Laboratorio di Algoritmi e Strutture Dati

Studente: Giacomo Guidi 988004

Progetto “Piastrelle digitali”

Indice:

1. Il problema

Il problema proposto richiede di analizzare le configurazioni di insiemi di piastrelle digitali disposte su un piano bidimensionale e di studiare l'influenza che queste configurazioni esercitano sulle piastrelle circostanti sulla base del loro stato e sulle operazioni richieste in input da parte dell’utente.

Il piano in esame è suddiviso in quadrati di lato unitario, ciascuno dei quali è occupato da una piastrella inquadrata tra quattro vertici (a,b), (a, b+1), (a+1, b+1), (a+1, b). Ogni piastrella potrà essere accesa o spenta, con un proprio colore e una propria intensità. Inoltre, ogni piastrella possiede delle piastrelle circonvicine, ovvero quelle piastrelle che condividono con essa almeno un vertice sul piano.

Ogni piastrella influenza le sue circonvicine in base alle regole di propagazione del piano, definite come $k1\alpha1 + k2\alpha2 + \cdots + kn\alpha n \rightarrow \beta$, dove ki sono numeri e αi e β sono colori, dove una piastrella assume il colore β se è circondato da almeno ki *piastrelle di colore* αi, *per ogni* $i=1, 2, \ldots, n$, *con* $k1+k2+\ldots+kn <= 8$. Le regole possono essere propagate a comando, applicandole nell’ordine in cui sono salvate su una piastrella o su un blocco.

Le regole di propagazione possono essere ordinate per inserimento o per numero di utilizzo crescente.

Si possono definire delle piste di piastrelle da P1 a Ph, di lunghezza h, ovvero sequenze di piastrelle accese P1, P2, …, Ph tali che Pi è circonvicina a Pi+1 per ogni $1 \leq i \leq h - 1$. Le piste, essendo definite dalle piastrelle circonvicine, ed essendo una piastrella circonvicina di sé stessa (condivide ben quattro vertici con sé stessa), possono essere anche di una sola piastrella P.

Le regioni invece sono insiemi di piastrelle tali che per ogni coppia di piastrelle nell’insieme c’è una pista tra di esse; dunque, avremo anche regioni composte da un solo elemento.

Un blocco infine è una regione massimale, ovvero quella con il maggior numero di elementi in essa tra le regioni possibili composte dalle sue piastrelle, mentre un blocco omogeneo è una regione massimale in cui tutte le piastrelle hanno lo stesso colore.

2. Modellazione del problema

Modelliamo il problema: dobbiamo salvarci n*m piastrelle, con n, m ≥ 0, che possono essere accese o spente e sono identificate univocamente dalla loro posizione. Possiamo pensare al piano bidimensionale che contiene le piastrelle come un grafo non orientato composto da un nodo per ogni piastrella, il quale avrà un numero di archi pari al numero dei suoi nodi circonvicini, e partendo da un grafo vuoto aggiungerne nodi quando accenderemo nuove piastrelle. Di fatto non sarà propriamente un grafo interamente connesso, ma inizialmente il piano sarà composto da un insieme di grafi che andranno ad unirsi ogni qualvolta viene accesa una piastrella.
Definito come un grafo, potremo definire una pista come una generica sequenza di nodi A1, …, Ah collegati tra loro e una regione come un insieme specifico di nodi tra di loro collegati, mentre definiremo un blocco come una regione di dimensione massima

Ogni piastrella dovrà mantenere il proprio stato, definito dalla coppia (colore, intensità). I valori di questa coppia saranno (“”, 0) se la piastrella è spenta o (*colore basato sull'alfabeto {a, b, …, z}, intero > 0*) altrimenti. Ogni piastrella nascerà spenta, potrà essere accesa attraverso un comando e conseguentemente rispenta attraverso un altro comando

Il piano, oltre alle piastrelle, avrà associate una sequenza di regole proprie, raccolte in una coda semplice con possibilità di riordinarle in base al loro utilizzo e di scorrerle dalla prima inserita per applicarle. Esse dovranno mantenere lo stato richiesto alle piastrelle vicine e il colore derivante se la regola è applicabile, nonché sarà necessario contare quante volte la regola viene applicata con successo

3. Strutture dati utilizzate

Una volta modellato il problema, inizio a ragionare sulle strutture dati da utilizzare.
Dapprima ho creato una struttura Punto, composto dagli interi x e y, che rappresenta le coordinate di un punto nel piano bidimensionale.

Per le piastrelle ho creato una Struct Piastrelle che contiene, oltre al campo String colore e al campo Intero intensità che mi permettono di mantenere lo stato della piastrella, due strutture ausiliarie: una slice di Punto chiamata punti che contiene le coordinate univoche dei quattro vertici della piastrella, sempre inseriti nell'ordine {*x*,*y*}, {*x*,*y*+1}, {*x*+1,*y*+1}, {*x*+1,*y*}, e permette sia di inquadrare la piastrella sul piano sia di ottenere facilmente i suoi punti; una slice di puntatori a Piastrelle, chiamato circonvicini, contenente i puntatori alle piastrelle che sono circonvicine alla piastrella x,y presa in oggetto, permettendone un accesso più facile e veloce essendo esse sempre disponbili all'interno di una slice di dimensione fissata 9 e disposti sempre nell'ordine

[x-1, y+1 | x, y+1 | x+1, y+1
x-1, y | x, y | x+1, y
x-1, y-1 | x, y-1 | x+1, y-1],

portandoci ad avere un costo di ricerca fissato nelle varie operazioni ad O(9).
Questa struttura, oltre alle due variabili semplici legate allo stato dal costo di spazio O(1), costa in termini di spazio Θ(9) per la slice di circonvicini e Θ(4) per la slice di punti.

