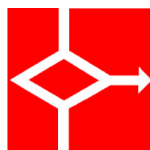
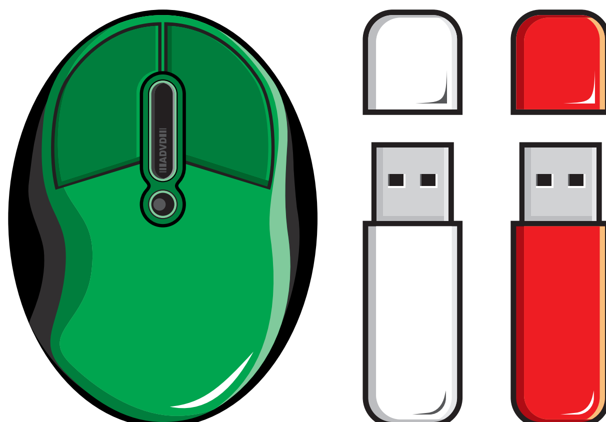


AICA
Associazione Italiana per l'Informatica
ed il Calcolo Automatico



OLIMPIADI ITALIANE DI INFORMATICA

14 APRILE 2015

Selezione territoriale

Testi e soluzioni ufficiali

Testi dei problemi
William Di Luigi, Gabriele Farina, Luigi Laura, Gemma Martini, Romeo Rizzi, Luca Versari

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

Coordinamento
Monica Gati

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Numero semiprimo (semiprimo)

Difficoltà: 1

Gemma ha appena imparato che cos'è un numero semiprimo, e presa dall'euforia non riesce a smettere di parlarne. In particolare, **un numero semiprimo è un intero ≥ 2 che si fattorizza come prodotto di due numeri primi** (non necessariamente distinti).

☞ I numeri primi sono tutti quegli interi ≥ 2 divisibili solo per se stessi e per 1.

Sono quindi esempi di numeri semiprimi i numeri:

- 15, prodotto di 3 e 5.
- 169, prodotto di 13 e 13.

Aiuta Gemma a scrivere un programma che verifichi se un numero N è semiprimo oppure no!

Dati di input

Il file `input.txt` contiene l'unico intero N, di cui Gemma vuole verificare la semiprimalità.

Dati di output

Il file `output.txt` contiene:

- I due primi che fattorizzano N, stampati su un'unica riga, in ordine non-decrescente, se N è semiprimo.
- L'unico intero -1 se N **non** è semiprimo.

Assunzioni

- $2 \leq N \leq 1\,000\,000$.

Esempi di input/output

input.txt	output.txt
961	31 31
884053	101 8753
16	-1

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Supponiamo di conoscere il più piccolo divisore primo del numero N in input. Detto D questo numero, è immediato notare che N è semiprimo se e solo se N/D è un numero primo.

■ Una soluzione lineare

Forti di questa osservazione, possiamo immaginare di scandire in ordine tutti gli interi $2, 3, \ldots, N$ in ordine, alla ricerca del primo valore che divida N. Questo valore infatti coinciderà proprio con D, il più piccolo divisore primo di N (perché?). Una volta determinato D, calcoliamo N/D.

La verifica di primalità di un dato numero Q (in questo caso, di N/D) è un'operazione semplice: infatti, come prima è sufficiente scandire in ordine tutti gli interi $2, 3, \ldots$ fino a $Q - 1$ compreso. Se nessuno di questi interi risulta un divisore di Q, allora Q è primo. Vale anche il contrario: se almeno uno degli interi è un divisore di Q, allora Q è un numero composto.

La complessità di questa soluzione cresce linearmente con N. Infatti, per determinare D è necessario, nel caso peggiore, scandire tutti i numeri da 2 fino ad N, per un totale di $O(N)$ controlli. Trovato il valore giusto di D, dobbiamo verificare se il numero N/D è primo. L'algoritmo descritto sopra per determinare se un generico intero Q è primo esegue $O(Q)$ controlli, perciò ha un tempo di esecuzione che cresce linearmente con Q. Notando che $N/D \leq N$, si conclude che la complessità dell'intero algoritmo è proprio $O(N)$.

