```
[elia@arch-elia ~/go/src/brickProject/]$ ./brickproject
[elia@arch-elia ~/go/src/brickProject/]$ m elia cortesi 01911A
[elia@arch-elia ~/go/src/brickProject/]$ m algorithm exam brickProject
[elia@arch-elia ~/go/src/brickProject/]$ s brickProject
```

brickProject by Elia Cortesi

Indice:

1 – Introduzione
- 1.1 – Struttura del progetto
- 1.2 – Descrizione del problema

2 - Modellazione e implementazione
- 2.1 – Modellazione del problema
- 2.2 – Implementazione delle strutture dati
  - 2.2.1 – Implementazione di gioco
  - 2.2.2 – Implementazione di mattoncino
  - 2.2.3 – Implementazione di fila
  - 2.2.4 – Implementazione delle liste concatenate
  - 2.2.5 – Implementazione delle code

3 - Spiegazione delle funzioni
- 3.1 – Spiegazione delle funzioni ausiliarie
- 3.2 – Spiegazione di inserisciMattoncino
- 3.3 – Spiegazione di stampaMattoncino
- 3.4 – Spiegazione di disponiFila
- 3.5 – Spiegazione di stampaFila
- 3.6 – Spiegazione di eliminaFila
- 3.7 – Spiegazione di sottostringaMassima
- 3.8 – Spiegazione di indiceCacofonia
- 3.9 – Spiegazione di disponiFilaMinima

1 - Introduzione

1.1 – Struttura del progetto

```
brickProject/
│
├── input/
│   ├── input_disponi_fila_e_indice_cacofonia.txt
│   ├── input_fila_minima.txt
│   ├── input_inserimento_mattoncini.txt
│   ├── input_lungo.txt
│   └── input_sottostringa_massima.txt
│
├── output/
│   ├── output_disponi_fila_e_indice_cacofonia.txt
│   ├── output_fila_minima.txt
│   ├── output_inserimento_mattoncini.txt
│   ├── output_lungo.txt
│   └── output_sottostringa_massima.txt
│
├── controllaMattoncini.go
├── disponiFila.go
├── disponiFilaMinima.go
├── eliminaFila.go
├── gestioneCode.go
```

```
├── gestioneListe.go
├── indiceCacofonia.go
├── inserisciMattoncino.go
├── main.go
├── sottostringaMassima.go
├── stampaFila.go
└── stampaMattoncino.go
```

I file all'interno della directory *brickProject/input/* sono dei file di input in formato *.txt* che servono a simulare una serie di comandi per il programma.
I file all'interno della directory *brickProject/output/* sono dei file in formato *.txt* che contengono l'output atteso dei corrispettivi file di input.
I file all'interno della directory *brickProject/* sono tutti i file in formato *.go* che compongono il programma.
Per avviare il programma correttamente eseguire il seguente comando all'interno della directory principale del progetto:

```
[elia@arch-elia ~/go/src/brickProject/]$ go run .
```

E poi inserire manualmente a piacimento i comandi relativi al programma elencati alla fine di questa sezione.

Se si volesse utilizzare un input presente nella cartella *brickProject/input/* eseguire i seguenti comandi all'interno della directory principale del progetto:

```
[elia@arch-elia ~/go/src/brickProject/]$ go build .
[elia@arch-elia ~/go/src/brickProject/]$ ./main < input/input_a_piacere.txt
```

Se invece si volesse usare un input proprio spostare il file *.txt* nella cartella *brickProject/input* ed eseguire i comandi sopracitati con il nome del vostro file.

Lista dei comandi:
`m α β σ` : inserisciMattoncino (α, β, σ)
`s σ` : stampaMattoncino (σ)
`d ±σ1 ±σ2 . . . ±σn` : disponiFila (±σ1 , ±σ2 , . . . , ±σn)
`S σ` : stampaFila (σ)
`e σ` : eliminaFila (σ)
`f α β` : disponiFilaMinima (α, β)
`M σ τ` : sottostringaMassima (σ, τ)
`i σ` : indiceCacofonia (σ)
`c σ α1 α2 . . . αn` : costo (σ, α1 , α2 , . . . , αn)
`q` : termina l'esecuzione del programma

1.2 - Descrizione del problema

Il problema proposto è una simulazione di un gioco da tavolo ispirato al domino, in cui si manipolano mattoncini contenuti in una scatola per formare file su un tavolo. I mattoncini hanno i bordi sinistro e destro identificati da due forme ben definite. I mattoncini si concatenano in base alla compatibilità delle loro estremità. Si possono concatenare in due modi: in modo normale o invertendone uno dei due (o entrambi), l'importante e' che lo forme adiacenti dei bordi combacino. La formazione di una fila avviene concatenando due o piu' mattoncini presenti nella scatola. Ulteriori dettagli sul problema e sulle sue dinamiche saranno forniti nelle sezioni successive.