Anche per le regole ho creato una struct apposita, chiamata Regola, contenente un contatore usato che viene incrementato ogni volta che la regola viene applicata con successo al piano, un campo stringa beta rappresentante il colore che otterrà una piastrella se le piastrelle nel suo intorno rispetteranno i requisiti, una mappa[stringa]intero chiamata alfa che conterrà al suo interno le coppie *αi* colore *ki* numero rappresentanti il numero di piastrelle di colore *αi* necessarie (basandoci sull'idea che i colori saranno inseriti in modo non ripetuto all'interno di una stessa regola) e una slice di stringhe ordine che mantengono l'ordine con cui i parametri alfa sono ineriti. Questo perchè sebbene la mappa permetta di mantenere la relazione chiave valore, ha un costo di inserimento basso ed è studiata appositamente per permettere il minimo costo di ricerca di un elemento sia data la chiave sia scorrendo tutta la mappa collegata in un ciclo for, essa non

mantiene l'ordine con cui i parametri alfa sono stati inseriti come farebbe un array e dunque questa struttura di supporto permette di mantenere tale ordine.
Per ogni regola inoltre prendo per assodato che il valore di ogni *ki* sarà sempre minore uguale di 8 e che la somma dei vari *ki* intesa come *k1+k2+…+kn* sarà sempre minore uguale di 8, questo poiché il numero massimo di piastrelle circonvicine ad una piastrella è 8 e dunque una regola non potrà richiedere un numero di alfa maggiore.
A livello di spazio, dunque, questa struttura avrà una mappa di dimensione O(8) ed equivalentemente una slice di dimensione O(8).

Infine, potremo unire piastrelle e regole all'interno della struct Piano, che conterrà all'interno di una mappa[Punto]*Piastrella chiamata Piastrelle le piastrelle contenute nel piano, identificate attraverso il punto univoco che ne identifica il vertice basso a sinistra permettendone così l'individuazione nel grafo, e all'interno di un campo slice di Regole l'elenco delle regole, che non dovendo essere ricercate per singolo elemento ma richiedendo uno specifico ordine (inizialmente ordinate per inserimento, su richiesta ordinate per utilizzo) appare meglio di una mappa, permettendo difatti un facile riordino.
La scelta della mappa è sensata pensando che inizialmente le piastrelle non inizializzate, ottenendo all'avvio del programma un piano e di conseguenza un grafo completamente vuoto, per poi essere generato un nuovo elemento della mappa quando la piastrella corrispondente viene accesa. Questo ci permette di risparmiare spazio e tempo e attraverso le proprietà delle mappe di go di riconoscere facilmente se una piastrella esiste o no (sicuramente spenta).
Il costo in spazio risulterà dunque essere di O(n) per la mappa e di O(m) per la slice delle regole, con n=#piastrelle accese almeno una volta nel piano e m=#regole del piano

4. Progettazione degli algoritmi

Di seguito elencherò tutti gli algoritmi utilizzati e le funzioni implementate, soffermandomi principalmente su quelli più complessi e presentandone lo pseudocodice. È presente una parziale suddivisione tra algoritmi e funzioni che eseguono direttamente le operazioni richieste dal problema e quelle di pura utilità del codice.

Note sulle funzioni di go: nel programma faccio un uso costante di diverse funzioni presenti in go, quali len(), make() e append(), tutte quante con costo costante O(1). Per brevità, dunque, esse non saranno menzionate nel calcolo dei costi asintotici, così come gli assegnamenti e le operazioni di controllo del flusso Caso diverso invece per la funzione fmt.Println(), che con costo O(k*n) verrà invece menzionata brevemente nei calcoli.

Funzioni di utilità:
Sono le funzioni che servono come supporto alle operazioni richieste dal progetto senza essere esplicitamente richieste.

- **Funzione esegui(Piano, string) Piano** → funzione che riconosce gli input inseriti dall'utente e ne esegue le relative funzioni. Suddivide la stringa con la funzione Split() di go e poi chiama le operazioni relative al comando usando uno switch.
    - **Costo in tempo:** O(1) per tutte le operazioni di assegnamento e controllo + O(n) per l'operazione Split(), con n=lunghezza del comando. **Totale costo O(n)**.
    - **Costo in spazio:** Non ci sono strutture dati aggiuntive, **costo O(1).**

- **Funzione creaCirconvicini(Piano, intero, intero)** → inserisce alla piastrella passata come parametro il campo circonvicini, una slice di dimensione fissa 9 contenente i puntatori alle piastrelle circonvicine, ed aggiorna il campo circonvicini delle piastrelle circonvicine con il proprio puntatpre.
    - **Costo in tempo:** Esegue unicamente controlli di flusso in ordine per inserire il puntatore o nil all'interno della slice, dunque con **costo O(1).**
    - **Costo in spazio:** creo una slice contenente i 9 circonvicini, quindi **costo Θ(9).**