Essendo in tutte le istanze di prova $N \leq 1\,000\,000$, l'algoritmo appena descritto era più che sufficiente per guadagnare la totalità dei punti.

■ Una soluzione ancora più efficiente

Anche se non necessario per risolvere completamente il problema proposto, proponiamo un algoritmo più efficiente.

L'osservazione centrale per ridurre la complessità dell'algoritmo è la seguente:

> Se un intero Q non possiede alcun divisore maggiore di 1 e minore o uguale a $\sqrt{Q}$, allora è primo.

▷ **Dimostrazione:** la dimostrazione dell'affermazione è semplice. Supponiamo per assurdo che l'affermazione sia falsa e che esista un numero composto Q che non ammetta divisori maggiori di 1 e minori o uguali di $\sqrt{Q}$. Dato che Q è composto, devono esistere due suoi divisori A e B, entrambi maggiori di 1, tali che $Q = A \times B$. Per l'ipotesi fatta, sia A che B devono essere entrambi strettamente maggiori di $\sqrt{Q}$. Questo porta immediatamente ad una contraddizione: infatti, se $A, B > \sqrt{Q}$, sicuramente $A \times B > Q$, contraddicendo quello che abbiamo appena detto. □

Come prima conseguenza di questo lemma, possiamo interrompere la ricerca del più piccolo divisore primo D di N non appena $D > \sqrt{N}$. Infatti, in quel caso abbiamo la certezza che N è un numero primo, quindi non è semiprimo, e stampiamo immediatamente `-1` in output.

Allo stesso modo, supposto di aver trovato D, possiamo determinare se N/D è primo con $O(\sqrt{N/D})$ controlli, interrompendo il ciclo che cerca un divisore di N/D nonappena il potenziale divisore supera il valore $\sqrt{N/D}$. Essendo $N/D < N$, il controllo di primalità di N/D comporta un numero di operazioni inferiore a $\sqrt{N}$.

L'algoritmo risultante ha pertanto complessità $O(\sqrt{N})$.

Esempio di codice C++11

■ Una soluzione lineare

```
#include <iostream>

// Ritorna true se Q è primo, false altrimenti
bool primo(unsigned Q) {
    if (Q < 2)
        return false;
    for (unsigned divisore = 2; divisore < Q; ++divisore)
        if (Q % divisore == 0)
            return false;
    return true;
}

// Ritorna il più piccolo divisore primo di N
unsigned trova_D(unsigned N) {
    for (unsigned D = 2; D <= N; ++D)
        if (N % D == 0)
            return D;
}

int main() {
    // Input/output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    unsigned N;
    std::cin >> N;

    unsigned D = trova_D(N);
    if (primo(N/D))
        std::cout << D << " " << N/D << std::endl;
    else
        std::cout << -1 << std::endl;
}
```

■ Una soluzione ancora più efficiente

```
#include <iostream>

// Ritorna true se Q è primo, false altrimenti
bool primo(unsigned Q) {
    if (Q < 2)
        return false;
    for (unsigned divisore = 2; divisore * divisore <= Q; ++divisore)
        if (Q % divisore == 0)
            return false;
    return true;
}

// Ritorna il più piccolo divisore primo di N
unsigned trova_D(unsigned N) {
    for (unsigned D = 2; D * D <= N; ++D)
        if (N % D == 0)
            return D;
    return N;
}

int main() {
    // Input/output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    unsigned N;
    std::cin >> N;

    unsigned D = trova_D(N);
    if (primo(N/D))
        std::cout << D << " " << N/D << std::endl;
    else
        std::cout << -1 << std::endl;
}
```

Rispetta i versi (disuguaglianze)

Difficoltà: 2

Gabriele ha un nuovo rompicapo preferito, chiamato "Rispetta i versi". Si tratta di un solitario giocato su una griglia formata da N caselle separate da un simbolo di disuguaglianza; in figura è mostrato un esempio con $N = 6$.

