Soluzione problema "Escursioni" - OII - Selezioni Territoriali 2018

Mojito vuole pianificare un'escursione sulle colline di Volterra. Ha a disposizione una mappa rettangolare, in cui è indicata l'altitudine della zona. Mojito vuole fare un percorso che parte dall'angolo in alto a sinistra della mappa e raggiunge l'angolo in basso a destra, in modo tale che il dislivello massimo che è costretto a fare ad ogni spostamento sia il mimimo possibile. Aiuta Mojito a calcolare questo dislivello!

Dettagli

La mappa è una tabella di numeri interi: ciascuno esprime l'altitudine in metri nel corrispondente punto della mappa. La tabella è composta di H righe e W colonne, numerate rispettivamente da 1 a H e da 1 a W. Nella cella di coordinate (i,j), ovvero in corrispondenza della riga i e della colonna j, è indicato il valore dell'altitudine $A_{i,j}$.

Mojito inizia l'escursione dalla cella di coordinate (1,1), in alto a sinistra, ed arriva alla cella di coordinate (H,W), in basso a destra. Ogni minuto si sposta di esattamente una cella, in una della quattro possibili direzioni (in alto, in basso, a destra o a sinistra). Non può però uscire dalla mappa.

Stabilito un percorso lungo la mappa, il pericolo associato a quel percorso è il massimo dislivello tra due celle consecutive lungo il percorso, ovvero la differenza di altitudine fra due celle consecutive: non cambia nulla se lo spostamento è in salita o in discesa.

Calcola il pericolo minimo, fra tutti i percorsi possibili che partono dalla cella (1,1) e arrivano alla cella (H,W).

Assunzioni

- T=27, ci sono 27 casi di prova.
- 1≤H,W≤100, la mappa ha dimensione massima 100×100.
- (1,1)≠(H,W, ovvero la mappa è abbastanza grande da avere partenza e arrivo in punti diversi.
- 1≤Ai,j≤1.000.000, l'altitudine in ogni cella è compresa fra 1 e 1.000.000.

Dati di input

La prima riga del file di input contiene un intero T, il numero di casi di test. Seguono T casi di test, numerati da 1 a T. Ogni caso di test è preceduto da una riga vuota.

In ciascun caso di test, la prima riga contiene due interi H e W separati da uno spazio che corrispondono all'altezza, H, e alla larghezza, W, della mappa. Le successive H righe contengono ciascuna W interi separati da spazi, corrispondenti all'altitudine in metri lungo una riga della mappa. Ovvero, in ciascun caso di test, l'altitudine Ai,j alle coordinate i e j appare sulla riga (i+1)-esima, in posizione j.

Dati di output

Il file di output deve contenere la risposta ai casi di test che sei riuscito a risolvere. Per ogni caso di test che hai risolto, il file di output deve contenere una riga con la dicitura
Case #t: p
dove t è il numero del caso di test (a partire da 1) e p è il minimo valore di pericolo trovato per quel test case.

Esempi di input/output

Input:
3

2 2
100 150
110 130

4 4
1 5 6 7
2 4 3 8
2 9 2 8
3 3 2 9

1 10
2 4 6 8 10 12 14 16 18 20

Output:
Case #1: 20
Case #2: 1
Case #3: 2

Spiegazione

Nel primo caso d'esempio, Mojito sceglie il percorso:

```
100   150
 ▼
110 ► 130
```

ovvero, con i seguenti spostamenti:

- in basso, da (1,1) a (2,1), con un dislivello pari a 110−100=10
- a destra, da (2,1) a (2,2)=(H,W), con un dislivello pari a 130−110=20.

Il pericolo del percorso è 20 (il massimo fra i dislivelli, 10 e 20).
Non ci sono percorsi migliori, quindi la risposta corretta è 20. L'altro percorso possibile è:

```
100 ► 150
       ▼
110   130
```

che ha dislivelli 50 e 20, e quindi ha pericolo 50.

