OII 2023

Soluzioni dei problemi

Bergamo, 13 ottobre 2023

Staff OII

Statistiche

- Sottoposizioni: 2782 (+129 dal 2022)
- 4 file vuoti, 6 file con CRLF, 70 UTF-8
- Parentesi graffe:
  - 1 file con 10000 (tutte aperte!)
  - 4 file, C++ valido, con 0
- 5.0 MB totali
- Parentesi tonde aperte: 68667, chiuse: 68656
- Parentesi quadre aperte: 47347, chiuse: 47346
- Parentesi graffe aperte: 43902, chiuse: 33903
- Parole chiave:
  - int: 29666
  - float: 35
  - if: 16586
  - bug: 1
  - todo: 24
  - flag[10]: 5

Problema A: artemoderna

Subtask 3 (solo 0 e 1)

- In questo caso si dimostra che la risposta è no se e solo se compare la sottosequenza 1010.

Subtask 5 (N ≤ *5000*)

- Varie soluzioni che sfruttano la programmazione dinamica funzionano. Un esempio è calcolare la risposta al problema per ogni coppia (i, j) con i<=j, dove risolviamo il problema per il prefisso [0, j] imponendo che [i, j] sia l'ultimo intervallo girato. Le transizioni impiegano O(1) se fatte con ordine.

Soluzione ottima

- Funziona l'approccio greedy, che scorre da sinistra a destra e ribalta tutti gli intervalli ordinati in senso decrescente più lunghi possibile (così che ribaltati siano ordinati in senso crescente) e alla fine controlla se l'array ottenuto è ordinato. Notiamo che è obbligatorio spaccare se V[i] > V[i+1] e che, in caso di numeri uguali consecutivi, conviene spaccare più a destra possibile.

Problema B: bastioni

Subtask 3 (N ≤ *1000*)

- Una possibile soluzione usa la programmazione dinamica, risolvendo il problema per ogni prefisso.

Soluzione ottima (senza maniglioni antipatici misteriosi)

- Scorriamo i bastioni da sinistra a destra, salvando la direzione in cui ci conviene andare dal punto in cui ci troviamo (ci dice se andando da sinistra a destra stiamo facendo il percorso ottimale in avanti o all'indietro). Ci sono tre opzioni: destra, sinistra, non decisa (all'inizio o dopo un muro).
- Se stiamo andando verso destra e incontriamo un '<' dobbiamo scendere dal bastione, mentre invece con '>' e '=' possiamo andare avanti (e scegliamo di farlo). Analogamente se stiamo andando verso sinistra. Se la direzione non è decisa, la fissiamo non appena incontriamo un '<' o un '>'.
- Stiamo andando, per ogni stanza, il più avanti possibile in una delle due direzioni. Si dimostra (per assurdo) che l'approccio greedy qui presentato fornisce una soluzione ottima.

Problema B: bastioni

Soluzione ottima (con maniglioni antipatici misteriosi)

- Notiamo intanto che nel caso ‘?????’ conviene sostituirli con ‘<><><’, mentre in generale è ottimale mettere una porta nel senso contrario a quello desiderato appena si può.
- Modifichiamo la soluzione ottima aggiungendo una quarta direzione, riassumibile in un “*c’è una direzione, ma non sappiamo ancora se è sinistra o destra*” (se iniziamo la visita con un ‘?’ è indifferente la direzione presa e sapremo quale conviene sostituire solo quando incontreremo un ‘<’ o ‘>’).
- Vediamo i due nuovi casi (‘=’ e ‘#’ si trattano facilmente):
  - direzione qualsiasi, porta ‘?’: basta assumere ‘?’ sia nella direzione contraria (che la nostra sia nota o non nota)
  - direzione non nota, porta ‘<’ o ‘>’: basta assumere che la nostra direzione sia contraria a quella della porta
- Scegliendo ogni volta quella che è per noi l’opzione peggiore in quel momento, si ottiene un numero di arrampicate da eseguire che è il massimo tra tutti gli assegnamenti dei ‘?’ possibili.

