Gara di selezione ACM ICPC

Università di Bologna

Cesena, 22 e 27 ottobre 2014

Indice

Prefisso telefonico (`prefisso`)

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

Paperino ha n amici nella città di Paperopoli. Egli conosce il numero di telefono di tutti i sui amici, sotto forma di stringa $(s_1, s_2, \ldots, s_n)$. Ogni stringa ha la stessa lunghezza e contiene solo cifre da 0 a 9. Egli vuole scoprire il prefisso telefonico della città, assumendo che esso sia il più lungo prefisso comune a tutti i numeri dei suoi amici. In altre parole, vuole trovare la più lunga stringa c che è prefisso (sottostringa che parte dall'inizio) di ogni s_i $(1 \leq i \leq n)$. Aiuta Paperino a trovare la lunghezza del prefisso di Paperopoli.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: $n \leq 100$.
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: contiene l'intero n, il numero di amici di Paperino
- Le successive n righe contengono le stringhe $s_1, s_2, \ldots, s_n$, i numeri di telefono dei suoi amici.

Il tuo programma dovrà stampare a video i seguenti dati:

- Un numero intero: la lunghezza del prefisso della città.

Assunzioni

- $2 \leq n \leq 30\,000$.
- Tutti i numeri di telefono sono della stessa lunghezza e comprendono solo cifre da 0 e 9.
- La lunghezza dei numeri di telefono è al più 20.
- Tutti i numeri di telefono sono diversi tra loro.

Esempi di input/output

stdin	stdout
4 12345 12395 12349 12312	3
7 919239144321 919239143321 919239124891 919239144311 919239144621 919239142027 919239146352	7

Spiegazione

- Nel **primo caso d'esempio** il prefisso è `123`, di lunghezza 3.
- Nel **secondo caso d'esempio** il prefisso è `9192391`, di lunghezza 7.

Soluzione

Il problema chiede di trovare il più lungo prefisso comune a tutte le n stringhe date in input (sapendo che ciascuna è lunga al massimo $m = 20$).

■ Una soluzione $O(n^2m)$

Definiamo per comodità una funzione `lcp(i, j)` definita come la lunghezza del prefisso più lungo che è comune all'i-esima e j-esima stringa. È facile implementarla in modo che abbia complessità $O(m)$.

Un'idea intuitiva a questo punto è prendere il minimo tra tutti i valori di `lcp(i, j)`, per ogni coppia (i, j). Le coppie totali sono n^2, ma possiamo considerarne qualcuna in meno osservando che:

- `lcp(i, i)` = m, e il valore m non influisce sul minimo che stiamo cercando. Quindi ci basta considerare le coppie (i, j) dove $i \neq j$, che in totale sono $n(n-1)$.
- `lcp(i, j)` = `lcp(j, i)` quindi, per esempio, una volta che abbiamo considerato la coppia $(2, 3)$ non ha senso considerare anche $(3, 2)$. Perciò, ci basta considerare le coppie (i, j) dove $i < j$, che in totale sono $\frac{n(n-1)}{2}$.

Purtroppo queste "ottimizzazioni" non hanno effetto sull'andamento asintotico del tempo di esecuzione, infatti $\frac{n(n-1)}{2}$ è comunque $O(n^2)$. Quindi, il tempo di esecuzione dell'algoritmo è $O(n^2m)$.

```
risposta = m;
for (int i=1; i<=n; i++)
    for (int j=i+1; j<=n; j++)
        risposta = min(risposta, lcp(i, j));
```

■ Una soluzione $O(m^2n)$

La chiave per risolvere il problema efficientemente è ragionare "al contrario": partiamo facendo finta che la soluzione sia pari a x, verifichiamo quindi che tutte le stringhe date abbiano i primi x caratteri in comune.

```
risposta = m;
for (; x>=0; x--) {
    bool ok = true;
    for (int i=2; i<=n; j++)
        if (lcp(1, i) < x)
            ok = false;
    if (ok)
        break;
}
```

L'algoritmo ha tempo di esecuzione $O(m \cdot n \cdot m) = O(m^2n)$, che già basta per prendere punteggio pieno.

■ Una soluzione $O(mn)$

Non usiamo più la funzione `lcp`. Controlliamo che il primo carattere sia uguale per tutte le stringhe. Se è così, la soluzione è almeno pari ad 1. Controlliamo che il secondo carattere sia uguale per tutte le stringhe. Se è così, la soluzione è almeno 2. Ripetiamo m volte, fintantoché l'i-esimo carattere è uguale per tutti. Appena troviamo un i-esimo carattere *non* uguale per tutti, la soluzione sarà $i - 1$.

Matrice prima (`matrice`)

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

Peter ha una matrice $n \times m$ composta da numeri interi. In un turno, Peter può applicare una singola trasformazione alla matrice: scelto un elemento della matrice, lo incrementa di 1. Ogni elemento della matrice può essere scelto (e quindi incrementato) un numero arbitrario di volte. Una matrice è definita *prima* se e solo se vale almeno una delle seguenti condizioni:

- La matrice ha una riga composta da soli numeri primi.
- La matrice ha una colonna composta da soli numeri primi.

Il suo compito è quello di contare il numero minimo di turni necessari per rendere la matrice *prima*. Per farlo, chiede il tuo aiuto.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: ogni elemento della matrice è minore o uguale a 1000.
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: contiene gli interi n e m, il numero di righe e di colonne rispettivamente.
- n righe seguono: l'i-esima riga contiene gli m elementi che compongono l'i-esima riga della matrice.

Il tuo programma dovrà stampare a video i seguenti dati:

- Un singolo intero: il numero minimo di turni necessari per rendere *prima* la matrice.

Assunzioni

- $1 \leq n, m \leq 500$.
- Ogni elemento della matrice è minore o uguale a 100 000.

Esempi di input/output

stdin	stdout
3 3 1 2 3 4 5 6 7 8 9	1
2 3 4 8 8 14 2 14	3

Spiegazione

- Nel **primo caso d'esempio** posso incrementare una volta il valore 1 in posizione $(1, 1)$, così che la riga 1 sarà composta da soli numeri primi.
- Nel **secondo caso d'esempio** posso incrementare tre volte il valore 8 in posizione $(1, 2)$, così che la colonna 2 sarà composta da soli numeri primi.