Nel secondo caso d'esempio, Mojito sceglie il percorso:

```
1   5 ► 6 ► 7
▼   ▲       ▼
2   4 ◄ 3   8
▼       ▲   ▼
2   9   2   8
▼       ▲   ▼
3 ► 3 ► 2   9
```

Gli spostamenti hanno tutti dislivello 0 o 1, quindi il pericolo del percorso è 1. Non ci sono percorsi di pericolo pari a 0, quindi la risposta corretta è 1.

Nel terzo caso d'esempio c'è un solo percorso possibile.

Considerazioni generali e scelta della strategia risolutiva.

Una possibile tecnica risolutiva è rappresentata dal backtracking: si parte dal nodo (1,1) e si esplora tutti i possibili cammini per giungere al nodo (H,W), muovendosi ad ogni passo nelle 4 direzioni possibili; l'ordine dei 4 spostamenti è irrilevante purché si effettuino sempre tutti quelli possibili, ovvero senza uscire dalla mappa, e sempre nello stesso ordine scelto, naturalmente.
Nel caso di test delle figure seguenti, ipotizzando che da ogni cella ci si muova prima verso il basso, finché è possibile, poi verso destra, poi verso l'alto e infine a sinistra, i primi 2 cammini esplorati sarebbero i seguenti:

Il backtracking comporta che, giunti alla cella finale (H,W) (oppure ad una cella che, come vedremo, rende il percorso inutile) si torni indietro provando a sostituire l'ultimo spostamento effettuato con uno alternativo; se non vi sono alternative si torna ancora indietro fino al primo spostamento alternativo possibile da effettuare che ci consente di esplorare un altro cammino che conduca sempre a (H,W).
Per esempio, al termine del primo cammino, si torna indietro e la mossa da (3,3) a (3,4) è sostituita dalla mossa da (3,3) a (2,3) che dà seguito all'esplorazione del secondo cammino.
Scegliendo questo approccio occorre però fare i conti con la sua complessità: ad ogni invocazione della funzione ricorsiva vanno considerati 4 possibili spostamenti che danno origine ciascuno ad una nuova chiamata ricorsiva, ovvero nel caso peggiore 4x4…x4, tante volte quanti sono gli elementi della matrice, quindi $\mathbf{4^{H*W}}$ ricorsioni, per cui si ha una complessità esponenziale.
Le ricorsioni si possono certamente limitare memorizzando in una nuova matrice (che chiamiamo "dislivelli") il dislivello massimo trovato per ciascuna cella (r,c) della matrice, lungo il cammino percorso dalla prima cella (1,1) fino alla stessa cella (r,c).
Ogni volta che l'esplorazione di un cammino giunge ad una cella (r2,c2) e presenta un nuovo dislivello calcolato da (1,1) a (r2,c2) che non migliora il valore precedentemente rilevato e memorizzato nella matrice "dislivelli" per la cella (r2,c2), allora si evita di proseguire l'esplorazione del cammino e si passa ad esaminarne un altro.
Ciò consente di evitare anche i cicli, ovvero il ritorno su una cella (r,c) precedentemente visitata nel cammino in corso di esplorazione, perché il nuovo dislivello calcolato sul prolungamento del percorso sarà al più uguale a quello precedentemente memorizzato per (r,c).
Al termine delle esplorazioni dei cammini possibili, il dislivello corrispondente alla cella (H,W) sarà la soluzione al problema.
Nell'esempio precedente la matrice dislivelli, inizializzata con il valore massimo possibile per tutte le celle, serve a valutare passo dopo passo se proseguire il cammino in esame o arrestarlo.
Si supponga di spostarsi da una cella (r1,c1) a una cella adiacente (r2,c2): si noti che, per essere le due celle adiacenti, una delle due coordinate r2 o c2 coinciderà con r1 o c1, perché ci si sposta o sulla stessa riga o sulla stessa colonna.
Nel cammino in esame, se X è il dislivello finora calcolato per (r1,c1) e D è la differenza di livello tra (r1,c1) e (r2,c2) allora occorrerà aggiornare il dislivello di (r2,c2) con il valore Y=max(X,D) perché per ogni cammino conta lo spostamento con il massimo dislivello. Ma tale aggiornamento ha senso farlo solo se risulta migliorativo rispetto al valore già presente per (r2,c2) nella matrice dislivelli.