2 – Modellazione e implementazione

2.1 – Modellazione del problema

Analizzando attentamente la traccia fornita e le varie istruzioni che il programma deve elaborare, si possono fare alcune considerazioni importanti. Prima di tutto, la scatola in cui i mattoncini vengono inseriti e prelevati deve essere in grado di gestire un alto volume di richieste in modo efficiente. Per garantire ciò, è essenziale dotarla di un meccanismo che consenta di reperire i mattoncini richiesti rapidamente. Lo stesso principio si applica al tavolo. Entrambe queste strutture dati devono permettere l'aggiunta e la rimozione di mattoncini, o gruppi di mattoncini nel caso del tavolo, in modo efficiente e veloce. Ciò si può fare facilemente con delle *mappe*.

Inoltre, vi è una funzione particolare, `disponiFilaMinima`, che richiede di disporre sul tavolo una fila di mattoncini, da una forma ad un'altra, con la minor lunghezza possibile. Per affrontare questo problema, posso modellare i mattoncini come un *grafo* e utilizzare una delle tecniche di visita dei grafi per individuare questa fila minima. Il grafo in questione deve presentare alcune caratteristiche specifiche: non è orientato, poiché un mattoncino può essere concatenato in entrambi i versi, e non è pesato, poiché non esiste alcun criterio per favorire un mattoncino rispetto agli altri. Ulteriori dettagli saranno forniti nella sezione corrispondente.

Infine, la funzione `costo` richiede di manipolare le file di mattoncini, aggiungendo e rimuovendo elementi da un punto specifico della fila. A tale scopo, ritengo che le *liste concatenate* siano una scelta più opportuna rispetto agli *array*, sia in termini di efficienza che di gestione dello spazio.

EDIT: Successivamente alla fase finale del progetto, durante lo sviluppo della funzione `costo`, ho riscontrato che essa non opera direttamente sulle file di mattoncini, ma piuttosto sulle sequenze di forme dei mattoncini che compongono la fila. Pertanto, alcune funzionalità delle liste concatenate e le liste stesse non sono pienamente sfruttate. Nonostante ciò, a causa del momento avanzato della fase di sviluppo, ho deciso di mantenere l'implementazione con le liste concatenate senza fare modifiche.

2.2 – Implementazione delle strutture dati

2.2.1 – Implementazione di gioco

La struttura dati `gioco` rappresenta il gioco nella sua interezza, includendo sia la scatola sia il tavolo. In Go, è possibile implementarla con una *struct* contenente due campi: `scatola` e `tavolo`. Come discusso nella sezione 2.1, entrambe queste strutture devono essere efficienti e veloci negli inserimenti, nelle cancellazioni e nelle ricerche, e le *mappe* sembrano rispondere a questi requisiti. Le mappe sono strutture dati che associano una *chiave* a un *valore*. Le chiavi sono univoche, il che significa che non possono esistere due valori diversi nella mappa memorizzati con la stessa chiave.

La scatola rappresenta il contenitore di tutti i mattoncini e può essere implementata utilizzando una *mappa* in cui la chiave è una *stringa*, rappresentante il nome del mattoncino, e il valore è un *vettore* di due stringhe che contiene il bordo sinistro e il bordo destro del mattoncino (ulteriori dettagli nella sezione 2.2.2).

Il tavolo rappresenta il piano di gioco su cui verranno disposte le file e può essere implementato utilizzando una *mappa* in cui la chiave è una *stringa*, rappresentante il nome della fila, e il valore è il tipo `fila` (come descritto nella sezione 2.2.3).

In questo modo, le operazioni di inserimento, cancellazione e ricerca hanno un costo O(1), poiché sfruttano il meccanismo di ricerca chiave-valore, che è immediato rispetto alla scansione dell'intera struttura.

Per rendere l'utilizzo della funzione `disponiFilaMinima` più comodo all'interno di gioco, istanzio anche un altro campo chiamato `forme`. Questo campo rappresenta l'elenco di tutte le forme disponibili per i mattoncini e può essere implementato utilizzando una *mappa* in cui la chiave è una stringa, rappresentante la forma di un bordo di un mattoncino, e il valore è un *bool*, che può assumere il valore *true* o *false*. Tuttavia, per ottimizzare lo spazio, il tipo del valore è scelto in modo che occupi il minor spazio possibile: infatti non uso questa mappa per il valore booleano, ma sfrutto la comodità di ricerca e inserimento offerta dalla struttura dati mappa

2.2.2 – Implementazione di mattoncino

La struttura dati `mattoncino` rappresenta un singolo mattoncino. Un mattoncino è identificato da una tripla *m(alpha, beta, sigma)*, in cui `alpha`, `beta` e `sigma` sono *stringhe*. `alpha` e `beta` corrispondono ai bordi sinistro e destro del mattoncino, mentre `sigma` rappresenta il suo nome. È possibile utilizzare una *struct* con tre campi, tutti di tipo *string*.