- **Funzione Accesa(Piastrella) boolean →** restituisce True se la piastrella è accesa, ovvero se la sua intensità è maggiore di 0, false altrimenti.
    - **Costo in tempo:** controllo + ritorno, **costo O(1).**
    - **Costo in spazio:** solo return, **costo O(1).**

- **Funzione trovaBlocco(Piano, intero, intero) []Piastrella →** trova il blocco, ovvero la regione di ampiezza massima, di piastrelle circonvicine alla piastrella in posizione x, y, se la piastrella x,y non è spenta. Sfrutta due slice, una che si riempie e verrà restituita e una che si riempie e si svuota come una coda, assieme ad una mappa[Punto]bool che mi permette di tenere salvati le piastrelle già visitate senza dover eseguire un altro ciclo. Poi esegue 2 cicli innestati: il primo viene eseguito finché la coda non è vuota e ad ogni iterazione ne estrae il primo elemento, mentre il secondo scandisce le piastrelle circonvicine alla piastrella estratta dalla coda (nello specifico caso implementativo viene eseguito sempre 9 volte). All'interno del ciclo sfrutto anche la mappa per controllare, oltre che la piastrella non sia spenta o nil, che essa non sia già stata visitata. In caso contrario aggiungo il circonvicino alle due slice e alla mappa segnandolo come visitato.
    - **Costo in tempo:** O(1) per assegnamenti e controlli di flusso, 2 O(1) per le due chiamate alla funzione Accesa(), O(n) per il primo ciclo e **Θ(9)** per il secondo. **Totale costo O(1) + 2 * O(1) + O(n)* Θ(9) = O(n*9) = O(n).**
    - **Costo in spazio:** O(n) per ciascuna slice e per la mappa. **Totale costo O(n + n + n).**

```
Algoritmo TrovaBlocco(Piano p, intero x, intero y) []Piastrella
    If piastrella{x,y} è spenta then return
    P ← crea un insieme vuoto e aggiungi la piastrella{x,y}
    C ← crea una coda vuota e aggiungi la piastrella{x,y}
    While C non è vuota do
        Curr ← estrai il primo elemento dalla coda C
        C ← C-{Curr}
        For each I ∈ piastrella circonvicina non nulla e non spenta di Curr do
            If I non è già stato visitato then        //utilizzo una mappa[Punto]bool
                P, C ← aggiungi I
    Return P
```

- **Funzione trovaBloccoOmogeneo(Piano, intero, intero) []Piastrella →** trova il blocco omogeneo, ovvero la regione di ampiezza massima con piastrelle tutte dello stesso colore, di piastrelle circonvicine alla piastrella in posizione x, y, se la piastrella x,y non è spenta. L'algoritmo è essenzialmente identico alla funzione trovaBlocco() con la sola aggiunta nel for che scandisce le piastrelle circonvicine (il secondo ciclo for per intenderci) del controllo che la piastrella in esame sia dello stesso colore della piastrella da cui facciamo partire il blocco.
    - **Costo in tempo:** O(1) per assegnamenti e controlli di flusso, 2 O(1) per le due chiamate alla funzione Accesa(), O(n) per il primo ciclo e **Θ(9)** per il secondo. **Totale costo O(1) + 2 * O(1) + O(n)* Θ(9) = O(n*9) = O(n).**
    - **Costo in spazio:** O(n) per ciascuna slice e per la mappa. **Totale costo O(n + n + n).**

- **Funzione merge([]regola, []regola) []regola →** svolge il merge tra due array contenenti regole nell'applicazione del mergeSort per ordinare un array di regole. Si basa sul classico algoritmo di merge, adattato al programma.
    - **Costo in tempo:** Assumendo che i due array derivino entrambi dallo stesso array di lunghezza n, avremo O(n-1) perr il primo ciclo for (quello primario) e O(1) per gli altri due cicli for secondari. **Totale costo O(n-1) + 2 * O(1) = O(n-1).**
    - **Costo in spazio:** O(n) per creare la slice da ritornare. **Totale costo O(n).**

Funzioni basate sulle operazioni richieste dal programma:

Sono le funzioni che eseguono le operazioni richieste dal progetto e che vengono chiamate direttamente dall'utente in base all'input passato al programma.