Soluzione

Il problema chiede il minor numero di incrementi da eseguire su singoli elementi della matrice al fine di ottenere una *matrice prima*, ovvero al fine di ottenere almeno una riga o una colonna composte da soli numeri primi.

La prima osservazione da fare per risolvere il problema è che, dato che possiamo solo *incrementare* gli elementi, il numero di incrementi da fare su ciascun elemento si può calcolare facilmente. Cioè, se per assurdo decidessimo di incrementare *tutti* gli elementi in modo da farli diventare *tutti* numeri primi, trovare il minimo numero di incrementi è intuitivo.

Calcoliamo innanzitutto i numeri primi nell'intervallo che ci interessa, per esempio $[1, 100\,000]$. A questo punto, per ogni elemento della matrice, dobbiamo cercare il numero primo *più vicino* (tra quelli maggiori). Per esempio, se abbiamo il numero 9, il numero primo più vicino sarà 11. Volendo, possiamo usare una ricerca binaria.

Il problema tuttavia non richiede di rendere *tutti* i numeri primi, ma solo una riga o una colonna. Calcoliamo il numero di incrementi per tutti i numeri come descritto, costruendo una seconda matrice che, in posizione i, j, ci dice l'incremento che dobbiamo fare all'elemento i, j per renderlo primo. Infine, cerchiamo la riga/colonna di somma minima (che si fa facilmente in $O(n^2)$).

L'ultimo tassello per completare la soluzione è capire come fare a calcolare i numeri primi in modo efficiente. Un modo è utilizzare il Crivello di Eratostene, che funziona velocemente anche per intervalli dell'ordine di $[1, 10\,000\,000]$. In questo problema tuttavia non era obbligatorio usare quel metodo; era sufficiente infatti cercare i primi nell'intervallo di interesse con un semplice test di primalità.

Per controllare se x è un numero primo basta verificare, per tutti i numeri interi d compresi tra 2 e $x - 1$, che il resto della divisione di x per d non sia zero. Questo metodo impiega tempo $O(x)$ e va bene per intervalli dell'ordine di $[1, 10\,000]$, ma possiamo ottimizzarlo fermando il controllo quando d supera la radice quadrata di x, impiegando così tempo $O(\sqrt{x})$[1] e coprendo in tempo ragionevole intervalli dell'ordine di $[1, 250\,000]$.

Riassumendo, l'algoritmo:

1. Calcola (in un modo efficiente a scelta) i numeri primi nell'intervallo richiesto e li salva in un array. Tempo $O(m\sqrt{m})$, dove m indica l'intervallo $[1, m]$, se si usa il test di primalità. Tempo $O(m \log \log m)$[2] se si usa il Crivello di Eratostene.
2. Per ogni elemento della matrice, fa una ricerca binaria nell'array dei numeri primi e trova il "vicino". Tempo $O(\log m)$, ma forse anche meno (i numeri primi sotto m sono $o(m)$).
3. Popola una matrice che in posizione i, j ha la differenza tra il valore trovato al punto 2 ed il valore iniziale. Tempo $O(n^2)$.
4. Calcola la somma di ciascuna riga e di ciascuna colonna. Tempo $O(n^2)$.
5. Restituisce il minimo tra i risultati trovati al punto 4.

L'algoritmo quindi ha un tempo di esecuzione $O(m\sqrt{m} + n^2)$ col test di primalità, $O(m \log \log m + n^2)$ con il Crivello di Eratostene.

[1] Se x non è primo, allora $x = ab$ per un'opportuna scelta di a e b interi. Se per assurdo sia a che b fossero maggiori di $\sqrt{x}$ avremmo che $ab > x$, in contraddizione con l'ipotesi. Quindi almeno un divisore è sotto la radice, e a noi uno basta.

[2] Con $\log \log m$ si intende $\log(\log(m))$.

Hello World! (`helloworld`)

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

Franco vuole imparare a programmare, e per questo ha scritto il suo primo programma che stampa `helloworld`. Dopo averlo scritto, in preda all'entusiasmo, ha mandato in esecuzione molte volte il suo programma (mandando ogni volta il risultato su un file); sfortunatamente però, suo padre passa lì vicino e inciampa con il filo dell'alimentazione, spegnendo il PC.

Dopo aver riacceso il PC, Franco nota che il file di testo si è danneggiato e adesso contiene una lunga stringa di caratteri mischiati. Invece di farsi prendere dal panico, Franco pensa ad un nuovo problema da risolvere: calcolare il numero di sottostringhe che cominciano per `hello` e finiscono per `world`.

Franco, essendo ancora alle prime armi, non riesce a risolvere il problema. Aiutalo nel suo intento!

✍ **Nota:** Si ricorda che una sottostringa è una sequenza continua di caratteri che fanno parte della stringa originaria, e che due sottostringhe sono considerate distinte se iniziano o finiscono in punti diversi della stringa originaria.

☞ **Nota bene:** Un intero a 32 bit potrebbe non bastare per contenere la risposta. In C/C++ esiste il tipo di dato `long long int` che dispone di 64 bit. Per leggerlo/scriverlo con printf/scanf si deve usare il parametro `%lld`.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: la stringa è lunga al massimo 1000 caratteri.
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: una singola stringa senza spazi composta da lettere dell'alfabeto minuscole.

Il tuo programma dovrà stampare a video i seguenti dati:

- un singolo numero: il numero di sottostringhe che iniziano per `hello` e terminano con `world`.

Assunzioni

- La stringa di input è lunga al massimo 1 000 000 di caratteri.

Esempi di input/output

stdin	stdout
helloworldishelloworld	3
jhelloworldandworld	2
worldhello	0

Soluzione

Il problema chiede di contare le sottostringhe che cominciano per `hello` e finiscono per `world`

■ Una soluzione $O(n^2)$

Un'idea "straight-forward", che produce un algoritmo quadratico, è la seguente:

1. Consideriamo tutte le coppie di indici (i, j) tali che $1 \leq i, j \leq n$ e che $i < j$. Come sappiamo queste sono $\frac{n(n-1)}{2}$, quindi $O(n^2)$.
2. Per ciascuna coppia, verifichiamo se la sottostringa che inizia in posizione i e finisce in posizione j è nella forma `hello*world`, dove `*` può essere qualsiasi cosa.

