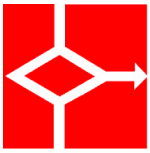
Ministero dell'Istruzione dell'Università e Ricerca



AICA
Associazione Italiana per l'Informatica ed il Calcolo Automatico

Olimpiadi Italiane di Informatica 2012

Testi e Soluzioni ufficiali dei problemi

Problemi a cura di
Luigi Laura

Coordinamento
Monica Gati

Testi dei problemi
Giorgio Audrito, Matteo Boscariol, Roberto Grossi, Luigi Laura,
Giuseppe Ottaviano, Giovanni Paolini, Romeo Rizzi

Soluzioni dei problemi
William Di Luigi, Luca Versari

Supervisione a cura del Comitato per le Olimpiadi di Informatica

Indice

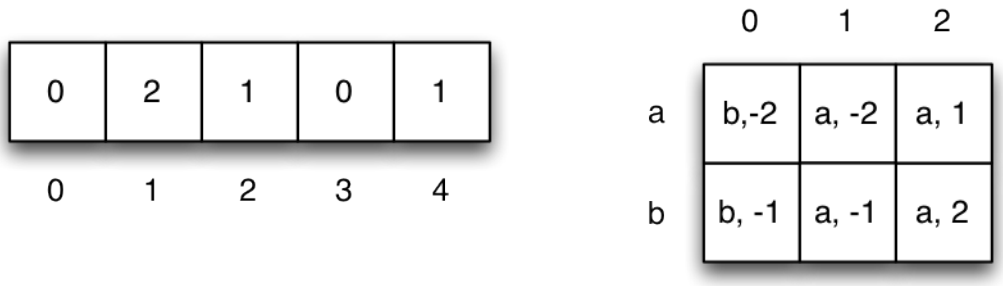
1 Entscheidungsproblem, o problema della fermata (`fermata`)

1.1 Descrizione del problema

Nota storica: nel suo famoso articolo del 1937, On computable numbers, with an application to the Entscheidungsproblem, Alan Turing dimostrò che il problema della fermata non è decidibile: tra le conseguenze, quindi, il fatto che non è possibile scrivere un programma che decida se una macchina di Turing si arresti, dato un particolare input.

Turing però è convinto che il problema della fermata sia decidibile nel modello di seguito descritto, dove si utilizza una macchina di Turing di sola lettura. La macchina ha un nastro di N celle, numerate da 0 a $N-1$, da sinistra verso destra. In ogni cella c'è un numero intero, e le celle sono di sola lettura: la macchina non può cambiare il contenuto della cella. La macchina di Turing ha una tabella di transizione, che in funzione dello stato attuale e del numero letto, cambia lo stato interno della macchina e comanda alla macchina di spostarsi di un certo numero di celle, verso destra o verso sinistra. La cella numero 0 è una cella speciale: quando la macchina di Turing arriva nella cella 0, termina la sua computazione e si ferma.

Figura 1:



Considerate la figura: qui vedete il nastro, con 5 celle numerate da 0 a 4, contenenti interi compresi tra 0 e 2, e la tabella di transizione, che in funzione dei due stati possibili della macchina (a e b) e dei tre interi letti dalla cella, riporta lo stato successivo e lo spostamento della macchina, rappresentato da interi positivi per spostamenti verso destra e interi negativi per spostamenti verso sinistra. Per esempio, supponiamo che la macchina di Turing sia inizialmente nello stato a e che parta dalla cella 1. Nella cella 1 la macchina legge l'intero 2: come si vede dalla tabella, la macchina di Turing rimane nello stato a e si sposta di una cella a destra. Finisce quindi nella cella 2, dove legge l'intero 1: a questo punto rimane nello stato a e si sposta di due celle a sinistra; raggiunge quindi la cella 0 e si ferma.

Se la macchina di Turing parte, sempre nello stato a, dalla cella 2 vediamo che termina direttamente nella cella 0, fermandosi. Viceversa, se la macchina di Turing, sempre nello stato a, parte dalla cella 3 si vede che la macchina cambia stato, passando allo stato b e spostandosi di due celle all'indietro. Si ritrova quindi nella cella 1 ma qui, dalla tabella di transizione, si vede che ritorna nello stato a e si risposta di due celle in avanti, ritornando nella cella 3. Da qui continuerà a spostarsi, alternativamente, di due celle in avanti e due celle indietro, cambiando stato a ogni spostamento. Quindi, la macchina di Turing a partire da questa configurazione iniziale, NON termina. Il vostro compito è quello di aiutare Alan Turing, scrivendo un programma che, presa in ingresso la descrizione di una macchina di Turing, lo stato iniziale della macchina e la configurazione del nastro, stampi tutti e soli i numeri delle celle tali che, se la computazione parte da quella cella, la macchina di Turing si arresta. Ad esempio, con riferimento alla figura, le celle per cui la macchina di Turing termina, partendo

dallo stato iniziale a sono la 0 (che per definizione appartiene alla soluzione), la 1, la 2 e la 4. Partendo dallo stato iniziale b le celle in cui la macchina di Turing termina sono la 0, la 2 e la 3.