- **Funzione colora(Piano, intero, intero, stringa, intero)** → colora la piastrella in posizione x, y con il colore e l'intensità passati come parametro. Se la piastrella è inesistente la crea, la inserisce nella mappa e la accende. Se invece la piastrella è esistente, a prescindere che essa sia spenta o meno, la funzione ne cambierà il colore e l'intensità con quelli passati in input.
  - **Costo in tempo:** O(1) per assegnamenti e controlli, O(1) per la chiamata alla funzione creaCirconvicini(). **Totale costo O(1) +O(1) = O(1).**
  - **Costo in spazio:** O(1) se la piastrella esiste, O(1)+O(1) se la piastrella non esiste (l'oggetto piastrella costa in spazio O(1)). **Totale costo O(1).**

- **Funzione spegni(Piano, intero, intero)** → spegne la piastrella in posizione x, y, ponendone il colore a "" e l'intensità a 0. Esegue la chiamata alla funzione Accesa poiché sarebbe inutile spegnerla se è già spenta e al contempo potrebbe non esistere la piastrella.
  - **Costo in tempo:** O(1) per lo spegnimento, O(1) per la chiamata alla funzione Accesa(). **Totale costo O(1) + O(1) = O(1).**
  - **Costo in spazio:** non vengono usate strutture dati ausiliarie. **Totale costo O(1).**

- **Funzione regola(Piano, stringa)** → Definisce la regola di propagazione $k_1\alpha_1 + k_2\alpha_2 + \cdots + k_n\alpha_n \rightarrow \beta$ data dalla stringa passata in input e la inserisce in fondo all'elenco delle regole, dopo averla suddivisa nelle sue componenti attraverso la funzione split(). Dopodichè assegna β e poi inserisce in una mappa con la coppia (α_n, k_n) e in una slice gli elementi α nell'ordine in cui sono scritti nella regola, così da poter poi stampare gli elementi α in maniera ordinata.
  - **Costo in tempo:** O(1) per gli assegnamenti, O(8) per il ciclo for che scandisce gli elementi $k_n\alpha_n$. **Totale costo O(1) + O(8) = O(8) = O(1).**
  - **Costo in spazio:** O(8) per la slice ordina e per la mappa, O(1) per le variabili. **Totale costo O(8) = O(1).**

- **Funzione stato(Piano, intero, intero) (stringa, intero)** → Stampa e restituisce il colore e l'intensità di Piastrella(x, y). Se Piastrella(x, y) è spenta, non stampa nulla. Esegue un controllo chiamando la funzione Accesa() e poi stampa lo stato della piastrella.
  - **Costo in tempo:** O(1) per gli assegnamenti e i controlli, O(k+n) per funzione di println(), con k= 2 e n= lunghezza dello stato della piastrella. **Totale costo O(1) + O(k+n) = O(n)**
  - **Costo in spazio:** non vengono usate strutture dati ausiliarie. **Totale costo O(1).**

- **Funzione stampa(Piano)** → stampa l'elenco delle regole di propagazione, nell'ordine attuale. Prima stampa la parentesi aperta, poi crea una stringa riscrivendo la regola nel formato $\beta : k_1\ \alpha_1 \ldots k_n\ \alpha_n$ attraverso un ciclo for per ogni regola del piano. Infine stampa una parentesi di chiusura. Di conseguenza la funzione avrà un costo di O(k*1) per le due stampe delle parentesi, il ciclo esterno richiederà O(n) iterazioni, mentre al suo interno troveremo un ciclo che richiede O(8) iterazioni e una funzione di stampa da O(k*n), portando il tutto ad un costo di O(k*n^2)
  - **Costo in tempo:** O(1) per gli assegnamenti, O(8) per il ciclo for che scandisce gli elementi $k_n\alpha_n$. **Totale costo O(1) + O(8) = O(8) = O(1).**
  - **Costo in spazio:** O(8) per la slice ordina e per la mappa, O(1) per le variabili. **Totale costo O(8) = O(1).**

- **Funzione blocco(*Piano, intero, intero)** → **c**alcola e stampa la somma delle intensità delle piastrelle contenute nel blocco di appartenenza di Piastrella(x, y). Se Piastrella(x, y) è spenta, stampa 0. Prima controlla se la piastrella è spenta, poi esegue una chiamata alla funzione trovaBlocco per ottenere

una slice contenete i puntatori alle piastrelle che fanno parte del blocco ed infine per ogni elemento della slice risultante somma le intensità, per poi stampare il risultato.

- **Costo in tempo:** O(1) per gli assegnamenti, O(1) per la chiamata alla funzione Accesa(), O(k+m) con k=1 e m≤8, quindi O(1) per la funzione di println(), O(n) per la funzione di trovaBlocco(), O(n) per il ciclo che calcola l'intensità complessiva. **Totale costo O(1) + O(1) + O(1) + O(n) + O(n) = O(n).**
- **Costo in spazio:** O(n) per la slice contenente il blocco, O(1) per le variabili. **Totale costo O(n).**

- **Funzione bloccoOmog(*Piano, intero, intero) → c**alcola e stampa la somma delle intensità delle piastrelle contenute nel blocco omogeneo di appartenenza di Piastrella(x, y). Se Piastrella(x, y) è spenta, stampa 0. Esegue esattamente gli stessi passaggi della funzione blocco(), salvo al suo interno chiamare la funzione trovaBloccoOmogeneo().
  - **Costo in tempo:** O(1) per gli assegnamenti, O(1) per la chiamata alla funzione Accesa(), O(k+m) con k=1 e m≤8, quindi O(1) per la funzione di println(), O(n) per la funzione di trovaBloccoOmogeneo(), O(n) per il ciclo che calcola l'intensità complessiva. **Totale costo O(1) + O(1) + O(1) + O(n) + O(n) = O(n).**
  - **Costo in spazio:** O(n) per la slice contenente il blocco, O(1) per le variabili. **Totale costo O(n).**