Per fare la verifica al passo 2, ci basta controllare che:

- i sia sufficientemente distante da j, ovvero $j - i \geq 9$. Tempo costante (1 confronto).
- La "sottosottostringa" che comincia in i e finisce in $i + 5$ sia uguale ad `hello`. Tempo costante (5 confronti).
- La "sottosottostringa" che comincia in $j - 5$ e finisce in j sia uguale ad `world`. Tempo costante (5 confronti).

In definitiva, eseguiamo delle operazioni che richiedono tempo costante, però lo facciamo $O(n^2)$ volte. L'algoritmo è quindi quadratico.

■ Una soluzione $O(n)$

Immaginiamo di scorrere la stringa. Supponiamo di incontrare ad un certo punto un `world`. Ora ci chiediamo: qual è il numero di sottostringhe che iniziano per `hello` e che finiscono *esattamente* con il `world` che abbiamo appena incontrato? È facile convincersi che la risposta a questa domanda è: il numero di `hello` che abbiamo visto fin ora scorrendo la stringa.

Questo ci porta al seguente algoritmo lineare:

1. Inizializziamo `long long int risposta=0;`
2. Inizializziamo `int conta_hello=0;`
3. Scorriamo la stringa.
4. Ci troviamo in posizione i. Se qui comincia un `hello`, allora incrementiamo di uno `conta_hello`.
5. Se, invece, in posizione i comincia un `world`, allora incrementiamo `risposta` di `conta_hello`.

Alla fine, `risposta` conterrà il valore cercato.

Suffissi (`suffissi`)

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

Peter ha un array a di n numeri interi, $a_1, a_2, ..., a_n$. Non avendo nulla da fare, prende un foglio di carta e scrive m valori interi, $b_1, b_2, \ldots, b_m$ $(1 \leq b_i \leq n)$. Per ogni numero b_i vuole sapere quanti numeri distinti ci sono nel vettore a dalla posizione b_i in poi. Ovvero vuole sapere, dato b_i, quanti sono i valori distinti che si trovano nel sottoarray $a_{b_i}, a_{b_i+1}, a_{b_i+2}, \ldots, a_n$. Peter non riesce a risolvere il problema e chiede il tuo aiuto.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**25 punti**]: $n, m \leq 100$, $1 \leq a_i \leq 100$.
- **Subtask 3** [**25 punti**]: $n, m \leq 500$, $1 \leq a_i \leq 100\,000$.
- **Subtask 4** [**30 punti**]: $1 \leq a_i \leq 100\,000$.
- **Subtask 5** [**20 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: contiene gli interi n e m, di elementi nei vettori a e b rispettivamente.
- Riga 2: contiene n interi, di cui l'i-esimo rappresenta i-esimo elemento del vettore a.
- m righe seguono: l'i-esima riga contiene l'i-esimo elemento del vettore b.

Il tuo programma dovrà stampare a video i seguenti dati:

- m righe : l'i-esima riga contiene la risposta al problema per l'i-esimo valore di b.

Assunzioni

- $1 \leq n \leq 100\,000$.
- $1 \leq m \leq 100\,000$.
- $1 \leq a_i \leq 1\,000\,000\,000$.

Esempi di input/output

stdin	stdout
10 11 1 6 3 4 1 6 3 4 512 1024 1 2 3 4 5 6 7 8 9 10 5	6 6 6 6 6 5 4 3 2 1 6
4 4 1 2 1 4 1 2 3 4	3 3 2 1

Spiegazione

Nel **secondo caso d'esempio**:

- Per $b_i = 1$ si vuole stampare il numero di numeri distinti tra 1, 2, 1 e 4, cioè 3.
- Per $b_i = 2$ si vuole stampare il numero di numeri distinti tra 2, 1 e 4, cioè 3.
- Per $b_i = 3$ si vuole stampare il numero di numeri distinti tra 1 e 4, cioè 2.
- Per $b_i = 4$ si vuole stampare il numero di numeri distinti nel sottoarray composto dal solo elemento 4, dando come risultato 1.

Soluzione

Il problema chiede di trovare il numero di elementi unici in un dato suffisso di un array. Più precisamente, il problema fornisce un array $a_1, a_2, \ldots, a_n$ e chiede di rispondere ad m query $b_1, b_2, \ldots, b_m$ ciascuna delle quali chiede: «quanti sono i valori distinti nel sottoarray di a con indici $b_i \ldots n$?»

■ Una soluzione $O(mn \log n)$

Cominciamo prima di tutto con il risolvere una versione semplificata del problema, in cui viene chiesto semplicemente quanti sono i valori distinti nell'intero array. Un modo per calcolarlo è il seguente:

- Ordina l'array.
- Conta quante volte due elementi consecutivi sono diversi. Per esempio se l'array ordinato fosse $1, 3, 3, 6, 7, 7$, allora il risultato sarebbe 3 perché succede solo tre volte che due elementi consecutivi sono diversi: $(1, 3)$, $(3, 6)$, $(6, 7)$.
- Il numero di valori distinti sarà dato dal numero di elementi consecutivi diversi, più 1.

Questo metodo ha tempo di esecuzione $O(nlogn + n) = O(nlogn)$. Un altro metodo, sempre con tempo di esecuzione $O(nlogn)$, è:

- Istanziamo un `std::set<int>`, una struttura dati che non ammette duplicati e che implementa l'inserimento e la ricerca in modo efficiente, fornita dalla libreria standard del C++.
- Inseriamo, uno alla volta, tutti gli elementi dell'array nel set (`set.insert(valore)`).
- Il numero di valori distinti sarà dato dalla dimensione del set (`set.size()`).

Per ogni query, eseguiamo uno dei metodi sopra sul sottoarray con indici $b_i \ldots n$. Il tempo di esecuzione sarà $O(mn \log n)$.

■ Una soluzione $O(n + m)$ con consumo di memoria $O(k)$

Questa soluzione crea un array booleano `visto[1...k]`, dove con k intendiamo il massimo valore che può presentarsi come elemento dell'array a.

Partendo dal fondo dall'array, per ogni elemento controlliamo il valore di `visto` sulla posizione relativa al valore dell'elemento. Se il valore è `false`, abbiamo incontrato un elemento che non abbiamo mai visto prima, altrimenti l'elemento è già presente in una delle posizioni analizzate.