Problema C: corridoi

Subtask 3 (M = N - 1, Q = 1)

- Il grafo è un albero, quindi esiste un solo percorso tra ogni coppia di nodi.
- I percorsi $0 \rightarrow 1$ e $1 \rightarrow 2$ sono univocamente determinati e non dipendono da quali archi vengono abbassati. Quindi, conviene abbassare gli archi in modo greedy: prima quelli contenuti in entrambi i percorsi, poi quelli contenuti in almeno un percorso.
- Per trovare $0 \rightarrow 1$ e $1 \rightarrow 2$, è sufficiente una DFS in cui per ogni nodo salviamo il nodo precedente.

Subtask 4 (Q = 1, K_0 = 1)

- Si dimostra che l'arco abbassato deve appartenere ad almeno uno tra i percorsi minimi $0 \rightarrow 1$, $1 \rightarrow 2$.
- La risposta è quindi dist(0,1)+dist(1,2), diminuita di 1 o di 2 a seconda che l'arco abbassato compaia in uno solo dei percorsi minimi o in o in entrambi. Per controllare se un arco $Z \rightarrow W$ appartiene a un percorso minimo $X \rightarrow Y$ verifichiamo se vale dist(X, Y) = dist(X, Z) + dist(Z, W) +dist(W, Y).

Problema C: corridoi

Subtask 5 (Q = 1, N ≤ 1000, M ≤ 2000)

- Si dimostra (per assurdo) che almeno un percorso ottimale è del tipo $0 \to C \to 1 \to C \to 2$, dove C è un nodo, e i percorsi $0 \to C$ e $C \to 2$ sono disgiunti (può anche succedere C = 0,1 o 2)
- Fissato C, è ottimale abbassare gli archi lungo $C \to 1$ finché si può, e poi quelli sugli altri percorsi.
- Possiamo iterare su C. Effettuiamo Dijkstra partendo da C, ricavando dist(C,0), dist(C,1), dist(C,2) e calcolando il risultato. Il valore minimo calcolato al variare di C è la risposta cercata.

Subtask 6 (Q = 1)

- Fissato C, ci interessa conoscere dist(C,0), dist(C,1), dist(C,2) per calcolare la risposta al problema.
- Ma per calcolare questi valori, basta effettuare 3 Dijkstra all'inizio, partendo dai nodi 0, 1, 2.

Problema C: corridoi

Soluzione ottimale (Q qualsiasi)

- Per ogni C disegniamo il grafico, a C fissato, del risultato al variare di K (da qui indicato con y(x)).
- Il grafico contiene al massimo 3 tratti, con coefficienti angolari -2, -1, 0 (nel primo tratto sto abbassando gli archi del percorso C→1, in comune, nel secondo gli altri, nel terzo ho ridotto al massimo e la risposta è 0).
- Rispondere a una query (con un certo K) significa trovare il minimo al variare di tutti i grafici a x = K.

Problema C: corridoi

Proponiamo due modi per implementare la scelta del grafico ottimale.

Modo 1:

- Esistono solo 3 coefficienti angolari distinti. Quindi è possibile mantenere 3 set, ciascuno contenente tutti segmenti che hanno un certo coefficiente angolare. In realtà, il set con coefficiente angolare 0 non serve, perché dal momento in cui viene inserito il primo elemento la risposta alle query sarà sempre 0.
- Scorriamo le query in ordine crescente di K. Per rispondere a una query, si calcola il minimo valore per ogni set (basta che nei set si salvi l'estremo sinistro del segmento) e si prende il minimo.
- Gli eventi che possono accadere sono:
    - un segmento inizia (o finisce), lo inseriamo nel (o rimuoviamo dal) set associato
    - query: troviamo il minimo per quel valore di x, prendendo il minimo da ciascun set

Problema C: corridoi

Modo 2:

- Sfruttiamo la struttura dei grafici, che hanno equazione $y(x) = \max(0, s - x - \min(t,x))$ per qualche scelta dei parametri s, t che dipende da C. Per comodità, ignoriamo il coefficiente 0.
- Possiamo ordinare tutti i grafici in ordine crescente di $s = y(0)$ e tenerli in una coda, tenendo però da parte il grafico ottimale.
- Scorriamo le query in ordine crescente di K (ossia di x), A ogni passo, tolgo dalla coda tutti i grafici con coefficiente angolare -1 in x (notiamo dal disegno che se prima è ottimale il grafico *i* e poi *j*, con $y_i(0) < y_j(0)$, nella loro intersezione *j* ha coefficiente angolare -2) e una volta finito confronto quello salvato come ottimale con la testa della coda e in caso aggiorno.