Se è la prima volta che si giunge in (r2,c2), il suo corrispondente valore nella matrice dislivelli sarà +∞, se invece è già stata visitata sarà un valore Z.

Il cammino in esame, dunque, proseguirà solo se Y<Z, perché solo in tale evenienza il suo dislivello finale potrebbe risultare migliore rispetto a Z. Se viceversa Y>=Z e si proseguisse nel cammino aggiornando il valore Z con Y, il dislivello finale potrebbe solo peggiorare rispetto a Z e dunque meglio lasciare inalterato il miglior valore Z e arrestare il cammino per valutare uno spostamento alternativo, a ritroso, sull'ultima cella visitata su cui è possibile cambiare mossa.

Ritornando all’esempio precedente, dopo aver percorso i primi 2 cammini, la matrice “dislivelli” assumerà i seguenti valori:

	1	2	3	4
1	∞	∞	∞	∞
2	3	∞	∞	∞
3	6	6	6	6

	1	2	3	4
1	∞	∞	∞	∞
2	3	∞	6	7
3	6	6	6	6

Ma tutto ciò non è sufficiente a garantire tempi di esecuzione apprezzabili sui casi di test più stressanti, ovvero con matrice di dimensione 100x100.

Occorre infatti fare una considerazione importante che ci introduce ad un’ulteriore miglioramento dell’algoritmo ipotizzato, ovvero anche ad un altro approccio risolutivo più efficiente.

Ci sono infatti esplorazioni inutili che si possono evitare anche quando il nuovo dislivello trovato è inferiore al valore precedentemente individuato e memorizzato in corrispondenza della cella.

Per scoprirne la ragione, esaminiamo il terzo percorso da esplorare, ovvero, dopo essere ritornati indietro dalla cella finale (3,4) nella cella (2,4), non potendo spostarsi verso destra si va verso l’alto in (1,4):

	1	2	3	4
1	1	3	6	14
2	4	5	7	14
3	10	5	8	9

dove si trova come dislivello +∞, non avendo finora visitato tale cella, e dunque si migliorerebbe tale valore da +∞ a 7.

Così avverrebbe per la cella successiva (1,3) ma l’esplorazione sta proseguendo inutilmente in un vicolo cieco, poiché non ci sarà modo di arrivare alla cella finale (3,4) senza incrociare una cella precedentemente visitata, che avrà un dislivello precedentemente calcolato al più uguale al nuovo valore che le si vorrebbe attribuire e ciò comporterà l’interruzione del cammino in esame.

Inoltre i dislivelli che attribuiremmo alle celle (1,4) e (1,3) non sarebbero attendibili visto che il cammino in esame incrocerà una cella visitata e verrà scartato prima di raggiungere la cella di destinazione (H,W).

Per queste ragioni tali cammini vanno esclusi a priori dall’analisi.

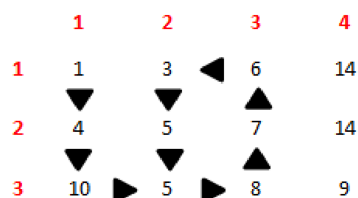
Il successivo percorso da esaminare, pertanto, dovrà essere il seguente:

	1	2	3	4
1	1	3	6	14
2	4	5	7	14
3	10	5	8	9

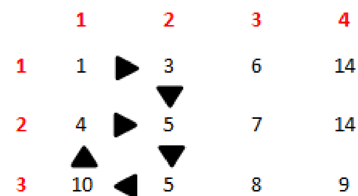
a seguito del quale la matrice dislivelli diventerà:

	1	2	3	4
1	∞	∞	6	8
2	3	∞	6	7
3	6	6	6	6

Altri esempi, invece, di percorsi che conducono a vicoli ciechi sono i seguenti:



e dopo essere ritornati alla cella di partenza (1,1):

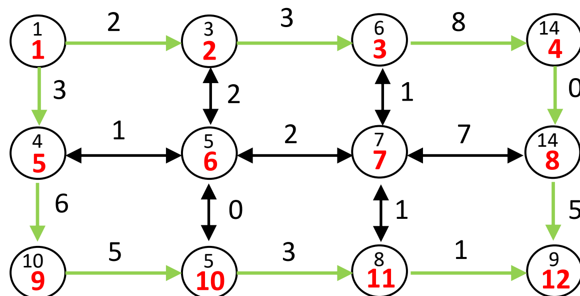


Tali percorsi inutili da esplorare possono avere un'incidenza non trascurabile dal punto di vista computazionale con matrici di grandi dimensioni, tenendo presente che ad ogni passo i cammini da esplorare aumentano esponenzialmente.