- **Funzione propaga(*Piano, intero, intero) →** Applica a Piastrella(x, y) la prima regola di propagazione applicabile dell'elenco, ricolorando la piastrella. Se nessuna regola è applicabile, non viene eseguita alcuna operazione. Inizialmente controlla che la piastrella sia esistente e nel caso negativo la crea accendendola parzialmente, ovvero la inserisce nella mappa delle piastrelle ma come se fosse spenta. Dopodiche crea due strutture di supporto: un array di 8 elementi che contiene le piastrelle che sono posizionate intorno alla piastrella(x, y) (copia l'array di circonvicini togliendo l'elemento di posizione 4, quello che punta alla piastrella(x,y) stessa) e una mappa[stringa]intero che attraverso un ciclo for sull'array intorno salva le coppie (colore, numero di piastrelle di quel colore che stanno intorno alla piastrella(x,y)). Una volta create le strutture ausiliarie vengono eseguiti due cicli for innestati: quello esterno scandisce le regole del piano, quello interno la mappa alpha di ogni regola. La mappa alpha di una regola contiene coppie (colore, quantità di piastrelle di quel colore necessarie nell'intorno di una piastrella) e per ogni colore l'algoritmo controlla che nell'intorno della data piastrella vi siano abbastanza piastrelle con quel colore, confrontando il k richiesto con quello presente nell'intorno della piastrella. Se anche solo un colore non è presente nella quantità richiesta, viene interrotto il ciclo e si passa alla regola successiva. In caso contrario, si potrà applicare quella regola alla piastrella, andandone a modificare il colore e accendendola in caso essa sia spenta. Siccome le regole sono contenute in un array che le mantiene sempre ordinate, la prima regola trovata sarà quella applicata e bloccherà il ciclo restante, andando in caso anche ad accendere la piastrella se essa risulta spenta.
  - **Costo in tempo:** O(1) per gli assegnamenti, O(1) per la chiamata alla funzione Colora(), Θ(8) per il ciclo che inserisce i valori della mappa degli intorno, O(n) per il ciclo che scandisce le regle, O(8) per il ciclo che scandisce gli elementi alpha all'interno del ciclo delle regole. **Totale costo O(1) + O(1) + Θ(8) + O(n) * O(8) = O(n*8) = O(n).**
  - **Costo in spazio:** Θ(8) per la slice degli intorni e per la mappa degli intorni, O(1) per le variabili. **Totale costo Θ(8) = O(1).**

```
Algoritmo propaga(Piano P, intero x, intero y)
    Intorno ← insieme vuoto di piastrelle
    For each circonvicino della piastrella{x,y}-{x,y} stessa do
        Intorno←Intorno U circonvicino
    coppiaIntorno ← crea insieme chiave-valore vuoto di coppie (colore, intero)
    for each intorno ∈ Intorno do
        if intorno.colore è gia presente in coppiaIntorno then
            aumenta di uno il valore della coppiaIntorno con chiave intorno.colore
```

```
        else
                crea una nuova coppia (intorno.colore, 1)
    for each regola ∈ piano.regole do
        for each coppia (α,k) di una regola do
                if valore della coppiaIntorno di chiave α < k then
                        la regola non è applicabile alla piastrella x,y
                        passa alla prossima regola
                else
                        continua
        if regola è applicabile alla piastrella x,y
                regola.usato ←regola.usato+1
                if piastrella(x,y) è spenta then
                        piastrella(x,y).intensità ←1
                piastrella(x,y).colore ← β
                return
```