L'idea è di precalcolare tutte le risposte alle possibili query (dalla query $n \ldots n$ alla query $1 \ldots n$) in un array ausiliario, per poi stamparle man mano che ci vengono richieste. Il procedimento sarà:

- Partiamo dal fondo dell'array e controlliamo il valore di `visto` in posizione a_n. Se è `false`, aumentiamo di 1 la risposta alla query attuale.
- Settiamo a `true` il valore di `visto` in posizione a_n.
- Continuiamo verso sinistra e controlliamo il valore di `visto` in posizione a_{n-1}. Se è `false`, aumentiamo di 1 la risposta alla query attuale.

- Settiamo a `true` il valore di `visto` in posizione a_{n-1}.
- ...
- Arrivati al primo elemento, controlliamo il valore di `visto` in posizione a_1. Se è `false`, aumentiamo di 1 la risposta alla query attuale.
- Settiamo a `true` il valore di `visto` in posizione a_1.

Finito il procedimento di precalcolo in tempo $O(n)$, sappiamo rispondere in tempo $O(1)$ a qualsiasi query ci verrà richiesta (perché abbiamo memorizzato tutte le risposte in un array ausiliario).

Aggiungiamo quindi un fattore $O(m)$ per tenere conto del tempo di risposta alle query, e otteniamo un tempo complessivo $O(n+m)$, che è ottimale (ovvero non si può fare di meglio). Tuttavia questa soluzione richiede una quantità di memoria $O(k)$, che può risultare eccessiva qualora il valore massimo dell'array a sia molto grande (infatti non riesce a risolvere tutti i casi nei limiti di memoria).

■ Una soluzione $O(n \log n + m)$

Si parte dal fondo come prima, ma senza usare l'array *visto*. Per limitare il consumo di memoria si inseriscono i valori in un `std::set<int>`. Come accennato precedentemente, la grandezza del set (`set.size()`) ci dirà quanti valori univoci abbiamo incontrato fin ora.

La differenza principale rispetto all'approccio con l'array è che settare una posizione di un array ad un certo valore (o controllarne il valore attuale) è un'operazione che impiega tempo $O(1)$. Con un set invece si impiega tempo proporzionale al logaritmo della dimensione corrente sel set. Abbiamo quindi fatto un tradeoff in cui abbiamo sacrificato l'efficienza per un miglior uso della memoria.

Il procedimento di precalcolo sarà quindi molto simile al precedente:

- Partiamo dal fondo dell'array e inseriamo l'elemento a_n in un set.
- La risposta alla query per il sottoarray con indici $n \dots n$ sarà data da `set.size()`.
- Continuiamo verso sinistra e inseriamo l'elemento a_{n-1} nello stesso set.
- La risposta alla query per il sottoarray con indici $n-1 \dots n$ sarà data da `set.size()`.
- ...
- Inseriamo l'elemento a_1 nello stesso set.
- La risposta alla query per il sottoarray con indici $1 \dots n$ sarà data da `set.size()`.

In definitiva, questo algoritmo ha tempo di esecuzione $O(n \log n + m)$.

Viaggio (viaggio)

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

Franco, dopo aver imparato a programmare, pensa di meritarsi una vacanza. Avendo letto statistiche sulla pericolosità di viaggiare in macchina, vuole prendere l'aereo. Nel mondo ci sono n città aeroportuali numerate da 1 a n. Tra una coppia di città può esserci una tratta aerea il cui biglietto costa c euro. Ogni tratta si può percorrere in entrambe le direzioni (allo stesso prezzo), e possono esserci diverse tratte tra la stessa coppia di città.

Franco parte dalla città numero 1 e vuole raggiungere la città n. Non gli interessa il tempo o il numero di scali da fare, ma vuole risparmiare il più possibile (si vedano i casi di esempio per chiarimenti). Franco ti chiede quanto deve spendere per raggiungere la destinazione.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: $n \leq 1\,000$, il prezzo di ciascuna tratta è 1.
- **Subtask 3** [**30 punti**]: $n \leq 1\,000$.
- **Subtask 4** [**20 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: contiene gli interi n e m, rispettivamente il numero di città e il numero di tratte.
- m righe seguono, ognuna contenente tre interi a_i, b_i e w_i, dove a_i è la partenza, b_i la destinazione e c_i il costo dell'i-esima tratta.

Il tuo programma dovrà stampare a video i seguenti dati:

- Un singolo intero: il costo minimo del viaggio. Nel caso in cui la destinazione non sia raggiungibile si stampi `-1`.

Assunzioni

- $1 \leq n \leq 100\,000$
- $1 \leq m \leq 100\,000$
- $1 \leq c_i \leq 1\,000\,000$
- $1 \leq a_i, b_i \leq n$

Esempi di input/output

stdin	stdout
3 4 1 2 2 2 3 5 2 3 11 1 3 9	7
5 6 1 2 1 2 4 1 2 3 1 4 3 1 1 4 1 3 5 1	3

Spiegazione

- Nel **primo caso d'esempio**, conviene passare per le città numero 1, 2 e 3 in sequenza, usando la prima e la seconda tratta disponibili.
- Nel **secondo caso d'esempio**, conviene passare per le città numero 1, 2 e 3, 5 in sequenza o, a pari prezzo, per le città 1, 4, 3, 5 in sequenza.

Soluzione

Questo problema si risolve con l'algoritmo di Dijkstra.

Quadrati perfetti (quadrati)

Limite di tempo: 0.1 secondi
Limite di memoria: 256 MiB

Giotto, essendosi stancato dei cerchi perfetti, ora vuole risolvere un problema sui quadrati perfetti. In particolare si chiede, dati due numeri interi positivi a e b, quanti siano i quadrati perfetti compresi tra a e b (estremi inclusi). Un numero x è un quadrato perfetto se esiste un valore z intero tale che $z \cdot z = x$.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: $a, b \leq 10^9$
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: due numeri interi a, b.

Il tuo programma dovrà stampare a video i seguenti dati:

- La prima riga contiene un singolo numero intero: il numero di quadrati perfetti nell'intervallo $[a, b]$.

