Galliano Andrea 05460A - Relazione Progetto "Piastrelle Digitali"

Indice

Introduzione

Per poter affrontare ragionevolmente il problema, sono state utilizzate apposite strutture ed algoritmi che risolvessero tutti i punti richiesti e che rappresentassero fedelmente il piano descritto all'interno della traccia.

Strutture utilizzate

Il piano

Per poter rappresentare fedelmente il **piano** contenente le **piastrelle** a cui poter applicare le **regole di propagazione**, è stato necessario utilizzare una struttura che, per avere a disposizione tutte le informazioni necessarie alla memorizzazione delle **piastrelle**, avesse un campo che mettesse in relazione le coordinate intere naturali ***(x, y)*** di una piastrella e i dati relativi all'*intensità con cui è accesa* ed il *colore*.
Per questo motivo, il primo campo del **piano** è il *puntatore all'indirizzo di memoria di una* ***mappa*** *dalla piastrella alla corrispondente colorazione*.
Il secondo campo della struttura è invece il *puntatore all'indirizzo di una slice*

di regole, che torna utile nel momento in cui si decide di applicare una **regola di propagazione** a una o più piastrelle, modificandone il colore nel caso in cui la regola o le regole applicate venissero soddisfatte.

```go
type piano struct {
    piastrelle *map[piastrella]colorazione
    regole     *[]regolaSingola
}
```

Le piastrelle

Le **piastrelle**, dalla specifica, sono delimitate da 4 punti, ma per rappresentarle univocamente, è bastato riprodurle tramite una struttura i cui campi sono 2 *interi* (ovvero le coordinate ***(x, y)*** della piastrella all'interno del piano).

```go
type piastrella struct {
    x int
    y int
}
```

La colorazione

Come abbiamo visto per la prima struttura, per ogni **piastrella** accesa facente parte del **piano**, è necessario avere a disposizione altri 2 dati oltre le sue coordinate: l'*intensità* con cui è accesa nel **piano** ed il *colore*; queste informazioni è possibile salvarle grazie ad un'apposita struttura che è stata chiamata **colorazione**, con un campo intero ed una stringa (tornerà particolarmente utile anche nella memorizzazione delle **regole di propagazione**, i cui singoli addendi sono formati da un coefficiente intero ed una stringa rappresentante un colore).

```go
type colorazione struct {
    coefficiente int
    colore       string
}
```

Le regole

Le **regole di propagazione** da poter applicare alle **piastrelle accese** nel **piano** necessitano di 3 campi per poter essere rappresentate con una struttura: gli *addendi* che formano la regola, il *colore* che assume la **piastrella** dopo l'applicazione della regola ed il *consumo* (ovvero il numero di volte che la regola è stata applicata; questo campo permette di **ordinare** le regole in maniera **non decrescente**).

```go
type regolaSingola struct {
    addendi      []colorazione
    coloreFinale string
    consumo      int
}
```

Le funzioni principali

Gli algoritmi e le funzioni implementate all'interno del programma, assumendo che l'*input* sia **sempre** conforme a quello della specifica, permettono di modificare il **piano** e prestando particolare attenzione all'uso delle **risorse sia spaziali che temporali**.

Colora

```go
func colora(p piano, x int, y int, alpha string, i int) {
    // implementazione di "colora"
}
```

La funzione ***colora*** riceve come parametri il **piano**, le coordinate intere **x** e **y**, il **colore** e l'**intensità** con cui si intende colorare la *piastrella*.
Per effettuare l'operazione di *colorazione*, viene assegnata alla mappa contenente le *piastrelle* nel **piano** il valore della corrispondente **colorazione**.
Consideriamo quest'operazione come **accensione di una piastrella nel piano**.

- **Analisi del tempo**: l'accesso alla mappa ha costo ***O(1)*** in termini di tempo.

- **Analisi dello spazio**: non viene allocato alcuno spazio, di conseguenza il costo in termini di spazio è costante e nell'ordine di ***O(1)***.