- **Funzione propagaBlocco(*Piano, int, int) →** Propaga il colore sul blocco di appartenenza di Piastrella(x, y). Inizialmente controlla che la piastrella sia accesa e nel caso negativo ritorna senza fare nulla. In caso positivo invece crea due strutture dati ausiliarie: una slice di puntatori a piastrelle che riempie chiamando la funzione trovaBlocco() che conterrà le piastrelle a cui applicare le regole e una slice di piastrelle che conterrà, attraverso l'inserimento all'interno di un ciclo for che scandisce le piastrelle del blocco, la copia delle piastrelle del blocco. Dopo di che esegue un primo ciclo for che scandisce le piastrelle del blocco: al suo interno, per ogni iterazione, vengono create due strutture di supporto: un array di 8 elementi che contiene le piastrelle che sono posizionate intorno alla piastrella del blocco che stiamo vagliando (copia l'array di circonvicini togliendo l'elemento di posizione 4, quello che punta alla piastrella(x,y) stessa) e una mappa[stringa]intero che attraverso un ciclo for sull'array intorno salva le coppie (colore, numero di piastrelle di quel colore che stanno intorno alla piastrella(x,y)). Una volta create le strutture ausiliarie vengono eseguiti due cicli for innestati: quello esterno scandisce le regole del piano, quello interno la mappa alpha di ogni regola. La mappa alpha di una regola contiene coppie (colore, quantità di piastrelle di quel colore necessarie nell'intorno di una piastrella) e per ogni colore l'algoritmo controlla che nell'intorno della data piastrella vi siano abbastanza piastrelle con quel colore, confrontando il k richiesto con quello presente nell'intorno della piastrella. Se anche solo un colore non è presente nella quantità richiesta, viene interrotto il ciclo e si passa alla regola successiva. In caso contrario, si potrà applicare quella regola alla piastrella presente nella slice delle copie, andandone a modificare il colore, e mantenendo lo stato iniziale del blocco inalterato (come richiesto dal problema). Infine, per applicare la propagazione alle piastrelle effettivamente presenti sul piano, eseguo un ciclo for che scandisce le piastrelle copia e modifica il colore di quelle effettive se necessario.
    - **Costo in tempo:** O(1) per gli assegnamenti, O(1) per la chiamata alla funzione Accesa(),O(n) per la chiamata alla funzione trovaBlocco(), O(n), con n=#piastrelle nel blocco, per eseguire la copia del blocco, O(n) per scandire nuovamente il blocco nel primo ciclo, Θ(8) per il ciclo che inserisce i valori della mappa degli intorno (essendo innestato O(n*8), O(m), con m=#regole del piano, per il ciclo che scandisce le regole per ogni piastrella (O(n*m), O(8) per il ciclo che scandisce gli elementi alpha all'interno del ciclo delle regole per ogni piastrella (O(n*m*8), O(n) per eseguire la colorazione delle piastrelle del piano effettivo. **Totale costo O(1) + O(1) + O(n) + O(n) + O(n)*Θ(8) + O(n) *O(m) * O(8) + O(n)= O(n*8) + O(n*m*8) = O(n*m*8) = O(n*m).**
    - **Costo in spazio:** O(n) per la slice del blocco, per la slice di copie, Θ(8) per la slice degli intorni e per la mappa degli intorni, O(1) per le variabili. **Totale costo O(n) + Θ(8) = O(n).**

```
Algoritmo propagaBlocco(Piano P, intero x, intero y)
    If piastrella{x,y} è spenta then return
    Blocco ←crea un array di piastrelle vuoto e riempilo con la funzione trovaBlocco()
    Copia ←crea un array di piastrelle vuoto
    For each piastrella p∈Blocco do
        Copia←Inserisci la piastrella p
    For each piastrella p∈Blocco do
        Intorno ← insieme vuoto di piastrelle
    For each circonvicino della piastrella{x,y}-{x,y} stessa do
            Intorno←Intorno U circonvicino
    coppiaIntorno ← crea insieme chiave-valore vuoto di coppie (colore, intero)
    for each intorno ∈ Intorno do
            if intorno.colore è gia presente in coppiaIntorno then
                aumenta coppiaIntorno[intorno.colore].valore di uno
            else
                crea una nuova coppia (intorno.colore, 1)
        for each regola ∈ piano.regole do
            for each coppia (α,k) di una regola do
                if valore della coppiaIntorno di chiave α < k then
                    la regola non è applicabile alla piastrella x,y
                    passa alla prossima regola
                else
                    continua
            if regola è applicabile alla piastrella x,y
                regola.usato ←regola.usato+1
                copia(x,y).colore ← β
                esci dal ciclo
    for each piastrella c ∈ Copia do
        if colore piastrella c diverso dalla corrispondente piastrella del blocco then
            piastrella del blocco.colore ←c.colore
```

- **Funzione mergeSort([]regola) →** Implementa l'algoritmo di merge sort per ordinare l'elenco delle regole di propagazione in base al consumo delle regole stesse: la regola con consumo maggiore diventa l'ultima dell'elenco. Se due regole hanno consumo uguale mantengono il loro ordine relativo. Dopo aver controllato se la slice passata come parametro non è diventata lunga 1, divide a metà la slice delle regole ed esegue chiamate ricorsive per poi riunificare le varie partizioni della slice con la chiamata alla funzione merge()
    - **Costo in tempo:** O(1) per gli assegnamenti, costo ricorsivo di C(n/2) per ogni chiamata ricorsiva alla funzione di mergesort() se n>1 o 0 se n=1, Θ(n-1) per la chiamata alla funzione merge(). Calcolando il costo ricorsivo, esso risulta essere Θ (n*logn). **Totale costo O(1) + Θ(n*logn) + O(n-1) = O(n*logn).**
    - **Costo in spazio:** Θ(n) per la slice delle regole, O(1) per le variabili. **Totale costo Θ(n).**

- **Funzione pista(*piano, intero, intero, string) →** Stampa la pista che parte da Piastrella(x, y) e segue la sequenza di direzioni s, se tale pista è definita. Altrimenti non stampa nulla. s sarà una stringa nel formatto NN,NO,NE,.... Dopo aver controllato che la piastrella non sia spenta, la funzione crea due slice di supporto, una per salvarsi le indicazioni suddivise e una per salvarsi le piastrelle che fanno parte della pista. Dopo aver eseguito un ciclo for che scandisce la stringa di indicazioni per caricare nella slice le indicazioni, viene eseguito un ciclo for che scandisce le indicazioni stesse: al suo interno viene riconosciuto quale circonvicino è indicato dalla direzione e, se tale piastrella non è spenta, viene aggiunta alla pista e usata per controllare la prossima indicazione. Infine, la funzione stampa la pista ottenuta attraverso un ciclo for che scorre tutte le piastrelle della lista.