Assunzioni

- $1 \leq a, b \leq 10^{18}$
- $a \leq b$

Esempi di input/output

stdin	stdout
1 1	1
2 3	0
1 10	3

Spiegazione

- Nel **primo caso d'esempio** l'unico quadrato perfetto da considerare è 1.
- Nel **secondo caso d'esempio** non ci sono quadrati perfetti compresi tra 2 e 3, quindi il risultato è 0.
- Nel **terzo caso d'esempio** i quadrati perfetti compresi nel'intervallo sono 1, 4 e 9.

Soluzione

Il problema chiede di trovare il numero di quadrati perfetti (numeri interi x tali che esiste un certo intero z per cui si ha $z^2 = x$) contenuti nell'intervallo $[a, b]$.

■ Una soluzione $O(\sqrt{b})$

La soluzione consiste nel generare con un ciclo tutti i numeri che sono quadrati perfetti. L'indice i su cui si cicla è la radice del quadrato perfetto (l'intero z accennato sopra), e ad ogni iterazione si controlla se $a \leq i^2 \leq b$. In caso affermativo si incrementa un contatore, che verrà stampato come risposta.

Dato che si itera su tutti i quadrati perfetti minori o uguali a b, la soluzione ha tempo di esecuzione $O(\sqrt{b})$.

■ Una soluzione $O(\log b)$

Tramite una ricerca binaria si cerca la radice del più piccolo quadrato perfetto maggiore o uguale ad a. Se tale valore, elevato al quadrato, risulta maggiore di b, la soluzione al problema è 0. Altrimenti si fa un'altra ricerca binaria che cerca la radice del più grande quadrato perfetto minore o uguale a b. La differenza delle due radici sommata ad 1 è la soluzione al problema.

Dato che facciamo due ricerche binarie su un intervallo di dimensione $O(b)$, la soluzione ha tempo di esecuzione $O(\log b)$.

Pile di mattoni (`mattoni`)

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

Franco deve costruire un muro. Un muro si può immaginare come una serie di "pile" di mattoni consecutive non necessariamente della stessa altezza. Per esempio, un muro con 5 pile (quindi di larghezza 5) potrebbe essere composto da pile di altezza: $3, 2, 2, 3, 4$.

A causa di recenti spiacevoli incidenti con la smerigliatrice autoadescante, la ditta di Franco ha stabilito che verranno costruiti solo muri di larghezza *dispari*.

È appena iniziata una nuova giornata lavorativa; il muro da costruire sarà composto da esattamente n pile consecutive (dove n è dispari). Franco riceverà m carichi, ciascuno contenente al più n mattoni. Ogni carico è contrassegnato da un indice l di inizio ed un indice r di fine: Franco ha il compito di disporre i mattoni di un certo carico (l, r) sulle pile l, $l + 1$, ..., $r - 1$, r, un mattone per pila.

Per esempio, il muro $3, 2, 2, 3, 4$ potrebbe essere il risultato della disposizione dei carichi:

$$(1, 3), (1, 1), (4, 5), (3, 5), (5, 5), (4, 5), (1, 2)$$

Al termine della giornata lavorativa, il capo vuole sapere qual è la *mediana* delle altezze delle n pile.

> ✍ La mediana di un array è quel valore che, ordinando l'array, si trova nella posizione centrale. Se l'array ha lunghezza dispari la mediana è univoca.

Per esempio, se il muro fosse $3, 2, 2, 3, 4$ la mediana delle altezze sarebbe uguale a 3. Infatti, in tal caso il muro "ordinato" sarebbe $2, 2, 3, 3, 4$, e nella posizione centrale troveremmo il numero 3.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**40 punti**]: $n, m \leq 1000$.
- **Subtask 3** [**30 punti**]: $n \leq 1\,000\,000$, $m \leq 10\,000$.
- **Subtask 4** [**30 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: due numeri interi n, m separati da spazio.
- Seguono m righe. Ciascuna riga contiene due numeri interi l, r separati da spazio (che rappresentano un carico di mattoni).

Il tuo programma dovrà stampare a video i seguenti dati:

- Un singolo numero intero: la mediana delle altezze delle pile di mattoni.

Assunzioni

- $1 \leq n, m \leq 1\,000\,000$.
- n è dispari.

Esempi di input/output

stdin	stdout
5 7 1 3 1 1 4 5 3 5 5 5 4 5 1 2	3

Soluzione

Il problema chiede la mediana di un array lungo n, il quale viene "costruito" con una serie di incrementi unitari su m suoi sottointervalli.

La soluzione può essere scomposta in due parti:

1. Costruzione del muro (array delle altezze).
2. Identificazione della mediana dell'array.

Il punto 2 si può risolvere facilmente in $O(n \log n)$, ordinando l'array e prendendo l'elemento centrale. Esistono tuttavia diversi modi per farlo in tempo $O(n)$.[3]

Come si può intuire, il punto 1 è quello che richiede più inventiva.

■ Una soluzione $O(mn)$

Una soluzione intuitiva è quella di simulare effettivamete la "posa" dei mattoni sul muro.

```
for (int i=1; i<=m; i++) {
    // leggi intervallo: [l, r]
    for (int j=l; j<=r; j++)
        muro[j]++;
}
```

Il ciclo interno esegue $r-l+1$ iterazioni, che è $O(n)$. Il tempo di esecuzione è quindi $O(mn)$. Aggiungendo il costo del calcolo della mediana (indipendentemente da quale dei metodi elencati sopra usiamo) il tempo di esecuzione complessivo rimane $O(mn)$.

■ Una soluzione $O(m+n)$

Una soluzione più efficiente, che elimina il ciclo interno, è quella di costruire "in due passate" l'array muro. Facciamo una prima passata in cui memorizziamo soltanto *i cambiamenti di valore*. Per esempio: abbiamo un muro $1, 2, 2, 3, 2$; arriva un carico identificato con $[2, 4]$ (quindi ci aspettiamo che il muro dovrà diventare $1, 3, 3, 4, 2$). I cambiamenti di valore subiti dal muro originario sono:

- $+1$, a partire dal secondo elemento.
- -1, a partire dal quinto elemento (quello dopo il quarto).

L'utilità del secondo cambiamento è semplicemente quella di "annullare" l'effetto del primo cambiamento dove non vogliamo che venga applicato.