L'obiettivo del gioco è quello di riempire le celle vuote con tutti i numeri da 1 a N (ogni numero deve comparire esattamente una volta), in modo da rispettare le disuguaglianze tra caselle adiacenti. Per la griglia della figura, una delle possibili soluzioni al rompicapo è la seguente:

Dati di input

Il file `input.txt` contiene due righe di testo. Sulla prima è presente l'intero N, il numero di caselle del gioco. Sulla seconda è presente una stringa di $N - 1$ caratteri, ognuno dei quali può essere solo `<` o `>`, che descrive i vincoli tra le caselle, da sinistra a destra.

Dati di output

Il file `output.txt` contiene su una sola riga una qualunque permutazione dei numeri da 1 a N - separati tra loro da uno spazio - che risolve il rompicapo. I numeri corrispondono ai valori scritti nelle caselle, leggendo da sinistra verso destra.

Assunzioni

- $2 \leq N \leq 100\,000$.
- Nel 30% dei casi, il valore di N non supera 10.
- Nel 60% dei casi, il valore di N non supera 20.
- Si garantisce l'esistenza di almeno una soluzione per ciascuno dei casi di test utilizzati nella verifica del funzionamento del programma.

Esempi di input/output

input.txt	output.txt
`6` `<><<>`	`2 5 1 3 6 4`
`5` `>><<`	`5 3 1 2 4`
`8` `>><>><>`	`6 5 4 7 3 2 8 1`

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Era possibile risolvere questo problema seguendo diversi approcci. Proponiamo qui sotto i due che riteniamo più istruttivi.

■ Una soluzione greedy

Consideriamo il seguente pseudocodice:

Algoritmo 1 Soluzione che assegna i numeri in modo greedy

```
 1: procedure AssegnaGreedy(simboli, N)
 2:     min ← 1
 3:     max ← N
 4:     for i ← 1, 2, . . . , N − 1 do
 5:         if simboli[i] = “<” then                              ▷ Caso 1: simbolo “<”
 6:             print min
 7:             min ← min + 1
 8:         else                                                   ▷ Caso 2: simbolo “>”
 9:             print max
10:             max ← max − 1
11:     print min        ▷ Qui si avrà min = max, quindi è indifferente usare min o max
```

Omettiamo una dimostrazione completamente formale della correttezza dell'algoritmo: limitiamoci a notare che è facile convincersi che questo algoritmo greedy produce sempre una permutazione corretta. Infatti, da una parte ogni numero da inserire nella griglia viene utilizzato esattamente una volta, mentre dall'altra è chiaro che il numero da inserire nella casella i rispetta sempre il segno di disuguaglianza tra le caselle $i-1$ e i, per ogni $i > 1$.

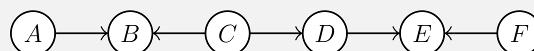
In effetti, come vedremo nella prossima sezione, è possibile rileggere l'algoritmo in termini di operazioni su un particolare grafo, da cui segue direttamente la correttezza dell'approccio.

Questo semplice algoritmo, di complessità lineare in N, era sufficiente a guadagnare la totalità del punteggio.

■ Una soluzione coi grafi di precedenza

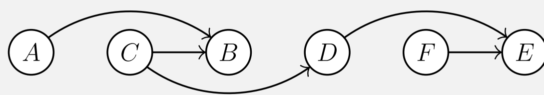
È possibile adottare un approccio più teorico per risolvere il problema, che illustriamo riprendendo l'esempio proposto dal testo, etichettando per comodità le caselle della griglia con le lettere da A a F.

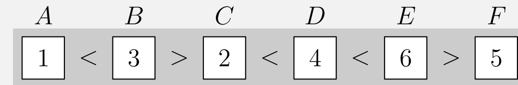
Disegniamo un grafo in cui i nodi rappresentano le caselle, e gli archi sono definiti tramite le relazioni tra le caselle adiacenti. In particolare, la relazione $X < Y$ definisce un arco che va da X a Y, mentre la relazione $X > Y$ definisce un arco che va da Y a X.

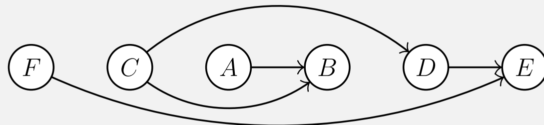


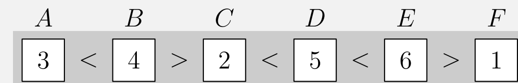
Notiamo che questo grafo diretto, che esprime le relazioni di precedenza tra coppie di caselle, non può avere cicli; in gergo tecnico è denominato “DAG”, dall'inglese *Directed Acyclic Graph*.