1.2 Dati di input

Il file di input contiene nella prima linea 3 interi N, S, C che denotano, rispettivamente, la lunghezza del nastro N, il numero di stati S della macchina di Turing e il numero di valori distinti C possibili nelle celle (i valori distinti su nastro sono rappresentati da interi compresi tra 0 e $C-1$). Le successive $S \cdot C$ righe contengono la tabella di transizione della macchina di Turing, con quattro interi per ogni riga: stato corrente, carattere letto, nuovo stato, spostamento (positivo o negativo). Le successive N righe rappresentano il contenuto di ogni cella, in ordine dalla cella 0 alla cella $N-1$: un intero per ogni riga. L'esempio qui sotto si riferisce alla figura; lo stato a è rappresentato dall'intero 0 e lo lo stato b è rappresentato dall'intero 1.

1.3 Dati di output

Il file di output contiene nella prima riga T, il numero di celle per cui la macchina di Turing, partendo da esse nello stato iniziale 0, termina la sua computazione. Le successive T righe contengono, disposti in ordine crescente, i numeri delle celle in cui la computazione della macchina di Turing si arresta (compresa la cella numero 0).

1.4 Assunzioni

- $1 \leq S \cdot C \leq 10.000.000$
- $1 \leq S \cdot N \leq 1.000.000$.

1.5 Valutazione delle soluzioni

- (SubTask 1 - 5 punti) Questo subtask è costituito da una sola istanza: il caso di esempio mostrato qui sotto.
- (SubTask 2 - 14 punti) Nelle istanze di questo subtask si ha che $S \ 1$ e, nella tabella di transizione, tutti i valori degli spostamenti sono compresi tra -1 e 1.
- (SubTask 3 - 21 punti) Nelle istanze di questo subtask vale $N, S \leq 100$.
- (SubTask 4 - 24 punti) Nelle istanze di questo subtask vale $S \cdot N \leq 1.000.000$ e $C \leq 10$
- (SubTask 5 - 36 punti) Nelle istanze di questo subtask non ci sono vincoli particolari.

1.6 Esempi di input/output

File `input.txt`	**File** `output.txt`

`5 2 3`
`0 0 1 -2`
`0 1 0 -2`
`0 2 0 1`
`1 0 1 -1`
`1 1 0 -1`
`1 2 0 2`
`0`
`2`
`1`
`0`
`1`
	`4`
	`0`
	`1`
	`2`
	`4`

1.7 Codice della soluzione

```cpp
#include <cstdio>
using namespace std;
/* Array contenente le informazioni sulle coppie (posizione, stato)
 * Ogni cella vale 0 se ancora non si sa nulla su quella coppia,
 * 1 se si sa che portera' nella posizione 0, e 2 se portera' a un ciclo.
 * Ha dimensione max(N*S) = 1000000
 */
char din[1000000];
/* Tabelle di transizione. st contiene lo stato successivo alla coppia
 * (numero, stato), mentre pt contiene il numero di celle di cui spostarsi
 * (positivo se ci si sposta a destra, negativo se ci si sposta a sinistra).
 * Hanno dimensione max(C*S) = 10000000
 */
int st[10000000];
int pt[10000000];
/* Tutte le matrici sono "appiattite" per evitare allocazioni dinamiche di
 * memoria. In particolare la cella [a][b] viene salvata nella posizione
 * [a*S+b], dato che b e' sempre nel range [0,S).
 *
 * Array che rappresenta i caratteri contenuti in ogni posizione, di dimensione
 * max(N) = 1000000
 */
int nastro[1000000];
int N, C, S;
/* Effettua una "ricerca in profondita'" sulle coppie (posizione,stato),
 * restituendo true se la coppia arriva in 0 e false se invece entra in
 * un ciclo. Innanzitutto controlla se la cella era gia' stata visitata.
 * Se lo era, restituisce cio' che aveva trovato la volta precedente.
 * Altrimenti, verifica se la coppia "successiva" lo porterebbe a 0 o in
 * un ciclo e restituisce la stessa cosa.
 */
bool dfs(int n, int s){
    /* Se n e' 0, vuol dire che sono appena arrivato nella cella 0,
     * dunque restituisco true.
     */
    if(n == 0) return true;
    /* Se ero gia' stato nella cella (n,s), allora la cella corrispondente
     * non e' 0. Se e' 1 vuol dire che la cella va in 0, altrimenti va in
     * un ciclo.
     */
    if(din[n*S+s] != 0)
        if(din[n*S+s] == 1)
            return true;
        else
            return false;
    /* Nel momento in cui inizio a visitare la cella "figlia", assumo che
     * la cella corrente porti a un ciclo. Infatti, se ci torno prima di
     * aver concluso la visita del figlio, non arrivero' mai in 0 perche'
     * continuero' a ripetere le stesse coppie di posizione e stato.
     */
    din[n*S+s] = 2;
    /* Visito la cella successiva. Se la funzione restituisce true, allora
     * la cella successiva porta in 0 e dunque anche questa cella porta in
     * 0. Dunque imposto la cella (n,s) a 1 e restituisco true.
```

```
55       */
56      if(dfs(n+pt[nastro[n]*S+s], st[nastro[n]*S+s])){
57          din[n*S+s] = 1;
58          return true;
59      }
60      /* Altrimenti la cella successiva porta a un ciclo, dunque lascio
61       * la cella (n,s) impostata a 2 e restituisco false.
62       */
63      return false;
64  }
65  int main(){
66      // Apertura dei files di input e di output
67      FILE* in = fopen("input.txt","r");
68      FILE* out = fopen("output.txt","w");
69      // Leggo le variabili N, S e C
70      fscanf(in, "%d%d%d", &N, &S, &C);
71      // Leggo le tabelle di transizione
72      int a, b, c, d;
73      for(int i=0; i<S*C; i++){
74          fscanf(in,"%d%d%d%d", &a, &b, &c, &d);
75          st[b*S+a] = c;
76          pt[b*S+a] = d;
77      }
78      // Leggo il nastro
79      for(int i=0; i<N; i++)
80          fscanf(in, "%d", &nastro[i]);
81      // Sicuramente la cella (0,0) mi porta a concludere.
82      din[0] = 1;
83      int cnt = 0;
84      /* Faccio partire la visita da ogni cella del tipo (i,0). Se questa
85       * cella porta a 0, allora ho trovato una posizione iniziale che porta
86       * in 0 e aumento il contatore.
87       */
88      for(int i=0; i<N; i++)
89          if(dfs(i, 0))
90              cnt++;
91      // Stampo il numero di posizioni che portano a 0.
92      fprintf(out, "%d\n", cnt);
93      // Per ogni cella (i,0), controllo se porta in 0 e se e' cosi' stampo i.
94      for(int i=0; i<N; i++)
95          if(din[i*S+0] == 1)
96              fprintf(out, "%d\n", i);
97      // Chiudo i files di input e di output.
98      fclose(in);
99      fclose(out);
100 }
```