- **Costo in tempo:** O(1) per gli assegnamenti, O(n) con n=#numero di indicazioni per il for che carica le indicazioni e per quello che le scandisce nel ciclo principale, O(m) con m=#lunghezza pista per il ciclo for che scandisce le piastrelle appartenenti alla pista, O(k+i) con k,i numeri interi relativamente piccoli e assumibili a O(1) per le funzioni di stampa. Se la pista sarà completa le piastrelle saranno una in più rispetto alle indicazioni. **Totale costo O(1) + O(n) + O(n) + O(m) + O(1) = O(m).**
- **Costo in spazio:** O(n) per la slice delle indicazioni, O(m) per la slice delle piastrelle, O(1) per le variabili. **Totale costo O(m).**

```
Algoritmo pista(*piano p, intero x, intero y, stringa s)
    If piastrella{x,y} non è accesa then return
    Indicazioni ←crea una coda vuota di stringhe
    For each indicazione in s do
        Indicazioni ←aggiungi indicazione
    Piastrelle ← crea un insieme vuoto di piastrelle
    Piastrelle ← Piastrelle U piastrella{x,y}
    P_controllata ← piatrella{x,y}
    For each indicazione ∈ indicazioni then
        P ←ottieni la piastrella indicata attraverso i circonvicini
        If P è accesa then
            P_controllata←P
            Piastrelle ← Piastrelle U P
        Else
            Return
    For each p ∈ Piastrelle do
        Stampa p
```

- **Funzione lung(*piano, int x, int y, int x2, int y2) →** Determina la lunghezza della pista più breve che parte da Piastrella(x1, y1) e arriva in Piastrella(x2, y2). Altrimenti non stampa nulla. Dopo aver controllato che le due piastrelle siano accese, la funzione crea una coda in cui saranno inseriti tutti i nodi da visitare per ogni livello e una mappa per tenere traccia delle piastrelle visitate. Dopo di che inizia il primo for, che scansiona i vari livelli. Per ogni livello, un altro ciclo for scansiona le piastrelle contenute, estraendole una ad una dalla coda. Dopo aver controllato se la piastrella vagliata non è quella di arrivo, nel qual caso stampa il risultato e ritorna, esegue un ultimo ciclo for per inserire all'interno della coda i nodi circonvicini al nodo appena visitato.
    - **Costo in tempo:** O(1) per gli assegnamenti, O(n) con n=#numero di livelli per il for che scandisce i livelli, O(m) con m=#numero di piastrelle in coda per il ciclo for che scandisce le piastrelle, O(k+i) con k,i numeri interi relativamente piccoli e assumibili a O(1) per le funzioni di stampa, Θ(9) per il for che scandisce i circonvicini. **Totale costo O(1) + O(n)*O(m)*O(1) + O(n)*O(m)* Θ(9) = O(n*m*9) = O(n*m).**
    - **Costo in spazio:** O(n) per la coda e per la mappa, O(1) per le variabili. **Totale costo O(n).**

```
Algoritmo lung(pista p, intero x1, intero x2, intero y1, intero y2)
    If piastrella{x1, y1} o piastrella{x2,y2} non è accesa then return
    Coda←crea una coda vuota
    Coda←inserisci la piastrella{x1,y1}
    Visitato←crea un insieme di coppie (piastrelle, booleano)
    Visitato←inserisci la coppia (piastrella{x1,y1}, true)
    Distanza ←1
    While coda non è vuota do
        For each elemento della coda do
            Piast←estrai il primo elemento dalla coda
```

```
                If Piast è la piastrella{x2, y2} then
                        Stampa la distanza
                        Return
                For each circonvicino di Piast do
                        If circonvicino esiste e non è presente nell'insieme Visitato
                                Coda ←inserisci il circonvicino
                                Visitato ←inserisci la coppia (circonvicino, true)
        Distanza←Distanza + 1
```

5. Esempi di funzionamento

Nella seguente parte, presenterò una serie di esempi di input creati da me e passati al programma ed i relativi output prodotti. Aggiungo che nel progetto passato sarà presente una cartella chiamata test_formato_txt in cui sono inseriti i file .txt di input/output qui sotto proposti, mentre nella cartella test è presente la seguente repo di github https://github.com/lochy54/test.git dove sono inseriti i test prodotti da me e dal collega Luca Carone Polettini, uniti a quelli proposti dalla professoressa, con un relativo file go per eseguire i vari test, sia singolarmente che tutti assieme, che permette di visualizzare anche eventuali differenze tra l'output prodotto e l'output aspetto.

- Input_1: testo il comportamento delle operazioni richieste per luglio (pista e lunghezza), con anche un paio di casi limite

```
C 1 2 f 48
C 3 5 o 66
C 4 3 r 18
C 5 1 j 5
C 2 0 g 26
C 4 0 x 45
C 4 1 j 6
```

```
C 0 0 d 77
C 1 5 m 79
C 0 2 e 71
C 4 4 g 6
C 0 1 j 47
C 1 3 o 43
C 3 1 h 68
```

```
C 4 5 c 50
C 1 5 e 66
C 2 4 m 49
C 2 2 g 96
C 5 0 h 28
C 3 2 f 56
C 6 7 a 2
```

```
t 0 1 NN,EE,NN,NE,NE,EE,SS
t 4 0 EE,NO,OO,SO,OO
L 4 0 1 3
L 9 9 2 0
L 5 0 5 0
L 2 4 1 5
q
```