La teoria dei DAG, impiegati per modellare problemi di precedenza ben più sofisticati del nostro, assicura che è sempre possibile riordinare i nodi del grafo in modo che tutte le frecce puntino da sinistra verso destra. Questo particolare ordinamento, detto ordinamento topologico[1], in generale non è unico; qui sotto mostriamo, nella colonna di sinistra, due ordinamenti tra i tanti possibili per il grafo che stiamo considerando.









Una volta scelto un ordinamento topologico, è immediato vedere che assegnare in ordine i valori $1, 2, \ldots$ ai nodi produce una permutazione valida ai fini del problema. Le permutazioni indotte dai due ordinamenti topologici sopra sono quelli mostrati nella colonna destra.

È possibile costruire un ordinamento topologico del grafo in tempo lineare nella dimensione del grafo, ovvero in questo lineare in N. Non entreremo nei dettagli di questo algoritmo, seppur semplice, e lasciamo alla curiosità dei lettori capirne il dettaglio del funzionamento. Ad ogni modo, ne proponiamo una implementazione in linguaggio `C++11`, all'interno delle prossime pagine.

Approfondimento. Come già accennato, possiamo interpretare la soluzione greedy esposta alla sezione precedente, mostrando che l'assegnamento prodotto è ottenuto costruendo un ordinamento topologico "dietro le quinte". Supponiamo che il primo simbolo sia "`<`": avremo che il primo nodo (ovvero A) sarà una *sorgente* del grafo delle precedenze, ovvero un nodo che non presenta archi *entranti*. Esiste sicuramente un ordinamento topologico in cui A ricopre la prima posizione, e per questo non perdiamo nulla ad assegnare ad A il più basso numero disponibile.

Analogamente, se il primo simbolo è "`>`", avremo che il nodo A sarà un *pozzo* del DAG, ovvero un nodo che non presenta archi *uscenti*. Come prima, non perdiamo nulla ad assumere che questo pozzo ricopra l'ultima posizione dell'ordinamento, e per questo gli attribuiamo il numero più grande disponibile.

Una volta allocato il nodo A, possiamo pensare di strapparlo dal grafo, assieme a tutti gli archi ad esso incidenti; questa operazione potenzialmente crea nuove sorgenti o nuovi pozzi. Possiamo ora reiterare il ragionamento, costruendo induttivamente un ordinamento topologico valido, che coincide con quello della soluzione greedy.

[1] https://it.wikipedia.org/wiki/Ordinamento_topologico

Esempio di codice C++11

■ Una soluzione greedy

```
1  #include <iostream>
2  #include <string>
3
4  int main() {
5      // Input/output da/su file
6      freopen("input.txt", "r", stdin);
7      freopen("output.txt", "w", stdout);
8
9      unsigned N;
10     std::string segni;
11
12     std::cin >> N >> segni;
13
14     unsigned min = 1, max = N;
15     for (unsigned i = 0; i < N - 1; ++i) {
16         if (segni[i] == '<')
17             std::cout << min++ << " ";
18         else
19             std::cout << max-- << " ";
20     }
21
22     std::cout << min << std::endl;
23 }
```

■ Una soluzione brute force (per il 30% dei punti)

```
1  #include <iostream>
2  #include <string>
3  #include <algorithm>
4  #include <iterator>
5
6  int main() {
7      // Input/output da/su file
8      freopen("input.txt", "r", stdin);
9      freopen("output.txt", "w", stdout);
10
11     int N;
12     std::string segni;
13
14     std::cin >> N >> segni;
15     std::vector<int> permutazione(N);
16
17     // permutazione = {1, 2, 3, ...}
18     std::iota(permutazione.begin(), permutazione.end(), 1);
19
20     do {
21         bool rispetta = true;
22
23         // Verifica che tutti i segni siano rispettati
24         for (int i = 0; i < N - 1; i++) {
25             if (segni[i] == '<') {
26                 rispetta &= (permutazione[i] < permutazione[i + 1]);
27             } else {
28                 rispetta &= (permutazione[i] > permutazione[i + 1]);
29             }
30         }
31
32         if (rispetta) {
33             // Stampa gli elementi della permutazione (separati da spazio)
34             std::copy(permutazione.begin(), permutazione.end(),
35                       std::ostream_iterator<int>(std::cout, " "));
36             std::cout << std::endl;
37
38             // Ci basta una sola soluzione
39             break;
40         }
41     } while (std::next_permutation(permutazione.begin(), permutazione.end()));
42 }
```