Spegni

```go
func spegni(p piano, x int, y int) {
    // implementazione di "spegni"
}
```

La funzione ***spegni*** permette di spegnere una piastrella che, al momento, si trova (accesa) all'interno del **piano** con intensità ≥ 1.
Per eseguire da codice questa operazione, ciò che viene fatto è un'*eliminazione della piastrella avuta per argomento tramite coordinate.*

- **Analisi del tempo**: Anche l'operazione di *delete* dalla mappa impiega tempo costante, di conseguenza la complessità temporale è nell'ordine di ***O(1)***.

- **Analisi dello spazio**: Come per la complessità temporale, anche l'uso dello spazio è costante: ***O(1)***.

Regola

```go
func regola(p piano, r string) {
    // implementazione di "regola"
}
```

La funzione ***regola*** permette, dati in ingresso il **piano** ed una **stringa**, di aggiungere una nuova regola all'interno del piano stesso.
Per poterlo fare, è necessario, in primo luogo, effettuare un *parsing* della stringa avuta per argomento, successivamente creare la regola (composta dai suoi 3 campi analizzati durante l'analisi della struttra ***"regolaSingola"***) e, infine, *aggiungere la regola appena creata alla slice di regole facenti già parti del piano*.

- **Analisi del tempo**: Per l'analisi temporale della funzione è necessario tenere conto di 2 macro-operazioni (le restanti operazioni possiamo ipotizzare impieghino tutte tempo costante ***O(1)***):

1. L'esecuzione della funzione ***Split***: complessità ***O(n)***, dove ***n = numero di caratteri della stringa avuta per argomento***;

2. Le iterazioni del ciclo *for* che scorre la *slice* di stringhe ritornata dalla stessa funzione ***Split***: ***O(m)***, con ***m = numero di elementi di args***; Concludendo, possiamo dire che la complessità in termini di tempo è pari a ***O(n) + O(m) = O(n)***, poiché $m \leq n$.

- **Analisi dello spazio**: Per l'analisi dello spazio occupato dalla funzione, partiamo con le variabili ***"nuovaRegola"*** ed ***"addendoRegola"***, che occupano spazio ***O(1)***; la *slice* ***addendi***, invece, cresce nell'ordine di ***O(8)*** (non posso **MAI** avere più di 8 addendi per ogni regola), mentre l'aggiunta della nuova regola alla lista di regole del **piano** possiamo ipotizzare occupi anch'essa ***O(1)***. L'operazione più onerosa è dunque collegata alla chiamata della ***Split***, che *crea una nuova slice di stringhe* in base alla lunghezza ***n*** di ***s*** (stringa passata per argomento): complessità pari a ***O(n)***.
  Conclusione: la complessità in termini di spazio è nell'ordine di ***O(n)***.

Stato

```
func stato(p piano, x int, y int) (string, int) {
    // implementazione di "stato"
}
```

La funzione **stato** *restituisce e stampa i valori relativi al colore e l'intensità della piastrella delle coordinate avute per argomento.*
Per farlo, assegno ad una variabile il valore della mappa contenente le piastrelle del piano e un'altra, di tipo *bool*, per stampare (e, conseguentemente, anche ritornare) **se e solo se quella piastrella esiste nel piano**.

- **Analisi del tempo**: Dal punto di vista del tempo, questa funzione è nell'ordine di ***O(1)***, poiché tutte le operazioni che effettua (ovvero la restituzione di un valore della *mappa di piastrelle*, di un valore *bool* che indichi se quel valore esiste, il controllo prima della stmpa e il ritorno finale di **colore** e **intensità** della piastrella) impiegano tempo costante.

- **Analisi dello spazio**: Anche lo spazio allocato, a livello di variabili dichiarate e memoria utilizzata, da parte di **stato** è nell'ordine di ***O(1)***.

Stampa

```
func stampa(p piano) {
    // implementazione di "stampa"
}
```

La funzione **stampa** mostra tutte le **regole** del **piano** nel seguente formato:
(
*coloreFinale **1**: coefficiente1 colore1 coefficiente2 colore2 ...*
*coloreFinale **2**: coefficiente1 colore1 coefficiente2 colore2 ...*
.
.
.
*coloreFinale **n**: coefficiente1 colore1 coefficiente2 colore2 ...*
)
Ciò che fa la funzione, a livello di codice, è *scorrere la slice di regole del **piano** e, per ognuna di essere scorrere gli addendi che la compongono stampando infine il coefficiente ed il colore dell'addendo* (separando opportunamente entrambi con uno spazio).