Tornando all'esempio, abbiamo quindi un muro $(1, 2, 2, 3, 2)$ ed un array ausiliario con i cambiamenti subiti $(0, +1, 0, 0, -1)$. Come otteniamo il "nuovo muro"? Ci basta scorrere l'array ausiliario tenendo traccia della *somma parziale* dei suoi elementi, e di volta in volta incrementare l'elemento i del muro con il valore della somma parziale degli elementi $1, 2, \ldots, i$ dell'array ausiliario. Tempo $O(n)$.

Ora dobbiamo fare un'importante osservazione: possiamo "accumulare" tutti gli m cambiamenti di valore ed infine applicarli (non stiamo facendo altro che raggruppare ed eseguire in un altro ordine le varie somme).

[3] Per approfondire: https://en.wikipedia.org/wiki/Selection_algorithm

Prendiamo l'esempio presente nel testo del problema, che riceve i seguenti carichi:

$$(1,3),(1,1),(4,5),(3,5),(5,5),(4,5),(1,2)$$

Avremo che l'array ausiliario attraverserà le seguenti fasi:

$$\begin{array}{l}
(\ 0,\ \ 0,\ \ 0,\ \ 0,\ \ 0) \\
(+1,\ \ 0,\ \ 0,-1,\ \ 0) \\
(+2,-1,\ \ 0,-1,\ \ 0) \\
(+2,-1,\ \ 0,\ \ 0,\ \ 0) \\
(+2,-1,+1,\ \ 0,\ \ 0) \\
(+2,-1,+1,\ \ 0,+1) \\
(+2,-1,+1,+1,+1) \\
(+3,-1,\ \ 0,+1,+1)
\end{array}$$

Il codice sarà più o meno il seguente:

```
for (int i=1; i<=m; i++) {
    // leggi intervallo: [l, r]
    muro[l]++;
    if (r+1 <= n) // non scrivere fuori dall'array
        muro[r+1]--;
}
```

Osserviamo che si può evitare l'uso dell'array ausiliario, ci basta salvare i cambiamenti direttamente nell'array `muro` e infine trasformarlo nell'array delle sue somme prefisse (con un algoritmo in place):

```
for (int i=2; i<=n; i++)
    muro[i] += muro[i-1];
```

☞ **Curiosità:** se avete studiato le derivate e gli integrali dovreste essere in grado di trovare delle analogie con questo problema, in particolare:

- Passare dall'array dei cambiamenti di valore al muro è analogo ad un integrale.
- L'operazione inversa (ovvero passare dal muro all'array dei suoi cambiamenti di valore) è analoga ad una derivata.

In definitiva sappiamo costruire l'array in tempo $O(m+n)$. Aggiungendo il costo del calcolo della mediana otteniamo $O(m+n+n\log n) = O(m+n\log n)$ se usiamo l'ordinamento, altrimenti $O(m+n+n) = O(m+n)$.

Edoardo e la lotteria (`lotteria`)

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

Il lotto è una lotteria in cui i giocatori giocano numeri a loro scelta. A Edoardo piace giocare il lotto di Cesena. In questo lotto bisogna scegliere n numeri da 1 a m. Se la lista scelta coincide con la lista vincente, ci si porta a casa una grande scorta di limoni. Edoardo pensa di avere uno schema fortunato: egli sceglie ogni numero almeno doppio rispetto al precedente. Quindi, se $n = 3$ e $m = 6$ le possibili liste fortunate sono:

$$(1, 2, 4) \qquad (1, 2, 5) \qquad (1, 2, 6) \qquad (1, 3, 6)$$

Dati n e m, Edoardo vuole sapere quante sono le possibili liste fortunate di numeri. Siccome tale numero può essere molto elevato, si stampi soltanto il resto della divisione per $1\,000\,000\,007$.

✍ Per eseguire l'operazione di modulo si può utilizzare l'operatore `%` del C/C++.
Ad esempio: `5 % 3 = 2`.

Quindi se il risultato fosse (per esempio) 10^{18}, il tuo programma dovrebbe stampare il valore di `1000000000000000000 % 1000000007`, ovvero: `49`.

☞ Per evitare di andare in overflow durante i calcoli intermedi, è necessario sfruttare l'aritmetica modulare:

- `(a + b) % c == ((a % c) + (b % c)) % c`
- `(a * b) % c == ((a % c) * (b % c)) % c`

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**30 punti**]: $n \leq 5$, $m \leq 200$.
- **Subtask 3** [**40 punti**]: $m \leq 1000$.
- **Subtask 4** [**30 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: due numeri interi n, m separati da spazio.

Il tuo programma dovrà stampare a video i seguenti dati:

- Un singolo numero intero: il numero di liste valide $\pmod{1\,000\,000\,007}$.

Assunzioni

- $1 \leq n \leq 20$.
- $1 \leq m \leq 200\,000$.

Esempi di input/output

stdin	stdout
3 6	4

Soluzione

Il problema chiede di contare le sottosequenze lunghe n di $1 \dots m$, tali che per ogni coppia (a, b) di elementi consecutivi si ha $a \leq 2 \cdot b$.

■ Una soluzione $O(m^n)$

Un'idea banale, che risolve solo il primo subtask nei limiti di tempo assegnati, è quella di fare n cicli nidificati:

```
risposta = 0;
for (int i=1; i<=m; i++)
    for (int j=2*i; j<=m; j++)
        for (int k=2*j; k<=m; k++)
            risposta++;
```

L'esempio qui sopra funziona quando $n = 3$. Dato che nel primo subtask vale $n \leq 5$, si possono fare 5 casi e fare cicli nificati in ognuno dei casi. Oppure si possono fare sempre e comunque 5 cicli nidificati, controllando e uscendo al momento giusto dai cicli superflui. Oppure si può fare ricorsivamente, che porta sulla buona strada per trovare la soluzione seguente.

■ Una soluzione $O(m^2 n)$

Una soluzione più efficiente, che risolve tutto tranne l'ultimo subtask, si ottiene approcciando il problema con la *programmazione dinamica*. Cerchiamo un'equazione di ricorrenza che ci permetta di esprimere la risposta che ci interessa in funzione di qualche parametro. Chiamiamo $f(i, k)$ il numero di sottosequenze che ci interessano sotto le seguenti condizioni:

- consideriamo *non* tutto l'intervallo $1 \dots m$, ma soltanto gli elementi successivi ad i, vale a dire: $i + 1 \dots m$.
- abbiamo già "messo nella schedina" k numeri (su n disponibili), scelti tra gli elementi che non consideriamo $(1 \dots i)$, quindi ce ne rimangono $n - k$ da scegliere.
- l'elemento i è tra i k che abbiamo già scelto.