Ad una più attenta osservazione degli esempi visti si può affermare che, generalizzando, il percorso si infila in un vicolo cieco quando ci si sposta lungo il perimetro della matrice nelle seguenti direzioni: verso sinistra sulla prima e sull'ultima riga, verso l'alto sulla prima e sull'ultima colonna.

Quindi basterà evitare questi spostamenti.

In altre parole, gli spostamenti lungo le celle perimetrali della matrice, evidenziati in verde nella figura seguente, sono utili se avvengono esclusivamente nell'unica direzione indicata, perché mai servirà, ai fini del problema, percorrerli in senso inverso:



Gli spostamenti che coinvolgono invece una cella interna alla matrice, cioè non posizionata sul perimetro, sono da esplorare in entrambe le direzioni di percorrenza.

Soluzione efficiente.

Pertanto, la figura precedente introduce di fatto un nuovo approccio risolutivo, ovvero un grafo orientato in cui i nodi sono le celle della matrice, gli archi sono i possibili spostamenti ai fini del problema, il peso di ciascun arco non è altro che il valore assoluto della differenza tra i livelli dei 2 nodi collegati dall'arco.

Sarà sufficiente visitare il grafo con una variante dell'algoritmo di Dijkstra che consideri, per ogni percorso, il dislivello massimo (pericolo) anziché la distanza complessiva del percorso.

Occorre individuare il percorso da (1,1) a (H,W) con il pericolo minimo.

Di seguito si riporta un esempio di implementazione dell'algoritmo, con cui si superano tutti i 27 casi di test previsti.

I nodi sono N=H*W, numerati da 1 ad N: nella figura sono scritti in rosso, mentre i livelli in nero.

Gli archi orizzontali sono W-1 per ogni riga, dunque in tutto H*(W-1) e analogamente gli archi verticali sono (H-1)*W.

Le istruzioni da 48 a 56 definiscono gli archi orizzontali: per definire i nodi da 1 a N numerati progressivamente (in rosso) come in figura, dato l'elemento (i,j) della matrice il corrispondente nodo x dalla formula x=(i-1)*W+j.

Gli archi orizzontali sulle righe interne sono bidirezionali, mentre quelli sulla prima e ultima riga sono unidirezionali.

Analogamente si definiscono gli archi verticali con le istruzioni da 57 a 65: in questo caso si è preferito prima iterare sulle colonne poi sulle righe.

L'algoritmo di Dijkstra è stato implementato con una priority queue che si differenzia dalla coda normale perché ha la caratteristica di porre in testa alla coda e rendere immediatamente estraibile l'elemento di valore più grande inserito nella coda stessa.

Come esempio di implementazione dell'algoritmo di Dijkstra mediante priority queue, si veda la seguente dispensa.
Poiché ad ogni passo dell'algoritmo di Dijkstra occorre scegliere tra i nodi da visitare quello con distanza minima (nel nostro problema quello con il più piccolo dislivello massimo calcolato, cioè il più piccolo pericolo) allora è sufficiente inserire nella priority queue la coppia **<-pericolo,nodo>** (cioè con il pericolo in negativo) perché così facendo il più piccolo valore tra i dislivelli (pericoli), assieme al suo nodo, verrà posto in testa alla coda perché in negativo risulterà essere il maggiore valore esistente nella priority queue.
Si noti che:

1. come mostrato prima, i nodi, anziché essere rappresentati dalla coppia (i,j) con le coordinate riga e colonna, sono numeri interi da 1 ad N, così da non modificare le strutture dati previste dall'algoritmo di Dijkstra "standard" implementato con la priority queue;
2. i booleani processed[] si accendono quando il nodo corrispondente viene visitato;
3. i valori peric[] memorizzano i pericoli calcolati per ciascun nodo: la soluzione del problema dunque sarà peric[N];
4. Non è necessario, ai fini del problema, memorizzare il nodo di provenienza per ciascun nodo visitato;
5. L'algoritmo di Dijkstra è ottimizzato dall'arresto (istruzione 21) quando il nodo finale passa tra i nodi visitati e dunque la sua etichetta diventa definitiva perché il suo pericolo non è più migliorabile; in tal modo si evita di analizzare tutti gli eventuali percorsi non ancora esplorati ma inutili, perché non potrebbero migliorare ulteriormente il pericolo calcolato sull'ultimo nodo, risparmiando tempo di esecuzione.

Codice sorgente

```
1  #include <bits/stdc++.h>
2  #define MAXH 101
3  #define MAXN 101*101
4  #define MAXM 101*100+100*101
5  using namespace std;

6  int A[MAXH][MAXH];
7  int H, W, N, M, INIZIO, FINE;
8  bool processed[MAXN];
9  vector <pair<int,int>> adi[MAXM]; // adiacenze
10 int peric[MAXN]; // pericoli totali

11 int calcola() {
       // algoritmo di Dijkstra
       // Inizializzazione (passo 1)
12     fill(processed,processed+MAXN,false);
13     for (int i = 1; i <= N; i++) peric[i] = INT_MAX;
14     peric[INIZIO] = 0;
15     priority_queue <pair<int, int>> q;
16     q.push({0,INIZIO});
       // ciclo (passi 2 e 3 ripetuti)
17     while (!q.empty()) {
18         int a = q.top().second; q.pop();
19         if (processed[a]) continue;
20         processed[a] = true; // Assegnazione etichetta definitiva
21         if (a==FINE) break; // Ottimizzazione: quando il nodo FINE e' processato
22         for (pair<int,int> u : adi[a]) {
23             int b = u.first, w = u.second;
24             int newMax=max(peric[a],w);
25             if (newMax < peric[b]) { // Assegnazione etichetta provvisoria
26                 peric[b] = newMax; // pericolo totale migliore (finora) fino al nodo b
27                 q.push({-peric[b],b}); // N.B. top() per la priority_queue è il max -->
                                          // il pericolo in negativo per avere in testa il min!
28             }
29         }
30     }
31     return peric[FINE];
32 }
33 int main() {
34     fstream fin("input.txt",fstream::in);
35     fstream fout("output.txt",fstream::out);
36     int T;
37     fin >> T;
```

```
38 	for (int t=1; t<=T; t++) {
39 		fin >> H >> W;
40 		for (int i=1; i<=H; i++)
41 			for (int j=1; j<=W; j++)
42 				fin >> A[i][j];
		// Costruzione grafo basato sulla matrice A
43 		for (int i=0; i<MAXM; i++) adi[i].clear(); // azzeramento archi per ogni caso-test
44 		N=H*W; // Numero nodi
45 		M=H*(W-1)+W*(H-1); // Numero archi
46 		INIZIO=1; FINE=H*W;
47 		int da,a,p;
		// archi orizzontali
48 		for (int i=1; i<=H; i++) // N.B. Prima ciclo su righe
49 			for (int j=1; j<W; j++) {
50 				da=(i-1)*W+j;
51 				a=(i-1)*W+j+1;
52 				p=abs(A[i][j]-A[i][j+1]);
53 				adi[da].push_back({a,p});
54 				if (i>1 && i<H)
55 					adi[a].push_back({da,p});// arco bidirez. (non su righe 1 e H)
56 			}
		// archi verticali
57 		for (int j=1; j<=W; j++) // N.B. Prima ciclo su colonne
58 			for (int i=1; i<H; i++) {
59 				da=(i-1)*W+j;
60 				a=i*W+j;
61 				p=abs(A[i][j]-A[i+1][j]);
62 				adi[da].push_back({a,p});
63 				if (j>1 && j<W)
64 					adi[a].push_back({da,p}); // arco bidirez. (non su col. 1 e W)
65 			}
66 		fout << "Case #" << t << ": " << calcola() << endl;
67 	}
68 	return 0;
69 }
```