- **Analisi del tempo**: Questa funzione contiene 2 cicli: il primo scorre le **regole** nel **piano**, mentre il secondo scorre gli **addendi** di ogni regola. Il primo ciclo ha complessità ***O(n)*** (con n = *numero di regole nel piano*) ed il secondo effettua sempre, al più, 8 iterazioni (questo perché, per definizione del piano e dell'intorno di ogni piastrella con *max piastrelle circonvicine* = *8*, **una regola di propagazione non può avere più di 8 addendi**).
  Di conseguenza, la complessità temporale totale della funzione **stampa** è pari a ***O(n) x O(8)*** = **O(n)**.

- **Analisi dello spazio**: La complessità spaziale di questa funzione di stampa è costante ***O(1)***.

Blocco

```
func blocco(p piano, x, y int) {
    // implementazione di "blocco"
}
```

La funzione **blocco** stampa la somma delle intensità delle piastrelle facenti parte del medesimo blocco; per poterlo fare con complessità spaziali e temporali contenute, è stato necessario partire dalle coordinate ***(x, y)*** di una piastrella avuta per argomento per poi *effettuare una **visita in ampiezza ("Breadth-First-Search")** ed avere a disposizione le piastrelle circonvicine del blocco.*

La ricerca degli adiacenti o delle piastrelle circonvicine ad un'altra, le cui coordinate ***(x, y)*** sono passate per argomento ad un'apposita funzione ***"cercaAdiacenti"***, non fa altro che *scorrere tutte le possibili **8 combinazioni** di coordinate di piastrelle circonvicine per poi restituirle all'interno di una slice di piastrelle.*

```
func cercaAdiacenti(p piano, piastrella_ piastrella) []piastrella {
    // le 8 combinazioni possibili per ogni piastrella:
    combX := []int{-1, 0, 0, 1, -1, -1, 1, 1}
    combY := []int{-1, -1, 1, -1, 1, 0, 0, 1}

    // implementazione di "cercaAdiacenti"
}
```

Per effettuare la *visita in ampiezza*, è stata inoltre utilizzata una **coda**, in cui vengono salvate temporaneamente le piastrelle visitate e dalle quali si andrà a visitarne le circonvicine.
La struttura dati **coda**, con campi e funzioni scritte all'interno di un file a parte chiamato ***"queue.go"***, è definita così:

```
type queue struct {
    head *queueNode
    tail *queueNode
}

type queueNode struct {
    next  *queueNode
    value piastrella
}
```

La memorizzazione di ***tail*** è particolarmente utile all'interno della funzione di ***enqueue***, poiché permette di ***NON*** *scorrere tutta la coda per aggiungere un elemento, ma di avere direttamente un puntatore all'ultimo nodo ed effettuare l'aggiunta risparmiando in termini di complessità temporale.*

Verrà tenuto conto delle piastrelle già visitate salvandole permanentemente all'interno di una **mappa usata come *set***.
Dato che, però, all'interno del linguaggio **Go** *non esiste **"Set"** come vero e proprio tipo*, ecco come è stato realizzato:

```
visitate := make(map[piastrella]struct{})
```

Questa mappa, **da piastrella a struct vuota**, permette di memorizzare solo le chiavi, in modo tale da trattare la struttura dati come un vero e proprio *set di piastrelle già visitate durante la BFS.*
Viene utilizzata una *struct vuota* al posto di una *variabile di tipo bool* **per poter risparmiare ulteriormente spazio in memoria**.

- **Analisi del tempo**: Questa funzione, oltre alle istruzioni di tempo costante nell'ordine di ***O(1)*** (come gli accessi/aggiunte di elementi delle mappe, controllo che la coda sia o meno vuota e l'incremento del valore di *intensità totale del blocco*), ha le seguenti operazioni rilevanti per la corretta stima dei costi temporali:

1. La funzione ***"enqueue"***: la coda, grazie al campo con il puntatore a ***tail***, permette di effettuare l'accodamento in tempo costante ***O(1)***.