Assumendo di aver scritto questa funzione f, la soluzione al problema sarà data da $f(0, 0)$. Ovvero, scelgo come primo elemento della schedina un elemento non esistente ($i = 0$) al quale posso "attaccare" uno qualsiasi dei numeri successivi (perché $2 \cdot i$ rimane zero) e assumo di aver già preso $k = 0$ numeri, quindi mi rimangono ancora n numeri da scegliere nella parte restante dell'intervallo.

Adesso dobbiamo effettivamente scrivere la funzione f. Intuitivamente, se $k = n$, avremo che $f(i, k) = 1$, indipendentemente dal valore di i (perché è "finito lo spazio" nella schedina). Se invece $k < n$, abbiamo ancora $n-k$ elementi da scegliere, e dobbiamo contare in quanti modi possiamo farlo (magari riutilizzando f stessa!).

Mettiamoci quindi nel caso $k < n$. Proviamo a scegliere il "prossimo" elemento della schedina. Per fare questo ci basta iterare da $i + 1$ a m verificando, per ciascun elemento, se può o non può essere inserito nella schedina. *Osservazione:* l'elemento i è l'ultimo che "abbiamo preso", quindi per la definizione del problema avremo che tutti e soli gli elementi da $2 \cdot i$ a m vanno bene.

A questo punto sappiamo chi sono i "candidati" ad essere il prossimo elemento della schedina. Ora sfruttiamo f per sapere "quante schedine" possibili ci sono per ogni candidato. In definitiva avremo questa ricorrenza:

$$f(i,k) = \begin{cases} 1 & \text{se } k = n \\ \sum_{j=2\cdot i}^{m} f(j,k+1) & \text{altrimenti} \end{cases}$$

Implementandola direttamente, otteniamo un algoritmo ricorsivo che impiega tempo esponenziale (probabilmente, è ciò che avremmo ottenuto scrivendo ricorsivamente la soluzione $O(m^n)$). Tuttavia, dato che abbiamo "decomposto" il problema in modo da far dipendere f da due soli parametri, ci basta rendere la funzione ricorsiva un po' più intelligente.

Come prima cosa, nel corpo della funzione $f(i,k)$, verifichiamo se per caso f è "già stata richiamata" con questi esatti valori di i e di k (usando una matrice globale). Se f è già stata richiamata, andiamo a prendere dalla matrice il valore restituito l'ultima volta. In questo modo ogni combinazione di parametri verrà processata una sola volta, e le chiamate superflue impiegheranno $O(1)$.

La complessità del nostro algoritmo sarà quindi data dal numero di combinazioni possibili di parametri, ovvero $|i| \cdot |k|$, con cui intediamo "il numero di valori assumibili da i" per "il numero di valori assumibili da k", ovvero $m \cdot n$. A questo dobbiamo aggiungere però il tempo che impieghiamo all'interno della funzione ricorsiva, che è $O(m)$ per via del ciclo che itera sui "candidati".

In definitiva, abbiamo un algoritmo $O(m \cdot n \cdot m) = O(m^2 n)$.

■ Una soluzione $O(mn)$

Per scendere da $O(m^2 n)$ a $O(mn)$ dobbiamo liberarci del ciclo all'interno della funzione f. Un modo per farlo è quello di definire una nuova funzione g:

$$g(x,k) = \sum_{i=x}^{m} f(i,k)$$

la quale andrà a sua volta "cachata" con una matrice come fatto prima per f (in gergo si dice "memoizzata", senza la **r**), e infine sostituita al ciclo presente nel corpo della funzione f, la quale si può ora scrivere così:

$$f(i,k) = \begin{cases} 1 & \text{se } k = n \\ g(2 \cdot i, k+1) & \text{altrimenti} \end{cases}$$

Notiamo però che adesso è g stessa ad avere un ciclo al suo interno. Lo possiamo far sparire molto facilmente, calcolando la funzione in quest'altro modo:

$$g(x,k) = \begin{cases} 0 & \text{se } x > m \\ f(x,k) + g(x+1,k) & \text{altrimenti} \end{cases}$$

che è essenzialmente l'operazione di somma $\sum$ implementata ricorsivamente.

Abbiamo quindi due funzioni, ciascuna con $m \cdot n$ combinazioni di parametri. Il costo delle operazioni eseguite all'interno di ognuna è $O(1)$. L'algoritmo quindi ha complessità $O(m \cdot n \cdot 1) + O(m \cdot n \cdot 1) = O(mn)$.

Equazione non quadratica (equazione)

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

Si consideri l'equazione $x^2 + s(x) \cdot x - n = 0$ dove x ed n sono numeri interi positivi, ed $s(x)$ è una funzione definita come la somma delle cifre di x. Ti viene dato n, e il tuo compito è trovare il più piccolo valore di x che soddisfa l'equazione.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: $n \leq 200\,000\,000$
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: il numero intero n.

Il tuo programma dovrà stampare a video i seguenti dati:

- La prima riga contiene un singolo numero intero: il valore x minimo che soddisfa l'equazione, o -1 se tale valore non esiste.

Assunzioni

- $2 \leq n \leq 10^{18}$.

Esempi di input/output

stdin	stdout
2	1
110	10
4	-1

Soluzione

Il problema chiede di trovare, se possibile, la più piccola soluzione dell'equazione $x^2 + s(x) \cdot x - n = 0$.

■ Una soluzione $O(\sqrt{n})$

Un approccio intuitivo è quello di provare tutte le x. Ci basta provare fintantoché $x^2 \leq n$ dato che il termine $s(x) \cdot x$ non può essere negativo. Il tempo di esecuzione di questo algoritmo è quindi $O(\sqrt{n})$.

■ Una soluzione $O(\log n)$

Osserviamo che l'intervallo dei possibili valori assumibili dal termine $s(x)$ non è molto grande: sappiamo già che $x^2 \leq n$, e dato che $n \leq 10^{18}$ si avrà $x \leq 10^9$. In altre parole, per ogni possibile soluzione x la "lunghezza" (in termini di cifre decimali) di x non supera le 10 cifre. Quindi il valore massimo di $s(x)$ si avrà quando $x = 9999999999$, e varrà $s(9999999999) = 10 \cdot 9 = 90$.