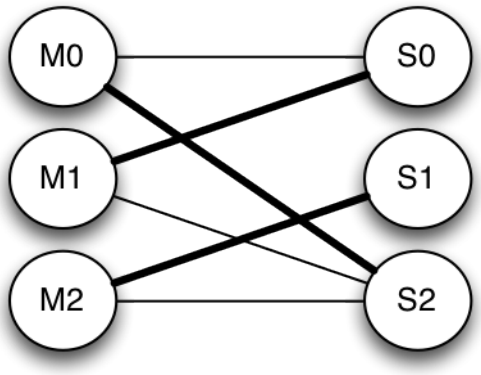
2 La battaglia del convoglio (`convoglio`)

2.1 Descrizione del problema

Nota storica: tra il 7 e il 10 marzo del 1943 c'è stata nell'Atlantico quella che è stata definita *la più grande battaglia di convogli mai combattuta* . Sottomarini tedeschi si comunicavano, in maniera cifrata, le posizioni dei convogli americani da attaccare. Gli alleati conoscevano, ovviamente, le posizioni dei loro convogli, ed intercettavano le comunicazioni dei tedeschi. Le informazioni acquisite da queste comunicazioni cifrate, insieme alle posizioni note dei convogli americani, sono state fondamentali per il lavoro di Alan Turing a Bletchley Park: qui Turing ha ideato la macchina Bomba, che ha consentito agli alleati di rompere il codice di Enigma, la macchina per comunicazioni cifrate dei tedeschi.

Torniamo alla battaglia: un convoglio americano, composto da N navi, è in viaggio nell'Atlantico. Sottomarini tedeschi si comunicano le posizioni delle navi e si coordinano per l'attacco. Gli alleati intercettano le comunicazioni tedesche ma riescono a decrittare solo parzialmente i messaggi: non sempre si riesce a identificare di quale nave stiano parlando i tedeschi, e spesso più di una nave americana potrebbe essere quella a cui fanno riferimento. In particolare, se indichiamo con $M_0..M_{N-1}$ gli N messaggi intercettati, e con S_0 .. S_{N-1} le N navi della flotta, alla luce di quanto decodificato ogni messaggio può riferirsi a una o più navi, come si vede nella figura (dove N 3), dove, per esempio, il primo messaggio può riferirsi sia alla prima ($S0$) che alla terza ($S2$) nave.

Figura 2:



Turing riesce a trovare una **corrispondenza univoca** tra i messaggi e le navi: una corrispondenza in cui *ad ogni messaggio distinto corrisponde una nave distinta*. Per esempio, le 3 linee a tratto spesso in figura evidenziano 3 coppie messaggio-nave ($M0-S2$, $M1-S0$, e $M2-S1$). Questa e' una corrispondenza univoca in quanto:

- per ogni i $1, 2, 3$ esiste uno ed un solo j tale che la coppia $Mi - Sj$ e' stata inclusa;
- per ogni j $1, 2, 3$ esiste uno ed un solo i tale che la coppia $Mi - Sj$ e' stata inclusa.

Per poter proteggere la flotta bisogna essere sicuri della corrispondenza, e quindi dobbiamo ora accertarci che non esistano altre corrispondenze univoche interamente costituite da coppie messaggio-nave consentite dall'istanza (gli archi in figura, sia in grassetto che in tratto semplice). Per esempio, nel caso della Figura 1 esiste anche una seconda corrispondenza univoca: $M0 - S0$, $M1 - S2$, $M2 - S1$.

Aiutate Turing a capire se la corrispondenza univoca da lui trovata è anche unica!

2.2 Dati di input

La struttura del file di input è la seguente: la prima riga contiene 2 interi, N e M, che rappresentano, rispettivamente, il numero di navi (che coincide con il numero dei messaggi intercettati) e il numero complessivo di possibili corrispondenze tra messaggi e navi. I messaggi sono identificati da interi compresi tra 0 e $N-1$, e anche le navi sono identificate da interi compresi tra 0 e $N-1$.

Le successive M righe contengono una coppia ordinata di interi rappresentanti, rispettivamente, un messaggio e una nave corrispondente. Di queste M righe, le prime N contengono tutti i messaggi e tutte le navi, e identificano la corrispondenza univoca trovata da Turing.

Con riferimento alla figura, rappresentando i messaggi M0, M1 e M2 con gli interi 0,1 e 2, e le navi S0, S1 e S2 con gli interi 0,1 e 2, l'istanza nella figura viene rappresentata dal file di input a fondo pagina.

2.3 Dati di output

Se non esiste nessuna altra corrispondenza possibile, il file di output deve essere costituito da una sola linea, contenente l'intero -1. Altrimenti, se esiste un'altra soluzione, il file di testo contiene N linee, corrispondenti alla soluzione trovata, rappresentata come lista di coppie di interi, messaggio e nave, ordinate per numero di messaggio.

Con riferimento alla figura, rappresentando i messaggi M0, M1 e M2 con gli interi 0,1 e 2, e le navi S0, S1 e S2 con gli interi 0,1 e 2, la seconda possibile soluzione univoca viene rappresentata dal file di output a fondo pagina.

2.4 Assunzioni

- $N \leq 100.000$, $M \leq 200.000$

2.5 Valutazione delle soluzioni

- (SubTask 1 - 5 punti) Questo subtask è costituito da una sola istanza: il caso di esempio mostrato qui sotto.
- (SubTask 2 - 16 punti) Nelle istanze di questo subtask si ha che $M \ \ N \ 2$.
- (SubTask 3 - 22 punti) Nelle istanze di questo subtask si ha che ciascuna nave e ciascun messaggio appare al più 2 volte nella lista delle possibili corrispondenze (quindi, $M \leq 2N$).
- (SubTask 4 - 27 punti) Nelle istanze di questo subtask $N \leq 3.000$, $M \leq 5.000$
- (SubTask 5 - 30 punti) Nelle istanze di questo subtask non ci sono vincoli particolari.

2.6 Esempi di input/output

File `input.txt`	**File** `output.txt`
3 6 0 2 1 0 2 1 0 0 1 2 2 2	0 0 1 2 2 1

2.7 Codice della soluzione

Nota: la soluzione usa l'algoritmo della DFS, o "ricerca in profondità".

```
#include <cstdio>
#include <vector>
using namespace std;
/* Lista di adiacenza del grafo (orientato) bipartito.
 * I vertici da 0 a N-1 corrispondono ai messaggi, mentre i vertici
 * da N a 2*N-1 corrispondono alle navi.
 */
vector<int> graph[200000];
int N, M;
/* Array che associa al nodo i del grafo il nodo che lo precede nella
 * DFS. In particolare viene salvato "id del padre"+1, in modo che il
 * valore associato alla cella i sia 0 se e solo se la cella non e' mai
 * stata visitata.
 */
int daddies[200000];
/* Array che associa a un nodo "1" se il nodo e' attualmente un nodo aperto
 * dalla DFS, "0" altrimenti.
 */
bool isancestor[200000];
/* Vettore usato per contenere l'eventuale ciclo trovato.
 */
vector<int> cycle;
/* Array con le corrispondenze messaggio -> nave del nuovo matching.
 */
int new_match[100000];
/* Funzione che cerca un ciclo. Per trovare un ciclo, viene effettuata una
 * ricerca in profondita' e se in questa ricerca si trova un back-edge, cioe'
 * un arco che porta a un nodo attualmente aperto, significa che si e'
 * formato un ciclo. Restituisce true se trova un ciclo.
 */
int dfs(int n,int d){
    // Imposta il padre del nodo al nodo precedente.
    daddies[n] = d+1;
    // Apre il nodo
    isancestor[n] = true;
    // Visita tutti i nodi figli
    for(unsigned i=0; i<graph[n].size(); i++){
        // Se il nodo figlio e' aperto, ho trovato un ciclo...
        if(isancestor[graph[n][i]]){
            // Aggiunge il figlio al ciclo
            cycle.push_back(graph[n][i]);
            int t = n;
            // E anche il nodo corrente
            cycle.push_back(t);
            /* Risalgo gli archi della DFS fino a quando non
             * arrivo al nodo figlio, inserendo ogni volta il
             * nodo nel ciclo.
             */
            while(t != graph[n][i]){
                t = daddies[t]-1;
                cycle.push_back(t);
            }
```

```
53              // Ho trovato un ciclo: restituisco true.
54              return true;
55          }
56          /* Se il nodo figlio e' stato visitato ma non e' aperto,
57           * e' un nodo gia' chiuso, dunque lo salto.
58           */
59          if(daddies[graph[n][i]]) continue;
60          /* Se la visita del nodo figlio trova un ciclo, restituisco
61           * true in quanto la DFS a partire dal nodo corrente trova
62           * un ciclo.
63           */
64          if(dfs(graph[n][i], n)) return true;
65      }
66      // Chiude il nodo.
67      isancestor[n]=false;
68      // Se arrivo fin qua, non ho trovato nulla: restituisco false.
69      return false;
70  }
71  int main(){
72  #ifdef EVAL
73      freopen("input.txt", "r", stdin);
74      freopen("output.txt", "w", stdout);
75  #endif
76      // Legge N,M.
77      scanf("%d%d", &N, &M);
78      // Legge tutti gli archi.
79      for(int i=0; i<M; i++){
80          int a, b;
81          scanf("%d%d", &a, &b);
82          /* Il messaggio a corrisponde al nodo a, la nave b
83           * invece al nodo b+N.
84           */
85          b += N;
86          /* Se sto leggendo gli archi da 0 a N-1, sono gli archi nel
87           * matching corrente, dunque creo l'arco messaggio -> nave.
88           * Altrimenti sono gli altri archi, e creo il collegamento
89           * nave -> messaggio.
90           */
91          if(i<N) graph[a].push_back(b);
92          else graph[b].push_back(a);
93      }
94      // Per ogni nodo...
95      for(int i=0; i<N; i++){
96          /* Se il nodo corrente e' stato visitato da qualche DFS
97           * precedente, lo salto.
98           */
99          while(i<N && daddies[i]) i++;
100         /* Se non ho gia' finito tutti i nodi, effettuo una DFS a
101          * partire dal nodo corrente. Se trovo un ciclo, creo il
102          * nuovo matching, lo stampo ed esco.
103          */
104         if(i<N && dfs(i,i)){
105             /* Per ogni arco messaggio -> nave nel ciclo,
106              * associo quel messaggio a quella nave.
107              */
```

```
108             for(unsigned i=1; i<cycle.size(); i++)
109                 if(cycle[i-1]<N)
110                     new_match[cycle[i-1]] = cycle[i];
111             /* Ora completo il matching, aggiungendo la
112              *
113              */
114             for(int i=0; i<N; i++)
115                 if(!new_match[i])
116                     new_match[i] = graph[i][0];
117             for(int i=0; i<N; i++)
118                 printf("%d %d\n", i, new_match[i]-N);
119             return 0;
120         }
121     }
122     // Se non ho trovato cicli, stampo -1 e esco.
123     printf("-1\n");
124     return 0;
125 }
```

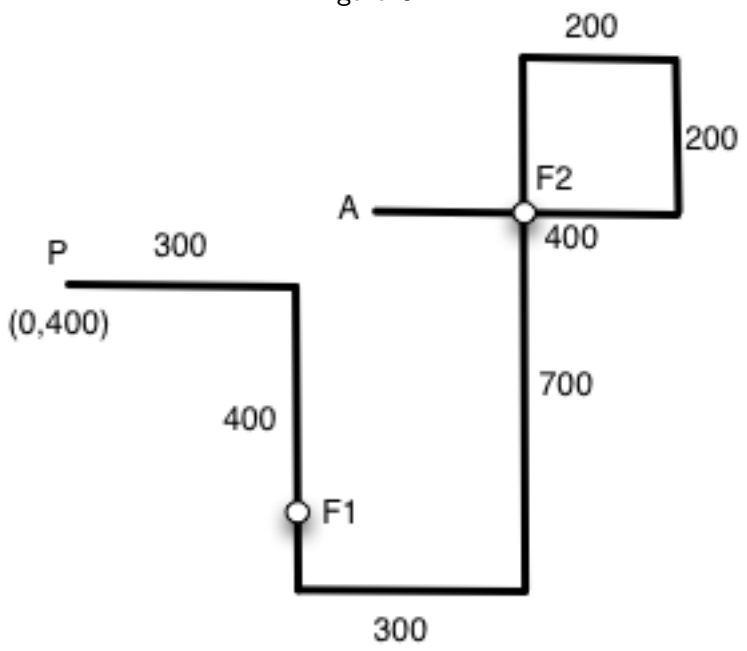
3 Allenamento per la maratona (`fontane`)

3.1 Descrizione del problema

Nota storica: in pochi sanno che Turing era un patito maratoneta, a tal punto che il suo record personale, ottenuto il 25 agosto del 1947, 2 ore e 46 minuti e 3 secondi, è stato di soli 11 minuti superiore a quello del vincitore delle Olimpiadi del 1948 (l'argentino Delfo Cabrera, che vinse in 2 ore, 34 minuti e 51 secondi).

Alan Turing si vuole allenare per la maratona. Il suo problema è quello di rifornirsi d'acqua. Ha una mappa piuttosto accurata della zona, con segnate tutte le fontanelle disponibili, e sulla quale ha riportato il percorso che intende fare. Ha scelto un percorso formato solo da tratti in direzione orizzontale (Est-Ovest) o verticale (Nord-Sud). Turing, per semplicità, consuma 1ml di acqua per ogni metro che corre: dopo aver bevuto 100ml, per esempio, è in grado di correre per 100 metri. Turing però non vuole bere mai più di 100ml per volta, e vuole correre senza essere appesantito: quindi, vuole portarsi appresso una borraccia più piccola possibile. Data la mappa con segnate le fontanelle, aiutate Turing a capire qual'è la capacità della più piccola borraccia che gli consente di correre avendo sempre acqua a sufficienza.

Figura 3:



Considerate l'esempio mostrato in figura, dove l'origine degli assi $0, 0$ è in basso a sinistra: qui Turing parte dal punto di coordinate $0, 400$ (marcato da una P) e corre lungo 7 tratti lunghi, in ordine, rispettivamente 300, 400, 300, 700, 200, 200 e 400 metri. Ci sono due fontanelle nel percorso, la prima (marcata come F1) nel punto di coordinate $300, 100$ e la seconda (marcata come F2) nel punto di coordinate $600, 500$. Per questo percorso, Turing ha bisogno di una borraccia da 800 ml: infatti, partendo con la borraccia piena, incontra la prima fontanella dopo 600 metri. Qui Turing beve (100ml), e riempie la borraccia (800ml), cosa che gli fornisce l'autonomia per raggiungere la seconda fontanella, che dista 900m nel percorso da lui seguito. A questo punto, seguendo il suo percorso, passa nuovamente per la seconda fontanella dopo 800m, e da qui gli mancano solo 200m per l'arrivo. Come si vede, una borraccia da 800ml gli è sufficiente per potersi allenare in questo percorso.

Si assume che Turing parte sempre con la borraccia e la pancia piena. Nota bene: Turing, oltre a riempire la borraccia, quando arriva a una fontanella può bere e, in ogni istante, Turing può avere al massimo 100ml in pancia: per esempio, se beve 100ml a una fontana e dopo 20 metri incontra un'altra fontana, a questa può bere solo 20ml.

3.2 Dati di input

Il file di input consiste di $N\ M\ 2$ righe. La prima riga contiene due interi N ed M, rispettivamente il numero di tratti in cui il suo percorso è suddiviso, e il numero di fontanelle presenti nella zona.

Le successive $N\ 1$ righe contengono ciascuna due interi X_i, Y_i, le coordinate (in metri) dell'i-esimo vertice del percorso di Turing.

Le ultime M righe contengono ciascuna due interi S_x, S_y, le coordinate (in metri) delle fontanelle.

L'istanza mostrata nel file di esempio si riferisce a quella mostrata in figura.

3.3 Dati di output

Il file di output consiste di un unica riga contente un unico intero T: la capacità in ml della borraccia che Turing dovrà comprare per poter completare il suo percorso utilizzando solo i rifornimenti lungo di esso.

3.4 Assunzioni

- $1 \leq N, M \leq 100000$
- $0 \leq X_i, Y_i, S_x, S_y \leq 10^9$
- $0 \leq T \leq 10^9$

3.5 Valutazione delle soluzioni

- (SubTask 1 - 5 punti) Questo subtask è costituito da una sola istanza: il caso di esempio mostrato qui sotto.
- (SubTask 2 - 17 punti) Nelle istanze di questo subtask il percorso di Turing è su una linea retta e $N\ 1$.
- (SubTask 3 - 15 punti) Nelle istanze di questo subtask $N \leq 100$ e $0 \leq X_i, Y_i, S_x, S_y \leq 1000$.
- (SubTask 4 - 48 punti) Nelle istanze di questo subtask il percorso di Turing passa al massimo 5 volte su ciascuna delle fontane.
- (SubTask 5 - 7 punti) Nelle istanze di questo subtask il percorso di Turing è su una linea retta.
- (SubTask 6 - 8 punti) Nelle istanze di questo subtask non ci sono vincoli particolari.

3.6 Esempi di input/output

File `input.txt`	**File** `output.txt`
7 2 0 400 300 400 300 0 600 0 600 700 800 700 800 500 400 500 300 100 600 500	800

3.7 Nota/e

- Non tutte le fontanelle sono sul percorso seguito da Turing; non esistono due fontane nella stessa posizione (ovvero, con le stesse coordinate).
- Il tratto tra due vertici consecutivi del percorso di Turing è sempre in orizzontale o in verticale.
- Il percorso di Turing potrebbe utilizzare più di una volta la stessa fontanella.
- Si assume che Turing parta con la borraccia piena (e la pancia piena: 100ml).
- Ci possono essere tratti consecutivi che sono paralleli (sia sovrapposti che con lo stesso verso).

3.8 Codice della soluzione

Nota: nella soluzione sono usati gli algoritmi della ricerca binaria e della RMQ, o "range minimum query".

```
#include <cstdio>
#include <algorithm>
#include <cstdlib>
using namespace std;
// Struttura per contenere una coppia di coordinate (x,y)
struct point{
    int x, y;
};
/* f[0] contiene le fontane ordinate prima per la coordinata x e poi
 * per la y, f[1] invece le fontane ordinate prima per y e poi per x.
 */
point f[2][100000];
// Elenco dei vertici toccati dal percorso di Turing
point vt[100001];
/* rmq[0] contiene le informazioni per effettuare una range maximum query
 * in tempo costante sulle fontane ordinate come in f[0]. Analogamente
 * rmq[1] contiene le informazioni relative alle fontane ordinate come in
 * f[1].
 */
int rmq[2][32][100000];
// Array che contiene il logaritmo in base 2 dell'i-esima posizione.
int off[100000];

int N, M;
/* Valore della distanza percorsa dall'ultima fontana e valore della
 * massima lunghezza percorsa finora senza trovare una fontana.
 */
int len, maxlen;
// Funzione che restituisce il numero di metri da percorrere tra due punti.
inline int dst(point a, point b){
    if(a.y==b.y || a.x==b.x)
        return abs(a.y-b.y) + abs(a.x-b.x);
    return 0;
}
// Inizializzazione delle strutture per l'RMQ O(n log n)
void init_rmq(){
    /* Lo strato 0 dell'RMQ contiene semplicemente le distanze tra
     * l'i-esima fontana e l'i+1-esima.
     */
    for(int i=1; i<M; i++){
        rmq[0][0][i-1] = dst(f[0][i-1],f[0][i]);
        rmq[1][0][i-1] = dst(f[1][i-1],f[1][i]);
    }
    /* Inizia a costruire gli strati successivi, partendo dallo strato
     * j=1 e proseguendo finche' 2^j e' minore di M.
     */
    for(int j=1; (1<<j)<M; j++){
        /* Calcola il numero di distanze di cui si occupa una cella
         * nello strato precedente, ovvero 2^(j-1)
         */
        int l = 1<<(j-1);
```

```
52          /* La cella i dello strato j si occupa delle distanze nel
53           * range [i,i+2*l], dunque i valori possibili di i sono da
54           * 0 a M-l-1.
55           */
56          for(int i=0; i<M-l; i++){
57              /* Il valore massimo nel range [i,i+l] e' pari al
58               * massimo tra i massimi di [i,i+l] e di [i+l,i+2*l]
59               */
60              rmq[0][j][i] = max(rmq[0][j-1][i],rmq[0][j-1][i+l]);
61              rmq[1][j][i] = max(rmq[1][j-1][i],rmq[1][j-1][i+l]);
62          }
63      }
64      // Costruisce l'array dei logaritmi in base 2
65      int p = 0;
66      for(int i=1; i<M; i++){
67          if (2<<p <= i) p++;
68          off[i] = p;
69      }
70  }
71  /* Funzione che fa l'RMQ sulle fontane ordinate prima per x e poi per y.
72   * Se sto chiedendo la distanza tra due fontane uguali, restituisco 0.
73   * Se chiedo quella tra fontane consecutive, restituisco quella presente
74   * nel primo strato dell'RMQ. Altrimenti, usando l'array dei logaritmi
75   * in base due ottengo lo strato corrispondente alla richiesta corrente
76   * e restituisco il massimo tra il massimo di [a,a+l] e quello di [b-l,b].
77   */
78  inline int rmq_x(int a,int b){
79      if(a==b) return 0;
80      if(a==b-1) return rmq[0][0][a];
81      int offset = off[b-a-1];
82      return max(rmq[0][offset][a], rmq[0][offset][b-(1<<offset)]);
83  }
84  // Come rmq_x, ma per le fontane ordinate prima per y e poi per x.
85  inline int rmq_y(int a,int b){
86      if(a==b) return 0;
87      if(a==b-1) return rmq[1][0][a];
88      int offset = off[b-a-1];
89      return max(rmq[1][offset][a], rmq[1][offset][b-(1<<offset)]);
90  }
91  // Confronta due fontane, prima per y e poi per x.
92  inline bool minY(const point &a, const point &b){
93      return (a.y<b.y) || (b.y==a.y && a.x<b.x);
94  }
95  // Confronta due fontane, prima per x e poi per y.
96  inline bool minX(const point &a, const point &b){
97      return (a.x<b.x) || (b.x==a.x && a.y<b.y);
98  }
99  /* Calcola la massima distanza tra due fontane nel caso in cui il
100  * tratto percorso sia verticale.
101  */
102 void doX(point b, point e){
103     // low contiene il punto piu' in basso, up quello piu' in alto
104     point low = minX(b,e)?b:e;
105     point up = minX(b,e)?e:b;
106     // l e u invece contengono la prima e l'ultima fontana del tratto
```

```
107     point* l = lower_bound(f[0], f[0]+M, low, minX);
108     point* u = upper_bound(f[0], f[0]+M, up, minX);
109     u--;
110     // Se non ho trovato nessuna fontana nel percorso corrente...
111     if(l==f[0]+M || l->x>low.x || u<l){
112         // Aumento la distanza percorsa senza incontrare fontane
113         len += dst(low, up);
114         return;
115     }
116     /* Altrimenti, trovo la massima distanza tra la prima e l'ultima
117      * fontana del tratto corrente.
118      */
119     int t = rmq_x(l-f[0], u-f[0]);
120     /* Un altro tratto da considerare e' quello tra l'ultima fontana
121      * incontrata e la prima del tratto attuale.
122      */
123     int a;
124     // Se sto percorrendo il tratto dal basso in alto (b<e)...
125     if(minX(b, e)){
126         /* La distanza dall'ultima fontana incontrata alla prima
127          * del tratto attuale vale len piu' la distanza tra
128          * il primo estremo del tratto e la prima fontana.
129          */
130         a = len+dst(b, *l);
131         /* La nuova distanza percorsa senza fontane e' pari alla
132          * distanza tra l'ultima fontana e la fine del tratto.
133          */
134         len = dst(e, *u);
135     }
136     else{
137         // Altrimenti, analogamente a prima.
138         a = len+dst(b, *u);
139         len = dst(e, *l);
140     }
141     /* I due candidati per la nuova massima lunghezza sono la massima
142      * distanza tra due fontane del tratto corrente (t) e la distanza
143      * tra l'ultima fontana incontrata e la prima di questo tratto (a)
144      */
145     if(t>maxlen) maxlen = t;
146     if(a>maxlen) maxlen = a;
147 }
148 // Analogamente a doX, ma nel caso in cui il tratto sia orizzontale.
149 void doY(point b, point e){
150     point low = minY(b,e)?b:e;
151     point up = minY(b,e)?e:b;
152     point* l = lower_bound(f[1], f[1]+M, low, minY);
153     point* u = upper_bound(f[1], f[1]+M, up, minY);
154     u--;
155     if(l==f[1]+M || l->y>low.y || u<l){
156         len += dst(low, up);
157         return;
158     }
159     int t = rmq_y(l-f[1], u-f[1]);
160     int a;
161     if(minX(b, e)){
```

```
162         a = len+dst(b, *l);
163         len = dst(e, *u);
164     }
165     else{
166         a = len+dst(b, *u);
167         len = dst(e, *l);
168     }
169     if(t>maxlen) maxlen = t;
170     if(a>maxlen) maxlen = a;
171 }
172 int main(){
173 #ifdef EVAL
174     freopen("input.txt","r",stdin);
175     freopen("output.txt","w",stdout);
176 #endif
177     // Lettura dell'input
178     scanf("%d%d", &N, &M);
179     for(int i=0; i<=N; i++) scanf("%d%d", &vt[i].x, &vt[i].y);
180     for(int i=0; i<M; i++) scanf("%d%d", &f[0][i].x, &f[0][i].y);
181     // Copia in f[1] le fontane salvate in f[0]
182     for(int i=0; i<M; i++) f[1][i].x = f[0][i].x;
183     for(int i=0; i<M; i++) f[1][i].y = f[0][i].y;
184     // Ordina le fontane in f[0] e in f[1]
185     sort(f[0], f[0]+M, minX);
186     sort(f[1], f[1]+M, minY);
187     // Inizializza la range maximum query
188     init_rmq();
189     // Per ogni tratto nel percorso...
190     for(int i=0; i<N; i++){
191         // Se e' orizzontale, eseguo doX
192         if(vt[i].x == vt[i+1].x)
193             doX(vt[i], vt[i+1]);
194         // Altrimenti eseguo doY
195         else doY(vt[i], vt[i+1]);
196     }
197     /* Se la distanza massima tra due fontane e' minore di
198      * 100, allora non serve nessuna borraccia e stampo 0.
199      * Altrimenti stampo (distanza max)-100.
200      */
201     printf("%d\n", (maxlen>100)?(maxlen-100):0);
202 }
```