2. La ***ricerca degli adiacenti***: per ogni piastrella, questa ricerca comporta, nel caso peggiore, 8 iterazioni; a questo punto, è possibile affermare che la complessità temporale dell'operazione è ***O(8)***.

3. La ***BFS***: la *ricerca in ampiezza* presenta 2 cicli: il primo che si interrompe quando *la coda è vuota*, il secondo che *scorre tutte le piastrelle circonvicine all'elemento corrente della coda.* La complessità è dunque ***O(n + m)***, dove ***n = insieme di vertici nel grafo/insieme di piastrelle del blocco*** e ***m = archi che collegano i vertici/piastrelle***.

Concludendo, poiché ***m = 0***, la complessità temporale di **blocco** è ***O(n)***.

- **Analisi dello spazio**: Per analizzare lo spazio occupato da questa *ricerca in ampiezza*, teniamo conto delle seguenti strutture dati:

1. Il *Set* ***"visitate"***: dovendo memorizzare ogni piastrella che viene visitata, nel caso peggiore contiene ***n*** elementi totali del piano, quindi nell'ordine di ***O(n)***;

2. La *slice* ***"piastrelleBlocco"***, che, esattamente come il *Set*, può contenere fino a ***n*** elementi totali del piano, dunque anche in questo caso abbiamo una complessità spaziale ***O(n)***;

3. Lo stesso ragionamento è possibile estenderlo anche alla *coda*, che avrà anch'essa complessità spaziale ***O(n)***.
   Conclusione: considerando l'analisi appena fatta e che la *slice* ***"circonvicine"*** (della funzione ***"cercaAdiacenti"***) occupa **sempre** ***O(8)***, la complessità **totale** della funzione **blocco** è ***O(n)***.

Blocco Omogeneo

```
func bloccoOmog(p piano, x, y int) {
    // implementazione di "bloccoOmog"
}
```

La funzione **bloccoOmog** *stampa la somma delle intensità delle piastrelle circonvicine facenti parte dello stesso blocco, utilizzando lo stesso principio di funzionamento della funzione* ***blocco***.

Proprio allo scopo di *fattorizzare* la parte di implementazione comune a **blocco** e **bloccoOmog**, entrambe le funzioni utilizzano una funzione "comune" chiamata

"bloccoGenerico", *alla quale viene passato un parametro di tipo* ***bool*** *(omogeneo = True/False) e che ha una condizione che valuta quando incrementare il valore della somma delle intensità.*

```
func bloccoGenerico(p piano, x, y int, omogeneo bool) (int, []piastrella) {
    // implementazione di "bloccoGenerico"
}
```

A questo punto, è facile dedurre che sia le prestazioni riguardanti il *tempo* che quelle riguardanti lo *spazio* non variano rispetto alla funzione ***blocco***.
Consideriamo inoltre ***n = numero di piastrelle totali nel piano*** come nella funzione analizzata precedentemente.

- **Analisi del tempo**: Complessità temporale nell'ordine di ***O(n)***.

- **Analisi dello spazio**: Complessità spaziale nell'ordine di ***O(n)***.

Propaga

```
func propaga(p piano, x, y int) {
    // implementazione di "propaga"
}
```