- Output_1:

```
[
0 1 j 47
0 2 e 71
1 2 f 48
1 3 o 43
2 4 m 49
3 5 o 66
4 5 c 50
4 4 g 6
]
4
1
2
```

- Input_2: testo il comportamento della propagazione delle regole su una sola piastrella, a partire dall'inserimento di una nuova regola, ed utilizzando anche l'operazione di stato

```
C 3 5 x 96
C 1 3 h 90
C 2 1 s 44
C 2 3 o 78
C 0 5 f 6
C 0 2 a 46
C 5 1 i 45
```

```
C 4 1 i 27
C 0 1 f 18
C 2 2 d 59
C 5 5 u 69
C 0 3 t 13
C 6 2 l 10
C 5 2 o 50
```

```
C 5 3 u 20
r b 2 i 1 l
r a 2 i
r s 1 h 1 d
r u 1 h
? 5 2
? 4 1
```

```
s
p 5 2
p 2 3
p 4 1
? 5 2
? 4 1
q
```

- Output_2:

```
o 50
i 27
(
b: 2 i 1 l
a: 2 i
s: 1 h 1 d
u: 1 h
)
b 50
i 27
```

- Input_3: testo le funzionalità di spegnimento e di stampa

```
C 1 2 j 51
C 5 2 u 55
C 4 2 n 50
C 0 2 v 3
C 3 4 i 95
C 2 0 d 65
C 5 4 p 40
```

```
C 3 0 c 49
C 2 1 a 3
C 0 1 h 19
C 3 2 s 8
C 4 4 a 32
C 2 2 k 53
C 5 0 s 73
```

```
C 5 5 i 41
C 1 1 c 13
S 3 2
S 2 0
? 3 4
? 2 0
q
```

- Output_3:

```
i 95
```

- Input_4: testo le funzionalità di propagazione a blocco e di ordinamento delle regole dopo averle usate

```
C 2 1 j 90
C 1 5 y 75
C 1 3 l 75
C 2 2 o 19
C 5 0 b 84
C 2 0 x 49
C 3 0 f 54
```

```
C 1 2 q 12
C 1 4 t 85
C 3 1 z 56
C 4 0 f 90
C 5 1 z 32
C 2 3 i 20
C 4 4 l 2
```

```
C 5 5 u 40
C 6 3 a 200
C 0 0 i 1
C 4 5 a 10
r a 2 f
r b 1 f 1 z
r c 1 q 1 j
```

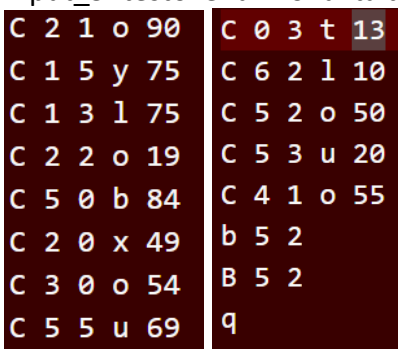
```
r d 2 y
r e 1 t 1 q
s
P 5 0
o
s
q
```

- Output_4:

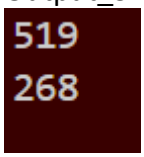
```
(
a: 2 f
b: 1 f 1 z
c: 1 q 1 j
d: 2 y
e: 1 t 1 q
)
```

```
(
d: 2 y
a: 2 f
c: 1 q 1 j
e: 1 t 1 q
b: 1 f 1 z
)
```

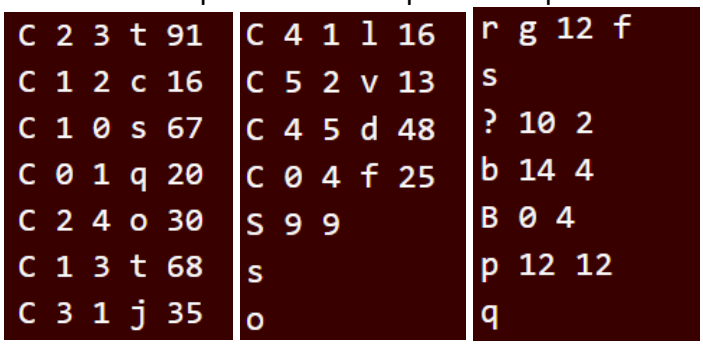
- Input_5: testo le funzionalità di calcolo dell'intensità di un blocco e di un blocco omogeneo



- Output_5:



- Input_6: testa alcuni casi limite delle varie funzionalità →
  - spegni una piastrella già spenta → non esegue nulla;
  - stampa le regole quando non ce ne sono → stampa le due parentesi vuote;
  - ordina le regole quando non ce ne sono → non esegue nulla;
  - stato di una piastrella spenta → non stampa nulla;
  - calcola intensità di un blocco che parte da una piastrella spenta → stampa 0;
  - calcola intensità di un blocco omogeneo che ha una sola piastrella al suo interno → stampa il solo valore di quella piastrella;
  - propaga il colore ad una piastrella spenta → non esegue nessuna propagazione ma comunque inizializza la piastrella spenta in memoria;



- Output_6: