ABC 2017

Testi e soluzioni ufficiali dei problemi

Pacchetti corrotti (`checksum`)

Limite di tempo:	2.5 secondi
Limite di memoria:	512 MiB

Gli studenti di un istituto dalle infrastrutture obsolete sono alle prese con un alto tasso di pacchetti corrotti nella rete scolastica (tutti hanno bisogno di scambiarsi file di vitale importanza, specialmente durante le verifiche!). Per provare a mitigare il problema, hanno ideato un algoritmo per garantire l'integrità dei segmenti TCP scambiati su tale rete.

Ogni segmento, come è noto, viene suddiviso in P pacchetti. L'algoritmo prevede di accodare a ciascun pacchetto un numero intero (tra 1 e M) chiamato *checksum*. Per ogni pacchetto, l'algoritmo sceglie un valore di checksum in modo che tale numero sia coprimo rispetto a tutti i checksum finora assegnati. Lato ricevente per verificare l'integrità del segmento è necessario controllare che tutti i P valori di checksum ricevuti rispettino la regola sopra esposta.

> ☞ Due interi si dicono *coprimi* o *relativamente primi* se il loro massimo comune divisore è 1.

Aiuta gli studenti implementando l'algoritmo di controllo lato ricevente! Per ogni pacchetto, rispondi 0 se il checksum rispetta la regola prevista. Altrimenti rispondi con i checksum di un pacchetto *valido* precedente che viola la regola (e quindi indica la corruzione del pacchetto!).

Implementazione

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

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

Dovrai implementare le seguenti funzioni:

- **Funzione `inizializza`**

C/C++	`void inizializza(int P, int M);`

- **Funzione `controlla`**

C/C++	`int controlla(int C);`

  - L'intero P rappresenta il numero di pacchetti in cui è stato suddiviso il segmento.
  - L'intero M rappresenta il massimo valore che può assumere il checksum.
  - L'intero C rappresenta il checksum di un pacchetto, lato ricevente.
  - La funzione dovrà restituire 0, se il checksum è valido, oppure il checksum di un altro pacchetto valido precedente che lo rende invalido.

Il grader chiamerà una sola volta la funzione `inizializza`, quindi chiamerà P volte la funzione `controlla` stampandone il valore restituito sul file di output.

Grader di prova

Allegata 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 da `stdin`, chiama le funzioni che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da 2 righe, contenenti:

- Riga 1: gli interi P e M, separati da uno spazio.
- Riga 2: P interi `C[`i`]` per $i = 0, \ldots, P-1$.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: i valori calcolati dalla funzione `controlla`.

Assunzioni

- $1 \leq P \leq 300\,000$.
- $P \leq M \leq 4\,000\,000$.
- $1 \leq$ `C[`i`]` $\leq M$ per ogni $i = 0, \ldots, P-1$.
- Quando il checksum viene corrotto, esso potrebbe non essere più coprimo con diversi altri checksum. È sufficiente indicarne uno qualsiasi.
- I valori di checksum sono tutti diversi.

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 [35 punti]**: $P \leq 2\,000$ e $M \leq 10\,000$.
- **Subtask 3 [20 punti]**: $P \leq 10\,000$ e $M \leq 80\,000$.
- **Subtask 4 [25 punti]**: $M \leq 500\,000$.
- **Subtask 5 [20 punti]**: Nessuna limitazione specifica.

Esempi di input/output

stdin	stdout
6 10 10 7 4 9 5 6	0 0 10 0 10 9
4 10 3 2 6 4	0 0 3 2

Spiegazione

Nel **primo caso di esempio**:

- Il checksum del primo pacchetto, 10, è valido.
- Il checksum del secondo pacchetto, 7, è valido.

- Il checksum del terzo pacchetto, 4, non è valido perché 4 non è coprimo con 10.
- Il checksum del quarto pacchetto, 9, è valido.
- Il checksum del quinto pacchetto, 5, non è valido perché 5 non è coprimo con 10.
- Il checksum del sesto pacchetto, 6, non è valido perché 6 non è coprimo né con 10 né con 9. È ammesso produrre una qualsiasi delle soluzioni.

Soluzione

■ Soluzione naïve

Possiamo, per iniziare ad affrontare il problema, implementare esattamente quanto viene indicato nel testo. In particolare, scegliamo di memorizzare in un vettore `ricevuti` tutti i checksum validi che man mano arrivano.

Per ogni nuovo pacchetto, caratterizzato da checksum c, iteriamo su tutti gli elementi presenti nel vettore e, per ciascuno, controlliamo che il valore sia coprimo con c.

Per verificare se due interi a, b sono primi tra loro possiamo utilizzare l'algoritmo di Euclide[1].

APPROFONDIMENTO

La complessità computazionale di questa soluzione è strettamente legata a *quante volte* viene eseguita la funzione che calcola il massimo comune divisore tra due numeri. Chiamiamo tale funzione `gcd()`.
Seguendo la prassi dell'analisi della complessità computazionale considereremo il caso peggiore, che si verifica quando tutti i checksum sono numeri primi. In questa particolare situazione, tutti i checksum saranno validi. Per semplicità supponiamo $P = M$ (aggiungendo quindi anche i non primi): per ciascun checksum c dobbiamo potenzialmente controllare tutti i primi fino a c. Questa quantità viene spesso formalizzata in matematica con la funzione $\pi(x)$ che conta il numero di primi $\leq x$.
Il numero n di chiamate a `gcd()` sarà esattamente

$$n = \sum_{k=2}^{P} \pi(k)$$

Grazie al teorema dei numeri primi, che afferma che il numero di primi che non eccedono x è asintotico a $x/\log x$, possiamo scrivere

$$n \sim \sum_{k=2}^{P} \frac{k}{\log k}$$

Osservando tale quantità possiamo notare abbastanza facilmente che è limitata superiormente da $\mathcal{O}(P^2/\log P)$. Tramite argomenti matematici non del tutto triviali si può dimostrare che tale limite superiore è anche stretto[2].

■ Soluzione $\mathcal{O}(P \cdot M)$

L'osservazione cruciale per risolvere il problema è che, per ogni possibile checksum c, vi è al massimo un solo altro checksum c' all'interno del vettore `ricevuti` tale che c divide c' (o viceversa). Infatti, un eventuale altro checksum c'' sarebbe stato scartato a priori, poiché non coprimo con c'.

Possiamo quindi memorizzare un vettore `int confligge`$[1 \dots M]$ in cui l'i-esima posizione indica il valore di un checksum valido c tale che i sia un divisore di c. Se non è ancora arrivato alcun pacchetto il cui checksum abbia i tra i suoi divisori, mettiamo il valore fittizio 0.

[1] http://it.wikipedia.org/wiki/Algoritmo_di_Euclide
[2] https://math.stackexchange.com/questions/2261881

A questo punto, per ogni nuovo pacchetto arrivato con checksum c, iteriamo sui *soli divisori* di c controllando che in tali posizioni il vettore `confligge` abbia solo zeri. In caso contrario, riportiamo come conflitto il valore non nullo trovato all'interno del vettore.

Nel caso in cui si sia determinato che nuovo checksum c è valido, è fondamentale aggiornare il vettore inserendo c in tutte le posizioni costituite dai divisori di c.

Avendo P pacchetti e (potenzialmente) M posizioni del vettore da controllare ed eventualmente aggiornare, la complessità computazionale risulta essere $\mathcal{O}(P \cdot M)$.

■ Soluzione $\mathcal{O}(P \cdot \sqrt{M})$

È possibile apportare una facile miglioria alla soluzione proposta sopra, accorgendosi che per controllare i divisori di c (escluso c stesso) possiamo fermarci a $\sqrt{c}$.

Tale proprietà si dimostra facilmente per assurdo. Supponiamo c composto, allora possiamo scrivere $c = a \cdot b$ con $a, b > 1$. Se sia a che b fossero strettamente maggiori di $\sqrt{c}$, allora si avrebbe $c > \sqrt{c} \cdot \sqrt{c}$, ovvero $c > c$ che è palesemente assurdo.

Ogni volta che incontriamo un divisore $d < \sqrt{c}$, sappiamo che anche c/d è un divisore da controllare ed eventualmente aggiornare con l'indicazione del checksum in conflitto.

■ Una soluzione ancora più efficiente

Per migliorare ulteriormente la nostra soluzione abbiamo bisogno di un'altra osservazione chiave: possiamo memorizzare le informazioni sui conflitti per i soli divisori primi. Infatti, per il teorema fondamentale dell'aritmetica, ogni intero > 1 può essere espresso come prodotto di numeri primi.

Determiniamo innanzitutto i numeri primi fino a M. Per farlo efficientemente eseguiamo una sola volta il crivello di Eratostene in $\mathcal{O}(M \log \log M)$, ottenendo come risultato un vettore contenente i soli numeri primi fino a M.

Adottando un approccio che ricorda lo stile "divide et impera" siamo ora in grado di calcolare rapidamente i fattori primi di un numero. Notiamo che ogni intero n può essere espresso come $n = p \cdot n'$ dove p è un numero primo: ci siamo quindi ridotti a trovare i fattori primi di n' (che è sicuramente minore di n).

Nel caso peggiore, quando un checksum c è un numero primo, siamo costretti a controllare i numeri primi fino a $\sqrt{c}$. Tale quantità è $\mathcal{O}(\pi(c))$ (dove $\pi(x)$ è la notazione matematica standard per indicare la funzione che conta il numero di primi $\leq x$). Possiamo approssimare la complessità a $\mathcal{O}(\sqrt{c}/\log \sqrt{c})$.

Una volta ottenuta la fattorizzazione di c la soluzione al problema è semplice e ricalca quanto visto nelle sezioni precedenti, avendo l'accortezza di controllare (ed eventualmente aggiornare) le soli posizioni di `confligge` corrisponenti ai fattori primi del checksum ricevuto.

Per determinare la complessità di quest'ultima parte occorre notare che essa è determinata dal numero di divisori primi distinti di un checksum. La funzione matematica $\omega(x)$ è definita esattamente in questo modo è può essere approssimata con $\mathcal{O}(\log x / \log \log x)$ o più semplicemente $\mathcal{O}(\log x)$.

Ricapitolando, la soluzione consta dei seguenti passi:

- Precalcoliamo i primi fino a M in $\mathcal{O}(M \log \log M)$ con il crivello di Eratostene
- Per ciascuno dei P checksum c:
  - Calcoliamo i fattori primi di c in $\mathcal{O}(\sqrt{c}/\log \sqrt{c})$
  - Eseguiamo le operazioni sull'array `confligge` in corrispondenza dei $\log c$ fattori primi.

Esempio di codice C++11

```cpp
#include <cmath>
#include <vector>

const int MAXM = 4000000;

std::vector<int> confligge(MAXM + 1);
std::vector<int> primi;
std::vector<bool> is_primo(MAXM + 1, true);

void inizializza(int P, int M) {
    // Trovo i numeri primi tramite il crivello di Eratostene.
    for (int i = 2; i <= M; i++) {
        if (is_primo[i]) {
            primi.push_back(i);
            if (i < sqrt(M))
                for (int j = i * i; j <= M; j += i)
                    is_primo[j] = false;
        }
    }
}

int controlla(int c) {
    // Calcolo efficentemente i fattori primi di c
    std::vector<int> fattori;
    int n = c, p = primi[0], idx = 0;
    while (n != 1 && (p * p <= n)) {
        while (n % p == 0) {
            n /= p;
            fattori.push_back(p);
        }
        p = primi[++idx];
    }
    if (n != 1)
        fattori.push_back(n);

    // Controllo che non ci siano conflitti.
    for (int p : fattori)
        if (p != 1 && confligge[p] != 0)
            return confligge[p];

    // Se invece va tutto bene segno i nuovi conflitti e ritorno 0.
    for (int p : fattori)
        confligge[p] = c;
    return 0;
}
```

Citazioni nei paper (paper)

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

Il mondo degli articoli scientifici, chiamati in gergo *paper*, si basa sul concetto di *citazione*. Infatti, quando si scrive un paper è fondamentale citare altri paper che contengono informazioni correlate o dai quali si è tratto parte del lavoro. Questo è evidentemente necessario allo scopo di fornire spunti di lettura e approfondimento potenzialmente interessanti per un lettore.

Luca si sta preparando a scrivere il suo primo paper. Facendo ricerche sull'argomento scelto ha trovato N paper particolarmente interessanti, indicati con interi da 1 a N, e ha catalogato con accuratezza tutte le M citazioni presenti. Ciascuna di queste citazioni è descritta con una coppia di interi `A[`i`]` e `B[`i`]`, a indicare che il paper `A[`i`]` contiene una citazione al paper `B[`i`]`.

Luca vorrebbe inserire nel suo nuovo articolo una citazione a tutti i paper che ha già trovato, ma un suo professore gli ha consigliato che è bene tenere questa lista corta per non tediare i lettori. Il professore gli ha quindi suggerito di seguire questa regola di buon senso: scegliere un unico paper da citare nel suo articolo, in modo che "a cascata" (ovvero, seguendo mano a mano le varie citazioni) un lettore incuriosito possa andarsi a leggere *tutti* gli altri paper.

Quanti paper può scegliere Luca come unica citazione da inserire nel suo articolo?

Implementazione

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

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

Dovrai implementare la seguente funzione:

Funzione `paper`

C/C++	`int paper(int N, int M, int A[], int B[]);`

- L'intero N rappresenta il numero di paper scientifici.
- L'intero M rappresenta il numero totale delle citazioni.
- Gli array `A` e `B`, indicizzati da 0 a $M-1$, contengono alla posizione i la seguente informazione: il paper `A[`i`]` cita il paper `B[`i`]`.
- La funzione dovrà restituire il numero di paper da cui si può iniziare la lettura in modo che, seguendo le citazioni, si arrivi a leggerli tutti.

Il grader chiamerà la funzione `paper` e ne stamperà il valore restituito sul file di output.

Grader di prova

Allegata 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 da `stdin`, chiama le funzioni che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da $M+1$ righe, contenenti:

- Riga 1: gli interi N e M, separati da uno spazio.
- Righe $2, \ldots, M+1$: i due interi `A[`i`]`, `B[`i`]` per $i = 0, \ldots, M-1$.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `paper`.

Assunzioni

- $1 \leq N \leq 75\,000$.
- $1 \leq M \leq 500\,000$.
- $1 \leq$ `A[`i`]`, `B[`i`]` $\leq N$ per ogni $i = 0, \ldots, M-1$.
- È possibile che la risposta al problema sia 0 (come nel secondo caso di esempio).
- Un paper non contiene mai una citazione a se stesso.
  Formalmente: `A[`i`]` $\neq$ `B[`i`]` per ogni $i = 0, \ldots, M-1$.
- Poiché un paper può essere aggiornato dopo la pubblicazione, sono lecite situazioni in cui p cita q (anche indirettamente) e q cita p (anche indirettamente).

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 [50 punti]**: $N \leq 500$ e $M \leq 4000$.
- **Subtask 3 [50 punti]**: Nessuna limitazione specifica.

Esempi di input/output

stdin	stdout
3 3 1 2 2 3 2 1	2
3 2 1 2 3 2	0

Spiegazione

Nel **primo caso di esempio** Luca può scegliere tra due paper (1 e 2) per la sua unica citazione. Infatti:

- Iniziando a leggere il paper 1 si trova una citazione al paper 2 e da quest'ultimo una citazione al paper 3.

- Iniziando a leggere il paper 2 si trovano le due citazioni ai paper 1 e 3.

Nel **secondo caso di esempio** Luca non ha modo di scegliere alcun paper tale che, seguendo le citazioni, si arrivi a leggere tutti i 3 paper.

Soluzione

Costruiamo anzitutto un grafo $G = (V, E)$ dove V è l'insieme dei paper, numerati da 1 a N, ed E è l'insieme delle M citazioni. Osserviamo che il grafo è orientato.

■ Una semplice soluzione

Per contare il numero di paper da cui si può partire, possiamo iterare su di essi e considerare ciascuno come nodo di partenza di una visita sul grafo. Possiamo utilizzare indifferentemente una visita in ampiezza (BFS) oppure una visita in profondità (DFS).

Al termine di ciascuna visita, controlliamo se tutti i nodi sono stati visitati. In caso affermativo, abbiamo determinato che tale nodo di partenza è un paper valido per Luca.

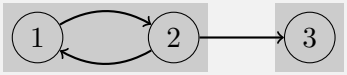
Ciascuna delle N visite richiede $\mathcal{O}(N + M)$, quindi la soluzione ha complessità $\mathcal{O}(N^2 + NM)$.

■ Una (non banale) soluzione lineare

Non è semplice immaginare una soluzione con un tempo migliore di quella già presentata. Per arrivarci gradualmente, iniziamo introducendo il concetto di *componente fortemente connessa (SCC)*.

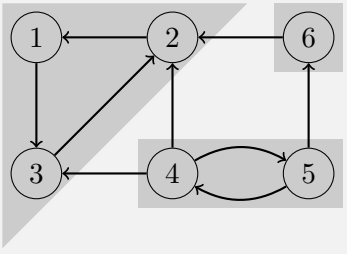
> ☞ Una *componente fortemente connessa* di un grafo diretto è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici $u, v \in C$ si ha $u \rightsquigarrow v$ e $v \rightsquigarrow u$ (cioè u e v sono raggiungibili l'uno dall'altro).

Nelle seguenti figure sono mostrate, racchiuse in aree a sfondo grigio, le SCC dei due esempi forniti nel testo e di un ulteriore esempio più articolato.



Primo esempio

Secondo esempio



Un esempio più articolato

Esistono algoritmi per determinare le componenti fortemente connesse di un grafo diretto in tempo lineare. Uno dei più semplici è l'algoritmo di Kosaraju, che fa uso di due DFS (una sul grafo stesso e una sul grafo trasposto). L'approfondimento completo di tale algoritmo è lasciato al lettore.

Per arrivare alla soluzione passiamo attraverso due lemmi.

Lemma. *Se da un nodo x, appartenente alla componente C, è possibile raggiungere qualsiasi nodo $v \in V$, allora anche da $y \in C$ è possibile raggiungere qualsiasi altro nodo.*

Dimostrazione. Segue immediatamente dalla definizione di SCC: poiché x e y appartengono entrambe a C, esiste sicuramente $y \rightsquigarrow x$. Per ipotesi comunque scelto v esiste anche $x \rightsquigarrow v$, quindi esisterà $y \rightsquigarrow v$. □

Definiamo `indeg` come il numero di archi "in ingresso" per ogni SCC. Formalmente:

$$\texttt{indeg}(C) \triangleq |\{(u, v) \in E \mid u \notin C \wedge v \in C\}|$$

Lemma. *Se da un qualsiasi nodo di C è possibile raggiungere tutti gli altri nodi del grafo, allora $\texttt{indeg}(C) = 0$.*

Dimostrazione. Per assurdo. Supponiamo che esista un nodo x, appartentente alla componente C, da cui è possibile raggiungere tutti gli altri nodi. Supponiamo inoltre $\texttt{indeg}(C) > 0$. Allora deve esistere almeno un arco (y, x) con $y \notin C$. Tuttavia abbiamo che per ipotesi da x si può raggiungere qualsiasi nodo, incluso y. Dunque, poiché esistono sia $y \rightsquigarrow x$ che $x \rightsquigarrow y$, per definizione di componente fortemente connessa i due nodi devono appartenere entrambi a C. In conclusione otteniamo $y \notin C$ e $y \in C$: assurdo. □

Grazie a questi due lemmi possiamo ora arrivare alla soluzione del problema.

Teorema. *Esiste una soluzione solo se esiste un'unica componente fortemente connessa C in G avente `indeg` nullo. In tal caso, la risposta al problema è $|C|$.*

Dimostrazione.

- Supponiamo che esistano due componenti C_1 e C_2 con $\texttt{indeg} = 0$ e che dai nodi di C_1 si possano raggiungere tutti gli altri nodi del grafo. Questo è evidentemente impossibile, perché per raggiungere i nodi della componente C_2 essa dovrebbe avere almeno un arco in ingresso.
- Viceversa, se esiste solamente una componente fortemente connessa C_1 con `indeg` nullo, tutte le altre $k - 1$ componenti avranno almeno un arco in ingresso. Senza perdita di generalità possiamo immaginare che esista un cammino $C_1 \rightsquigarrow C_2 \rightsquigarrow \ldots \rightsquigarrow C_{k-1} \rightsquigarrow C_k$. Grazie al primo lemma, concludiamo che tutti i nodi di C_1 possono raggiungere tutti gli altri nodi del grafo.

□

Per i dettagli relativi all'implementazione è possibile consultare il codice proposto sotto. La complessità finale dell'algoritmo è determinata da quella necessaria per trovare le componenti fortemente connesse e risulta essere lineare nel numero dei nodi e degli archi.

Esempio di codice C++11

```cpp
#include <algorithm>
#include <stack>
#include <vector>

const int MAXN = 75000;

std::stack<int> s;
std::vector<bool> visitato(MAXN + 1, false);
std::vector<int> scc(MAXN + 1), indeg(MAXN + 1);
std::vector<int> grafo[MAXN + 1], grafoT[MAXN + 1];

void dfs(int nodo) {
    visitato[nodo] = true;
    for (int prossimo : grafo[nodo])
        if (!visitato[prossimo])
            dfs(prossimo);
    s.push(nodo);
}

void dfsT(int nodo, int scc_id) {
    visitato[nodo] = true;
    scc[nodo] = scc_id;
    for (int prossimo : grafoT[nodo])
        if (!visitato[prossimo])
            dfsT(prossimo, scc_id);
}

int paper(int N, int M, int* A, int* B) {
    // Costruisco il grafo normale e quello trasposto.
    for (int i = 0; i < M; i++) {
        grafo[A[i]].push_back(B[i]);
        grafoT[B[i]].push_back(A[i]);
    }
    // Trovo le SCC in due passi.
    // Passo 1: eseguo una DFS sul grafo normale.
    for (int i = 1; i <= N; i++)
        if (!visitato[i])
            dfs(i);
    // Passo 2: sfrutto le informazioni sullo stack e il grafo trasposto
    // per etichettare correttamente le SCC.
    std::fill(visitato.begin(), visitato.end(), false);
    int scc_id = 0;
    while (!s.empty()) {
        int cur = s.top();
        s.pop();
        if (!visitato[cur]) {
            dfsT(cur, scc_id);
            scc_id++;
        }
    }
    // Per ogni componente connessa calcolo indeg, il numero di archi in ingresso.
    for (int i = 1; i <= N; i++)
        for (int prossimo : grafo[i]) {
          if (scc[i] != scc[prossimo])
            indeg[scc[prossimo]]++;
        }
    // Conto il numero di SCC aventi indeg nullo.
    int zero_deg = std::count(indeg.begin(), indeg.begin() + scc_id, 0);

    if (zero_deg > 1) // non c'è soluzione!
        return 0;
    else {
        // Conto il numero di nodi presenti nella SCC con indeg nullo.
        int quanti = 0;
        for (int i = 1; i <= N; i++)
            if (indeg[scc[i]] == 0)
                quanti++;
        return quanti;
    }
}
```

Studio amico (`studioamico`)

Limite di tempo:	0.5 secondi
Limite di memoria:	256 MiB

Presso l'ITIS Paleocapa, da qualche anno, si svolge il progetto "Studio amico" in cui uno studente più grande (ovvero frequentante una classe superiore) mette a disposizione gratuitamente il proprio tempo per aiutare uno studente più piccolo con qualche lacuna.