La funzione **propaga** permette di applicare ad una piastrella, le cui coordinate ***(x,y)*** vengono passate per argomento, *la prima* ***regola di propagazione*** *disponibile dell'elenco di regole nel* ***piano***.
Ciò che viene fatto dalla funzione è seguire i seguenti passaggi:
1. Cercare le piastrelle circonvicine a quella avuta per argomento (utilizzando, come già visto per **blocco** e **bloccoOmog** la ***ricerca degli adiacenti***): ***O(8)*** e **mai** di più;
2. Scorrere tutte le **regole di propagazione** del **piano**: ***O(n)***, dove ***n = numero di regole nel piano***;
3. Scorrere tutti gli **addendi** della *regola corrente*: ***O(8)***, non è possibile avere più di 8 addendi a regola (vale infatti la stessa regola della *ricerca degli adiacenti*);
4. Scorrere tutti gli **adiacenti** trovati nel punto 1: anche in questo caso ***O(8)***;
5. Se le piastrelle adiacenti rispettano una **regola di propagazione**, *ne viene incrementato il consumo e salvata sia la piastrella a cui applicare la regola che la regola stessa*: operazione che impiega tempo costante, quindi ***O(1)***; 6. Infine, se nel punto precedente è stato salvato qualcosa, viene effettuata la colorazione della piastrella (con *intensità = 1* nel caso in cui fosse stata spenta, oppure con l'intensità invariata rispetto a com'era prima della chiamata di **propaga**): ***O(m)***, dove ***m = numero di elementi all'interno della mappa che contiene le piastrelle del piano***.

- **Analisi del tempo**: Avendo analizzato tutte le macro-operazioni svolte da **propaga**, è possibile dedurre che i costi temporali sono nell'ordine di ***O(n) + O(m)*** in linea generale, ma nel caso di **propaga**, dovendo applicare una **regola** a una sola **piastrella**, abbiamo ***m =***

1, di conseguenza il consumo delle risorse temporali è semplicemente ***O(n)***.

- **Analisi dello spazio**: per capire a pieno la complessità spaziale della funzione **propaga**, è necessario analizzare in dettaglio tutte le variabili che vengono allocate durante il suo funzionamento:

1. *Mappa **piastrelleRegole***: dopo essere stata inizializzata, conterrà, al massimo, una sola *entry*, quindi ***O(1)***;

2. *Slice **adiacenti***: restituisce, al più, 8 piastrelle, quindi ***O(8)***;
   Possiamo dunque stabilire che la complessità spaziale di **propaga** sia ***O(1)***.

Propaga Blocco

```
func propagaBlocco(p piano, x, y int) {
    // implementazione di "propagaBlocco"
}
```

La funzione **propagaBlocco** segue lo stesso principio di funzionamento di **propaga**, con la sola differenza che ***TUTTE** le piastrelle del blocco a cui appartiene le piastrella di partenza vengono analizzate per stabilire se applicarvi o meno una **regola** in base all'attuale ordinamento.*
Come fatto per **blocco** e **bloccoOmog** le operazioni comuni di **propaga** e **propagaBlocco** sono racchiuse all'interno della stessa funzione, chiamata **propagaGenerico**.

```
func propagaGenerico(p piano, x, y int) map[piastrella]regolaSingola {
    // implementazione di "propagaGenerico"
}
```

L'analisi dei costi di **propagaBlocco** è però leggermente diversa rispetto a **propaga**.

- **Analisi del tempo**: In questo caso, a differenza della funzione **blocco**, che permetteva di applicare una **regola di propagazione** ad una e una sola **piastrella**, è necessario considerare il caso generale e concludere che la complessità temporale è ***O(n + m)*** (ricordando che ***n = numero di regole nel piano***, mentre ***m = numero di elementi all'interno della mappa che contiene le piastrelle del piano***).

- **Analisi dello spazio**: Le risorse spaziali più rilevanti utilizzate da **propagaBlocco** sono quelle riguardanti la *slice* di *mappe **"sliceCambiamenti"*** e la *slice **"piastrelleBlocco"***, che conterranno entrambe, al più, ***n* piastrelle totali del piano**.
  Poiché tutte le altre variabili e strutture dati allocate e utilizzate dalla funzione occupano spazio costante ***O(1)***, si può affermare che la complessità

spaziale di **propagaBlocco** sia nell'ordine di ***O(n)***.

Ordina

```go
func ordina(p piano) {
    // implementazione di "ordina"
}
```

La funzione **ordina** permette di *ordinare le* ***regole di propagazione*** *del* ***piano*** *in ordine* ***non decrescente*** *in base al consumo delle regole stesse.* Per fare l'ordinamento, è stata utilizzata la funzione di libreria di **Go** SortStableFunc, che permette di ordinare **in maniera stabile** riscrivendo il **comparatore** per confrontare gli elementi di una slice in modo analogo rispetto alla funzione SortFunc.

