i, d'ora in poi indicato con `migliore`$[i]$, l'albero j appartenente alla catena di abbattitori di i per cui è minima la quantità `risolvi`$(j+1)$.`numero_tagli`. Intuitivamente, `migliore`$[i]$ rappresenta, tra tutti gli alberi che sono in grado di abbattere i cadendo a sinistra, quello per cui risulta minimo il numero di tagli necessari per abbattere tutti gli alberi da i a $N-1$.

Ammettendo di essere in grado di calcolare velocemente `migliore`$[i]$ per ogni albero i, la funzione `risolvi` si ridurrebbe semplicemente a:

```
1  // risolvi(i) ritorna un oggetto info_t, che contiene
2  //   - numero_tagli: il minimo numero di tagli da effettuare per abbattere tutti gli alberi da
3  //                   i a N-1 inclusi.
4  //   - primo_albero: l'indice del primo albero da tagliare
5  //   - direzione:    la direzione della caduta del primo albero
6  info_t risolvi(int i) {
7      info_t risposta;
8
9      if (i == N) {
10         // Se non ci sono alberi da tagliare, numero_tagli = 0
11         risposta.numero_tagli = 0;
12     } else {
13         // Primo caso: abbatti i a destra
14         risposta.numero_tagli = risolvi(rep[i] + 1).numero_tagli + 1;
15         risposta.primo_albero = i;
16         risposta.direzione = DESTRA;
17
18         // Secondo caso: abbatti a sinistra migliore[i]
19         int j = migliore(i);
20         if (risolvi(j + 1).numero_tagli + 1 < risposta.numero_tagli) {
21             risposta.numero_tagli = risolvi(j + 1).numero_tagli + 1;
22             risposta.primo_albero = j;
23             risposta.direzione = SINISTRA;
24         }
25     }
26
27     return risposta;
28 }
```

dove la principale differenza con la versione quadratica consiste nell'aver eliminato il ciclo della riga 29.

Rimane solamente da capire come calcolare `migliore`$[i]$ in modo efficiente. Sicuramente, se `abbattitore`$[i] = \infty$, si avrà `migliore`$[i] = i$. Vice versa, supponiamo che esista l'abbattitore di i; per la natura della catena di abbattitori, per determinare il valore di `migliore`$[i]$ è sufficiente confrontare tra di loro gli alberi i e `migliore`[`abbattitore`$[i]$], e scegliere chi tra i due minimizza `risolvi`$(j+1)$.`numero_tagli`. Mostriamo una semplice implementazione in codice di questa idea all'inizio della prossima pagina.

[4] Per un'introduzione all'argomento dell'analisi ammortizzata si consideri ad esempio il documento disponibile al seguente indirizzo: http://goo.gl/7OegpB

```cpp
// migliore(i) ritorna l'albero j appartenente alla catena di abbattitori di i per cui
// è minima la quantità risolvi(j + 1).numero_tagli
int migliore(int i) {
    int risposta = i;

    // Se i ammette un abbattitore, confrontiamo l'albero i, col migliore albero della catena
    // di abbattitori dell'abbattitore di i
    if (abbattitore[i] != INF) {
        // j contiene il migliore albero della catena di abbattitori dell'abbattitore di i
        int j = migliore(abbattitore[i]);

        // Confrontiamo l'albero j con l'albero i
        if (risolvi(j + 1).numero_tagli < risolvi(i + 1).numero_tagli)
            risposta = j;
    }

    return risposta;
}
```

Ora che abbiamo tutti i tasselli del puzzle, non rimane che riutilizzare (inalterata) la funzione `ricostruisci_tagli` per risolvere completamente il problema.