Una soluzione coi grafi di precedenza

```
#include <iostream>
#include <vector>
#include <stack>
#include <string>

typedef unsigned vertice_t;

std::vector<std::vector<vertice_t>> vicini;
std::stack<vertice_t> ordinamento;
std::vector<bool> visto;

// Trova ricorsivamente un ordinamento topologico del DAG
void ordinamento_topologico(vertice_t u) {
    if (!visto[u]) {
        visto[u] = true;
        for (const vertice_t& v: vicini[u])
            ordinamento_topologico(v);
        ordinamento.push(u);
    }
}

int main() {
    // Input/output da/su file
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    unsigned N;
    std::string s;

    std::cin >> N >> s;
    vicini.resize(N);

    // Costruisci il grafo delle precedenze
    for (vertice_t i = 0; i < N - 1; ++i) {
        if (s[i] == '<')
            vicini[i].push_back(i + 1);
        else
            vicini[i + 1].push_back(i);
    }

    // Inizializza il vector globale "visto" con N valori false,
    // poi calcola un ordinamento topologico del grafo
    visto.resize(N, false);
    for (vertice_t i = 0; i < N; i++) {
        if (!visto[i])
            ordinamento_topologico(i);
    }

    // Assegna i numeri alle celle della griglia
    unsigned valore = 1;
    std::vector<unsigned> soluzione(N);
    while (!ordinamento.empty()) {
        // Estrai il prossimo nodo nell'ordinamento
        vertice_t cella = ordinamento.top();
        ordinamento.pop();

        // Assegna il valore
        soluzione[cella] = valore;
        ++valore;
    }

    for (vertice_t i = 0; i < N; i++)
        std::cout << soluzione[i] << " ";

    std::cout << std::endl;
}
```

Corsa mattutina (`footing`)

Difficoltà: 2

William sta pensando di trasferirsi in una nuova città e vuole selezionare, tra le varie possibilità, quella che si concilia meglio con la sua routine mattutina. Infatti, William è abituato a fare una corsetta attorno al proprio isolato tutte le mattine, e teme che traslocando debba rinunciare a questo hobby, qualora l'isolato in cui verrebbe a trovarsi fosse troppo grande.

La mappa della città si può rappresentare come un insieme di strade e di incroci tra queste. A ogni incrocio c'è una casa e le strade possono essere percorse in entrambi i sensi. Le case sono numerate da 1 a N. Per evitare di annoiarsi, William non ha intenzione di fare corsette che passino due volte davanti alla stessa casa, ad eccezione della sua (infatti la corsetta deve necessariamente cominciare e terminare nella stessa casa). Questo tipo di percorso prende il nome di *ciclo semplice*.