- **Analisi del tempo**: L'ordinamento delle regole in base al loro consumo è basato su confronti e, nel caso peggiore, non si può scendere al di sotto dell'ordine di ***O(n log n)***.

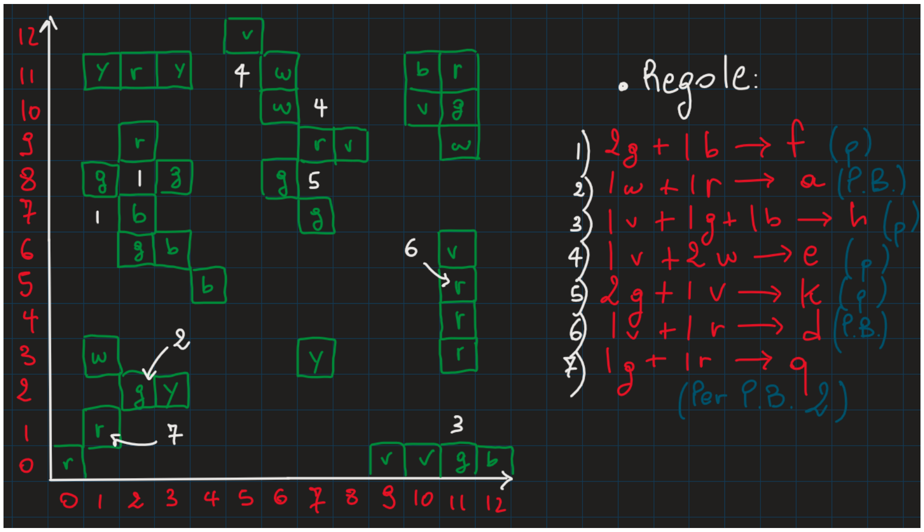
- **Analisi dello spazio**: Essendo un algoritmo di ordinamento ***in-place***, non utilizza spazio ulteriore per la creazione di copie di slice, di conseguenza la funzione **ordina** utilizza solo un **puntatore alla slice da ordinare** ed è nell'ordine di ***O(1)***.

Esempi di esecuzione

Per testare il corretto funzionamento del programma e le sue prestazioni, oltre all'esempio fornito dalla traccia, sono stati scritti ulteriori *file di input* con i relativi *file di output*.
Per questi esempi è stata inoltre creata una griglia per avere una **visualizzazione grafica** del piano per capire come viene modificato a fronte dei comandi in input.

Esempio piano 1

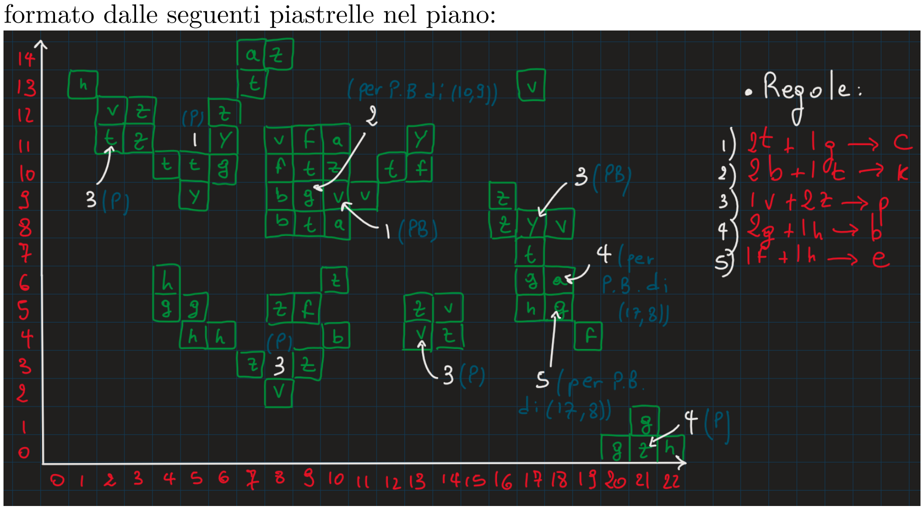
Per il primo esempio, il **piano** è composto dalle seguenti piastrelle e le seguenti **regole**:



I numeri e le frecce in bianco fanno riferimento alle regole applicate a quella piastrella, mentre le scritte (in blu) di fianco alle regole di propagazione indicano se viene applicato il comando per la chiamata della funzione **propaga** (p) o **propagaBlocco** (P).

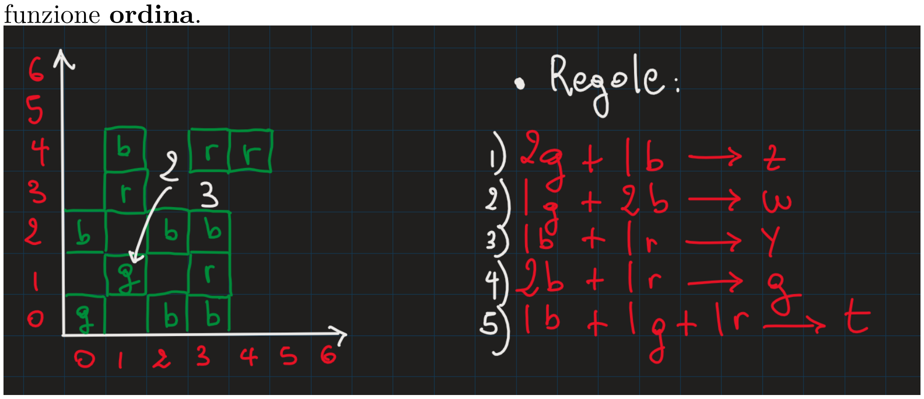
Esempio piano 2

Il secondo esempio, con un **piano** leggermente più grosso del precedente, è invece formato dalle seguenti piastrelle nel piano:



Esempio piano 3

Il terzo esempio ha lo scopo di testare in particolare le funzioni **propaga** e **propagaBlocco** dopo aver ordinato più volte le regole del **piano** attraverso la funzione **ordina**.



Nell'immagine d'esempio vengono riportate solo le prime 2 *propagazioni*, poiché le altre vengono effettuate in seguito a più ordinamenti delle regole.
Per vedere in maniera completa come cambia il **piano**, a fronte del *file di input*, ecco la tabella:

INPUT	OUTPUT
C 0 0 g 1	
C 0 2 g 1	
C 1 1 g 1	
C 1 3 g 1	
C 1 4 g 1	
C 2 0 g 1	
C 2 2 g 1	
C 3 0 g 1	
C 3 1 g 1	
C 3 2 g 1	
C 3 4 g 1	
C 4 4 g 1	
r z 2 g 1 b	
r w 1 g 2 b	
r y 1 b 1 r	
r g 2 b 1 r	
r t 1 b 1 g 1 r	
b 0 0	10
b 2 2	10
b 1 4	10
b 4 4	2
b 3 4	2
B 0 0	2
B 2 0	2
B 4 4	2
B 3 4	2

INPUT	OUTPUT
s	(
	z: 2 g 1 b
	w: 1 g 2 b
	y: 1 b 1 r
	g: 2 b 1 r
	t: 1 b 1 g 1 r
	)
p 1 1	
o	
s	(
	z: 2 g 1 b
	y: 1 b 1 r
	g: 2 b 1 r
	t: 1 b 1 g 1 r
	w: 1 g 2 b
	)
p 3 3	
o	
s	(
	z: 2 g 1 b
	g: 2 b 1 r
	t: 1 b 1 g 1 r
	y: 1 b 1 r
	w: 1 g 2 b
	)
P 1 1	
o	
s	(
	z: 2 g 1 b
	g: 2 b 1 r
	t: 1 b 1 g 1 r
	w: 1 g 2 b
	y: 1 b 1 r
	)
? 1 1	w 1
? 2 2	t 1
? 2 0	t 1
? 3 2	y 1
? 3 0	y 1
q	