Bonus: Con piccole modifiche alle soluzioni precedenti si può rispondere a query online. Come?

Problema D: disegno

Subtask 2 (griglia completa):

- Per induzione si dimostra che una griglia è valida **se e solo se** $L=2^k$ per qualche k naturale.

Subtask 3 (N ≤ *40*, *L* ≤ *8*):

- Più o meno qualunque soluzione corretta passa questo subtask. Una possibilità è provare tutte le candidate prime mosse (i centri delle "croci massime") e ricorrere sui quattro quadranti.

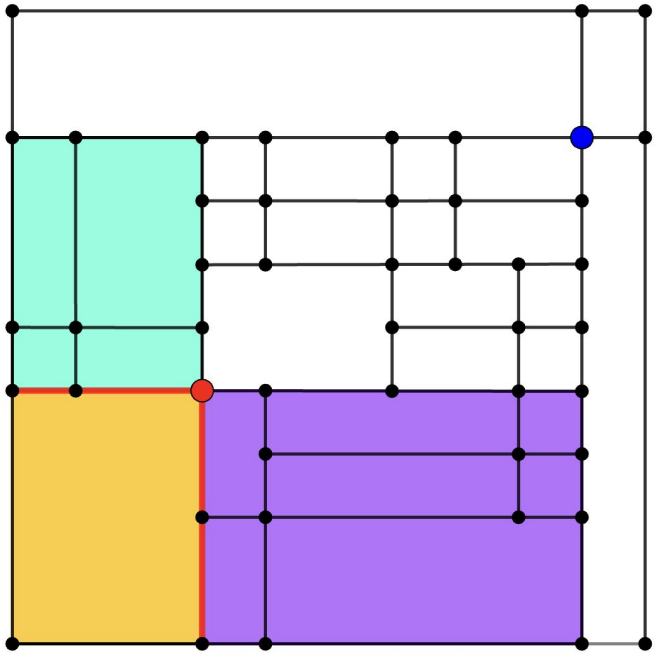
Per le altre soluzioni, ci occupiamo del problema *"Data una griglia valida, trova la sequenza di mosse che la generano"* . Ricostruiamo la sequenza di mosse al contrario, togliendo uno alla volta i "centri" delle mosse a partire da quelle più recenti. Se durante la soluzione "qualcosa va storto", allora la griglia non è valida.

Gli altri subtask usano le stesse idee della soluzione ottima, implementate in maniera meno efficiente. Il vincolo "griglia variegata" permette di distinguere i centri delle mosse, facilitando l'implementazione. Definiamo **nodo** un estremo di due o più segmenti; i **vicini** sono i nodi subito accanto nelle 4 direzioni.

Problema D: disegno

Soluzione ottimale 1:

- Esiste un **rettangolo minimo** con un vertice in *(0, 0)*: il **vertice opposto** è sicuramente il centro di una mossa (ne conosciamo due segmenti).
- Consideriamo i segmenti **rossi** in figura.
- Finché il **rettangolo a destra** (di cui non conosciamo il lato destro) non è libero risolviamo ricorsivamente partendo dal suo vertice in basso a sinistra (che conosciamo). Analogamente per il **rettangolo in alto**.
- Quando sono liberi, i tre rettangolini formano una L (di cui conosciamo le coordinate dei vertici); se il quadrante che completa la L è libero, possiamo fare la mossa, altrimenti ricorriamo su esso.