Ci basta quindi provare a sostituire a $s(x)$ tutti gli interi compresi tra 0 e 90, ottenendo ogni volta una normalissima equazione quadratica. La risolviamo e infine controlliamo che l'attuale valore (scelto arbitrariamente) di $s(x)$ sia uguale alla somma delle cifre della soluzione trovata. Se è così, abbiamo un candidato: dobbiamo tenere traccia del candidato minimo che incontriamo.

Assumendo di saper risolvere un equazione di secondo grado in tempo $O(1)$, questo metodo ha un tempo di esecuzione proporzionale alla dimensione dell'intervallo di valori assumibili da $s(x)$, che è 9 moltiplicato per la "lunghezza" (ovvero il $\log_{10}$) di $\sqrt{n}$. Ovvero: $O(9 \log \sqrt{n}) = O(\log \sqrt{n}) = O(\log n^{\frac{1}{2}}) = O(\frac{1}{2} \log n) = O(\log n)$.

Capitale di Alberolandia (albero)

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

Il paese di Alberolandia è composto da n città (numerate da 1 a n) connesse da strade percorribili in un solo senso. In tutto ci sono $n-1$ strade nel paese. Se non prendessimo in considerazione la direzione delle strade, allora ogni coppia di città sarebbe collegata da almeno un percorso. Dopo la recente rivoluzione, il governo si trova a dover decidere una nuova capitale per Alberolandia.

Per poter chiamare capitale una città, deve essere possibile raggiungere qualunque altra città a partire da essa. Se la città a viene scelta come capitale, le varie strade devono essere orientate in modo tale che, in qualche modo, si possa andare da a a tutte le altre città. Per questo motivo alcune strade devono essere invertite di direzione. Invertire la direazione di una strada può essere costoso (vanno spostati i cartelli di "senso unico"). Aiuta il governo a scegliere la capitale in modo tale da dover invertire il minor numero di strade.

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**]: i casi di esempio mostrati sotto.
- **Subtask 2** [**50 punti**]: $n \leq 1000$
- **Subtask 3** [**50 punti**]: nessuna limitazione.

Formato di input/output

Il tuo programma dovrà leggere da tastiera i seguenti dati:

- Riga 1: un numero intero n, il numero di città.
- Le successive $n-1$ righe: l'i-esima riga contiene la descrizione della i-esima strada. In particolare conterrà gli interi s_i e d_i che indicano che l'i-esima strada può essere percorsa da s_i a d_i.

Il tuo programma dovrà stampare a video i seguenti dati:

- La prima riga contiene un singolo numero intero: il numero minimo di strade che bisogna invertire.
- La seconda riga contiene tutti i modi ottimi per scegliere la capitale: una sequenza di indici in ordine crescente.

Assunzioni

- $2 \leq n \leq 100\,000$.

Esempi di input/output

stdin	stdout
4 1 4 2 4 3 4	2 1 2 3
3 2 1 2 3	0 2

Soluzione

Il problema ci dà un albero diretto non radicato (lo capiamo perché ci dice che è un grafo diretto, che è debolmente connesso e che ha n nodi e $n-1$ archi). Ci chiede di trovare, tra tutti i nodi, quelli che possono diventare "capitali" con il minor numero di inversioni di archi possibili. Se un nodo viene scelto come capitale, gli archi devono essere invertiti per rendere tutti i nodi raggiungibili dalla capitale.

■ Una soluzione $O(n^2)$

Memorizziamo l'albero diretto come un albero non diretto, dove ad ogni arco è associato un costo. Il costo sarà 0 per gli archi originali, mentre sarà 1 per gli archi inversi. Proviamo a chiamare *capitale* un nodo alla volta, e li proviamo tutti. Supponiamo che il nodo u sia il candidato attuale: eseguiamo una visita in profondità (DFS) usandola per calcolare il numero di archi da invertire: esso è uguale alla somma del numero di archi da invertire nei sottoalberi più il numero di archi uscenti da u che hanno costo 1.

Una DFS in generale ha tempo di esecuzione proporzionale alla somma del numero di nodi e del numero di archi, in questo caso quindi è $O(n)$. Dato che eseguiamo una DFS per ciascun nodo, il tempo complessivo è $O(n^2)$.

■ Una soluzione $O(n)$

Scegliamo arbitrariamente un nodo, per esempio 1, e diciamo che quello è la radice dell'albero. Adesso il grafo in input è un albero diretto *radicato*. Notiamo che ora ciascun arco è orientato verso "su" (verso la radice) o verso "giu" (verso le foglie). Chiamiamo il primo tipo di archi "rossi" e i secondi "verdi".

Scegliendo il vertice 1 come capitale il numero di archi da invertire è pari al numero di archi rossi nell'albero (ovvero il costo per renderli verdi) dato che, oltre agli altri nodi, dobbiamo raggiungere anche tutte le foglie.

Se invece scegliamo un altro vertice $u \neq 1$ come capitale notiamo che bisogna rendere rossi tutti gli archi tra la radice ed il nodo u, in modo che si possa raggiungere la radice partendo da u (tali archi sono verdi se immaginiamo u come nuova radice).

Calcoliamo quindi, per ogni nodo, il numero di archi totali (sia verdi che rossi) ed il numero di archi rossi che lo separano dalla radice con una DFS. Il numero di archi da invertire, quando la capitale è u, sarà:

$$\begin{aligned}&\text{“totale rossi”} - \text{“rossi tra radice e } u\text{”} + \text{“verdi tra radice e } u\text{”}\\ =&\text{“totale rossi”} - 2 \cdot \text{“rossi tra radice e } u\text{”} + \text{“archi tra radice e } u\text{”}\end{aligned}$$

Il calcolo del numero di archi rossi tra un qualsiasi nodo e la radice si può fare in un unica DFS. Si può sfruttare il fatto che tali archi hanno costo 1 nella nostra rappresentazione dell'albero.

Dato che questa soluzione richiede una DFS ed un controllo per ogni nodo, il suo tempo di esecuzione è $O(n)$.