Quest'anno il Dirigente Scolastico ha incaricato il professor Nevabe, che insegna in una classe seconda e in una classe quinta, di organizzare tale attività per la disciplina di sua competenza: informatica.

Il professor Nevabe ha quindi individuato un sottoinsieme di N studenti di classe quinta da appaiare ad altrettanti N studenti frequentanti la classe seconda. L'esperienza insegna che tale aiuto sarà efficace solo se lo studente più grande ha un voto in informatica *strettamente maggiore* dello studente più piccolo.

Preso dai tanti impegni che si avvicendano nella seconda parte dell'anno scolastico, il professor Nevabe non ha tempo di controllare se può effettivamente associare ad ogni studente del secondo anno uno studente del quinto anno adatto. Aiutalo tu!

Implementazione

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

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

Dovrai implementare la seguente funzione:

Funzione `associabili`

C/C++	`bool associabili(int N, int voti2[], voti5[]);`

- L'intero N rappresenta il numero di studenti di ciascuna classe.
- Gli array `voti2` e `voti5`, indicizzati da 0 a $N-1$, contengono rispettivamente i voti degli studenti della classe seconda e i voti degli studneti della classe quinta.
- La funzione dovrà restituire `true` se è possibile trovare un modo di abbinare ogni studente di seconda a uno studente di quinta con un voto strettamente maggiore, `false` altrimenti.

Il grader chiamerà la funzione `associabili` e ne stamperà il valore restituito sul file di output.

Grader di prova

Allegata 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 da `stdin`, chiama le funzioni che dovete implementare e scrive su `stdout`, secondo il seguente formato.

Il file di input è composto da 3 righe, contenenti:

- Riga 1: l'unico intero N.
- Riga 2: gli N voti degli studenti di seconda.
- Riga 3: gli N voti degli studenti di quinta.

Il file di output è composto da un'unica riga, contenente:

- Riga 1: il valore restituito dalla funzione `associabili`.

Assunzioni

- $1 \leq N \leq 10\,000\,000$.
- I voti degli studenti sono espressi come interi tra 1 e 10: $1 \leq$ `voti2[`i`]`, `voti5[`i`]` ≤ 10 per ogni $i = 0, \ldots, N-1$.
- L'associazione è rigorosamente uno-a-uno: ad uno studente di seconda viene associato uno e un solo studente di quinta (e viceversa).

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 [30 punti]**: $N \leq 10$.
- **Subtask 3 [30 punti]**: $N \leq 100\,000$.
- **Subtask 4 [40 punti]**: Nessuna limitazione specifica.

Esempi di input/output

stdin	stdout
3 4 6 5 7 6 9	1
3 7 4 5 5 4 10	0

Spiegazione

Nel **primo caso di esempio** uno dei possibili abbinamenti realizzabili consiste nell'assegnare al primo studente di seconda (con voto 4) il secondo studente di quinta (con voto 6), al secondo studente di seconda (voto 6) il primo studente di quinta (voto 7) e infine al terzo studente di seconda (voto 5) il terzo studente di seconda (voto 9).

Nel **secondo caso di esempio** non esiste alcun modo di associare gli studenti in modo che a ogni studente di seconda venga assegnato uno studente di quinta con un voto strettamente maggiore del proprio.

Soluzione

Osserviamo che il testo del problema potrebbe essere riformulato nel seguente modo: esiste una funzione *biiettiva* f : `voti2` $\to$ `voti5` che rispetti quanto richiesto?