Problema D: disegno

Implementazione:

- Identifichiamo un rettangolo che ha A come vertice in basso a sinistra (“vertice principale”) e B in alto a destra (“altro vertice”) con [A,B].
- Per ogni nodo, teniamo un **puntatore ai suoi vicini** nella quattro direzioni (inseriamo manualmente anche nodi e archi sul bordo). Questo ci serve per controllare i segmenti delle candidate mosse.
- Teniamo un **set dei quadrati liberi nel disegno**. Notiamo che un tale set identifica univocamente un disegno e ogni disegno valido è unione di rettangoli liberi.
- Funzioni aggiuntive:
    - **Togli(C)**: fa la mossa nel nodo C e aggiorna i suoi (al più) quattro vicini.
    - **Libera_dal_Vertice**(A): libera il rettangolo con vertice principale A e tale che l’ultima mossa sia nel vertice C trovato cercando il rettangolo minimo con vertice principale
    - **Libera_Rettangolo**(A,B): considera il rettangolo con vertice principale A e altro vertice B; chiama Libera_dal_Vertice(A) e toglie i quadranti dal set fino ad avere solo [A,B] come rettangolo libero

Problema D: disegno

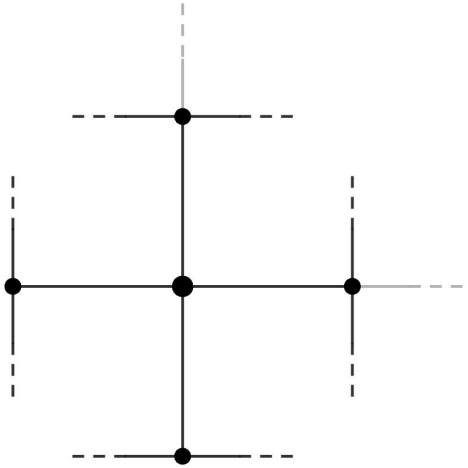
Esempio della figura:

- chiamata a Libera_Rettangolo((0;0),(L;L))
  - check: la griglia non è libera
  - chiamata a Libera_dal_Vertice((0,0))
    - trova come centro il vertice rosso di coordinate (u,v)
    - chiamata a Libera_dal_Vertice((u,0)) che libera il rettangolo viola
    - chiamata a Libera_dal_Vertice((0,v)) che libera il rettangolo celeste
    - usando i puntatori trovo (w,t) come vertice mancante
    - chiamata a Libera_Rettangolo((a,b),(w,t)) che libera il quadrante bianco
  - check: la griglia non è libera
  - chiamata a Libera_dal_Vertice((0,0))
    - trova come centro il vertice blu di coordinate (u’,v’) …

Problema D: disegno

Soluzione ottimale 2:

- In ogni nodo i segmenti che lo hanno come estremo possono formare una T o una croce.
- Dico che il vicino sinistro di un incrocio è "completo" se il suo incrocio ha entrambi i bracci verticali (ci “bloccano la strada” se proviamo ad andare a sinistra). Analogamente gli altri.



- Proposizione: il disegno ammette una sequenza di mosse valida se e solo se esiste un nodo tale che tutti e quattro i suoi vicini sono completi e gli incroci dei vicini sinistro e di sopra *non* sono croci. Inoltre, se valgono queste condizioni, l’ultima mossa ha come centro quel nodo, che chiamiamo “buono”.
- La dimostrazione del “se” procede per induzione sul numero di mosse. La dimostrazione del “solo se” la lasciamo al lettore.
- Rappresentiamo un disegno come un grafo con archi tra nodi vicini e facciamo la mossa ad uno ad uno sui nodi buoni, aggiornando i vicini e la lista dei nodi buoni se serve.

Problema D: disegno

Implementazione:

- Rappresentiamo il disegno come un grafo in cui un arco connette un nodo ai suoi vicini. Teniamo i nodi buoni in una coda.
- A ogni passo, prendiamo un nodo buono e facciamo la mossa, ossia marchiamo i suoi vicini come cancellati e
- While the queue is not empty, take the front intersection and make a move. Making a move means marking the intersection and all its neighbors as "erased", so that they can't be an open move anymore, and updating the neighbors of O(1) intersections.
- We also need to check if O(1) intersections have become open moves and, in that case, push
- them to the queue. At the end, we check if all intersections have been erased: if yes, the sequence of moves found s a valid solution, otherwise there is no solution.