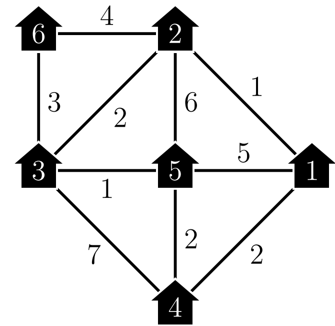
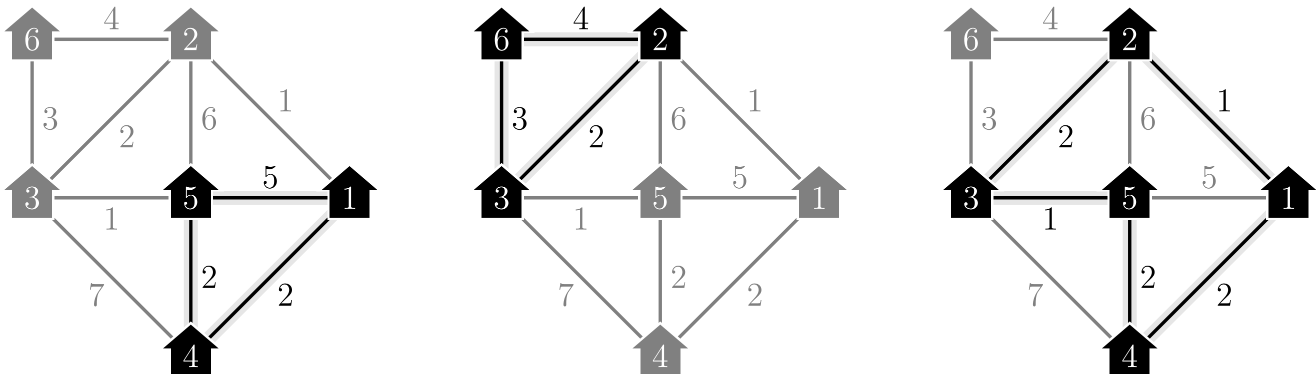


Figura 1: *La mappa della città descritta nel primo input di esempio.*

Nonostante i buoni propositi, William è molto pigro; per questo motivo ha intenzione di rendere la sua corsetta mattutina il più breve possibile: aiutalo scrivendo un programma che prenda in input la mappa di una città e determini la lunghezza del *ciclo semplice* più corto. Con questa informazione, William potrà decidere se trasferirsi nella nuova città, ovviamente solo se riuscirà poi ad andare ad abitare in una delle case che appartengono a questo percorso.

Si prenda ad esempio la mappa della città in Figura 1 (dove il numero a fianco di ogni strada indica la lunghezza della strada), alcuni dei suoi cicli semplici sono i seguenti:



Come si può vedere, i primi due cicli evidenziati hanno una lunghezza totale pari a 9, il terzo invece ha una lunghezza pari a 8 ed è quindi il percorso ottimale per la corsetta mattutina di William: adesso William sa quali sono le case coinvolte nel percorso più breve, e tra quelle potrà cercare la nuova casa in cui andare ad abitare.

Dati di input

Il file `input.txt` contiene $M + 1$ righe di testo. Sulla prima sono presenti due interi separati da spazio: N e M, rispettivamente il numero di case ed il numero di tratti di strada presenti nella città. Dalla riga 2 fino alla $M + 1$ troviamo la descrizione degli M tratti di strada. Ciascuna di queste righe contiene tre interi separati da spazio: u, v e w, dove u e v sono due case (quindi sono degli indici compresi tra 1 ed N) e w è la lunghezza del tratto di strada che le collega.

Dati di output

Il file `output.txt` contiene un singolo intero: la lunghezza del *ciclo semplice* più corto presente nella città in input.

Assunzioni

- $3 \leq N \leq 1000$.
- $3 \leq M \leq 10\,000$.
- $0 < w \leq 10\,000$, dove w è la lunghezza di un tratto di strada.
- È garantito che nella città esiste sempre almeno un ciclo semplice.
- Nel 40% dei casi di prova tutte le strade hanno lunghezza unitaria.
- È garantito che una coppia di case adiacenti è collegata da *un solo* tratto di strada.
- Una strada non collega mai una casa a se stessa.

Esempi di input/output

input.txt	output.txt
6 10 1 2 1 3 2 2 5 2 6 4 5 2 1 4 2 3 5 1 3 4 7 5 1 5 2 6 4 3 6 3	8

Note

- Per chi usa Pascal: è richiesto che si utilizzi sempre il tipo di dato `longint` al posto di `integer`.
- Un programma che stampa lo stesso output indipendentemente dal file di input non totalizza alcun punteggio.

Soluzione

Consideriamo un arco (u, v) e cerchiamo il più breve ciclo di cui esso fa parte. I cicli semplici contenenti l'arco (u, v) nel grafo della città corrispondono evidentemente ai cammini semplici che congiungono il nodo u al nodo v in un grafo a cui è stato eliminato l'arco (u, v).