■ Soluzione bruteforce

Una soluzione intuitiva è quella in cui verifichiamo *tutte* le possibili funzioni, che descrivono di fatto gli abbinamenti tra gli studenti, verificandone la validità. Dopo aver "generato" la funzione è sufficiente controllare se ciascuno studente del secondo anno ha un voto minore dello studente del quinto anno a cui è stato abbinato seguendo f.

Quante sono tali funzioni? Per il primo elemento di `voti2` abbiamo N scelte da `voti5`. Una volta fissato, per il secondo elemento abbiamo $N-1$ scelte (e così via). Si ottiene facilmente che il numero totale di differenti funzioni biiettive è $N \cdot (N-1) \cdot \ldots \cdot 2 \cdot 1 = N!$. Questa soluzione è quindi adatta a risolvere il secondo subtask.

■ Soluzione greedy in $\mathcal{O}(N \log N)$

È abbastanza facile intuire che il problema ammette una soluzione greedy. Omettiamo una dimostrazione puramente formale, ma cerchiamo di convincerci che ciò è vero. Consideriamo gli studenti del secondo anno in ordine di voto e concentriamoci sul primo, avente voto x: a quale studente del quinto anno conviene abbinarlo? Si possono presentare due casi:

- Non c'è alcuno studente del quinto anno avente un voto y tale che $y > x$: in questo caso concludiamo che non è possibile effettuare l'abbinamento richiesto.
- Esiste un insieme S di studenti (con $|S| \geq 1$) del quinto anno aventi voto maggiore di x: scegliamo uno studente avente voto $y = \min S$. Infatti non vi è alcuna convenienza a scegliere uno studente con voto $y' > y$, che potrebbe "servire" più tardi per l'abbinamento con uno studente più bravo del secondo anno.

Analogo ragionamento vale proseguendo con tutti gli altri studenti del secondo anno, fino a che completiamo un abbinamento valido oppure ci accorgiamo che non può esistere.

Per realizzare quanto descritto sopra in modo efficiente, ordiniamo separatamente gli array `voti2` e `voti5` per voto crescente. Per ogni $i = 0 \ldots N-1$, deve valere `voti5[`i`]` > `voti2[`i`]`.

Il tempo di esecuzione richiesto da questo algoritmo è pari al tempo richiesto dall'ordinamento dei due vettori, ovvero $\mathcal{O}(N \log N)$.

■ Soluzione greedy in $\mathcal{O}(N)$

Si può dimostrare che ogni algoritmo di ordinamento basato sui confronti richiede almeno $\Omega(N \log N)$. Tuttavia, in questo particolare caso possiamo sfruttare il fatto che il *range* dei valori da ordinare è molto piccolo rispetto a N (i voti sono infatti compresi tra 1 e 10). Con le stesse considerazioni della soluzione precedente e applicando l'algoritmo di ordinamento Counting-Sort, che non si basa sui confronti, possiamo risolvere il problema in $\mathcal{O}(N)$.

Esempio di codice C++11

```cpp
const int MAXN = 10000000;
const int MAXV = 10;

int b[MAXN], c[MAXV + 1];

void countingsort(int *a, int N) {
    for (int i = 0; i <= MAXV; i++)
        c[i] = 0;
    for (int i = 0; i < N; i++)
        c[a[i]]++;
    for (int i = 1; i <= MAXV; i++)
        c[i] += c[i - 1];
    for (int i = N - 1; i >= 0; i--) {
        int posc = a[i];
        int posb = c[posc] - 1;
        b[posb] = a[i];
        c[posc]--;
    }
    for (int i = 0; i < N; i++)
        a[i] = b[i];
}


bool associabili(int N, int* voti2, int* voti5) {
    // Ordino separatamente i due vettori in tempo lineare.
    countingsort(voti2, N);
    countingsort(voti5, N);

    // In modo greedy controllo che rispettino quanto richiesto.
    for (int i = 0; i < N; i++)
        if (voti2[i] >= voti5[i])
            return false;

    return true;
}
```