È facile trovare la lunghezza del cammino minimo tra u e v nel grafo "ridotto" utilizzando l'algoritmo di Dijkstra[2]. La lunghezza del ciclo è quindi la distanza trovata da Dijkstra, sommata alla lunghezza dell'arco (u, v).

A questo punto risolvere il problema è semplice: per ogni arco troviamo il ciclo più corto che lo contiene e la risposta sarà il minimo delle lunghezze di questi cicli.

La complessità computazionale di questo algoritmo è $O(M(M + N \log N))$, dato che esegue M volte l'algoritmo di Dijkstra che ha complessità computazionale $O(M + N \log N)$.

Approfondimento: è possibile modificare la soluzione precedente, basandosi sempre sull'algoritmo di Dijkstra, e ottenere una soluzione $O(NM + N^2 \log N)$. Si ottiene comunque un notevole miglioramento in prestazioni (per quanto non in complessità computazionale) facendo in modo che l'algoritmo di Dijkstra non esplori cammini più lunghi del ciclo minimo già trovato.

Esempio di codice C++11

```
1  #include <vector>
2  #include <iostream>
3  #include <limits>
4  #include <queue>
5
6  const unsigned INFINITO = std::numeric_limits<unsigned>::max();
7  typedef unsigned vertice_t;
8
9  struct arco_t {
10     vertice_t coda, testa; // I due estremi collegati
11     unsigned peso;         // Il peso dell'arco
12 };
13
14 std::vector<arco_t> archi;                // Lista degli archi
15 std::vector<std::vector<arco_t>> vicini;  // Liste di adiacenza
16 unsigned N, M;
17
18 struct info_t {
19     vertice_t ultimo;  // Il nodo finale del cammino
20     unsigned peso;     // Il peso (cumulativo) del cammino
21
22     bool operator< (const info_t& o) const {
23         return peso > o.peso;
24     }
25 };
26
27 unsigned percorso_minimo(vertice_t partenza, vertice_t arrivo) {
28     std::vector<unsigned> distanza(N, INFINITO);
29     std::priority_queue<info_t> coda;
30     coda.push({partenza, 0});
31
32     while (!coda.empty()) {
33         // Cerca nella coda il cammino che conviene "continuare"
34         vertice_t u = coda.top().ultimo;
35         unsigned w = coda.top().peso;
36         coda.pop();
37
38         if (distanza[u] == INFINITO) {
39             // Non ho mai visto il nodo u
40             distanza[u] = w;
```

[2] http://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra

```
41              // Visita i vicini
42              for (const arco_t& arco: vicini[u]) {
43                  if (arco.coda == partenza && arco.testa == arrivo) {
44                      // Salta l'arco tolto
45                      continue;
46                  }
47                  coda.push({arco.testa, w + arco.peso});
48              }
49          }
50      }
51
52      return distanza[arrivo];
53  }
54
55  int main() {
56      // Input e output da/su file
57      freopen("input.txt", "r", stdin);
58      freopen("output.txt", "w", stdout);
59
60      std::cin >> N >> M;
61      vicini.resize(N);
62
63      for (int i = 0; i < M; ++i) {
64          vertice_t u, v;
65          unsigned w;
66          std::cin >> u >> v >> w;
67
68          // Per comodità riportiamo i nomi dei vertici ad essere 0-based
69          --u;
70          --v;
71
72          // Popola le liste di adiacenza
73          vicini[u].push_back({u, v, w}); // arco u -> v di peso w
74          vicini[v].push_back({v, u, w}); // arco u <- v di peso w
75
76          // Popola la lista degli archi
77          archi.push_back({u, v, w});
78      }
79
80      // Prova per ogni arco
81      unsigned risposta = INFINITO;
82      for (const arco_t& arco: archi) {
83          // Calcola la distanza tra u e v togliendo l'arco tra essi.
84          unsigned pm = percorso_minimo(arco.coda, arco.testa);
85
86          // Se da u ho effettivamente raggiunto v
87          if (pm != INFINITO)
88              risposta = std::min(risposta, arco.peso + pm);
89      }
90
91      std::cout << risposta << std::endl;
92  }
```