Tuttavia, i mattoncini devono soddisfare alcune condizioni: non possono esistere due mattoncini con lo stesso nome e un mattoncino non può avere il bordo sinistro e destro uguali tra loro.

2.2.3 – Implementazione di fila

La struttura dati `fila` rappresenta una concatenazione di mattoncini. I mattoncini possono concatenarsi in due modi: in modo normale *beta-alpha* o in modo inverso *beta-beta*/*alpha-alpha* (invertendo il secondo/primo). In Go, è possibile implementarla con una *struct* contenente due campi: `componenti` e `indiceCacofonico`.

Il campo `componenti` rappresenta la fila in sé, ovvero la concatenazione di tutti i mattoncini, e può essere implementato utilizzando una *lista concatenata* (come descritto nella sezione 2.2.4).

Il campo `indiceCacofonico` rappresenta l'indice di cacofonia del nome di una fila (come definito nella sezione 3.8) e può essere implementato con un *int*.

2.2.4 – Implementazione delle liste concatenate

Come spiegato nella sezione 2.2.3, ho implementato la concatenazione effettiva dei mattoncini utilizzando una *lista concatenata*. In Go, il package `container/list` fornisce una serie di funzioni per creare e gestire le *liste*. Tuttavia, ho preferito implementare tipi e funzioni per la creazione e la gestione delle liste da zero, per adattarle specificamente al mio problema e avere un controllo completo sul loro funzionamento. Questa implementazione è contenuta nel file `brickProject/gestioneListe.go`.

All'interno di questo file, definisco il tipo `listNode`, che rappresenta un *nodo* della lista. Ha quattro campi: `next` e `prev`, *puntatori* a `listNode`, `segno`, un *byte* (per ulteriori dettagli, vedi sezione 3.4), e `data`, di tipo `mattoncino`. `next` e `prev` indicano rispettivamente il nodo successivo e il nodo precedente al nodo corrente, rendendo la lista *doppiamente concatenata*. Questa caratteristica offre il vantaggio di poter essere percorsa facilmente sia dall'inizio sia dalla fine, oltre a garantire un range più ampio di operazioni rispetto alla lista concatenata semplice. `data` rappresenta il valore contenuto nel nodo della lista, e è di tipo `mattoncino`, con i campi descritti nella sezione 2.2.2.

Oltre a `listNode`, definisco il tipo `linkedList`, che rappresenta una l*ista doppiamente concatenata*. `linkedList` ha due campi: `head` e `tail`, entrambi *puntatori* a `listNode`. `head` rappresenta il primo nodo della lista, mentre `tail` rappresenta l'ultimo.

Successivamente, definisco delle funzioni per la creazione e la gestione delle liste:

- `newList`: inizializza una nuova lista (complessità *O(1)*).
- `newNode`: inizializza un nuovo nodo (complessità *O(1)*).
- `addNode`: aggiunge un nodo in coda alla lista (complessità *O(1)*).
- `searchNode`: restituisce il nodo corrispondente a un mattoncino con un determinato nome (complessità *O(n)*, dove *n* è la lunghezza della lista).
- `printList`: restituisce una stampa della fila secondo il formato specificato nella sezione 3.5 (complessità *O(n)*, dove *n* è la lunghezza della lista).

Utilizzando queste funzioni, ho il completo controllo del mio codice e sono sicuro delle operazioni che esegue. Inoltre, posso modificare una funzione o un tipo secondo le mie esigenze, ad esempio come nella definizione del campo `segno` in `listNode` (sezione 3.4).

2.2.5 – Implementazione delle code

Nell'algoritmo di `disponiFilaMinima` (vedi sezione 3.9) utilizzo una *coda* come struttura ausiliaria. Per lo stesso motivo per cui ho scelto di implementare da zero `brickProject/gestioneListe.go`, ho deciso di creare anche `brickProject/gestioneCode.go`.

All'interno di questo file, definisco due tipi principali: `queue`, che rappresenta una struttura dati *coda*, e `queueElement`, che rappresenta un elemento della coda. `queue` ha due campi: `head` e `tail`, entrambi un *puntatore* a `queueElement`. `head` rappresenta l'elemento presente nella coda da più tempo, `tail` rappresenta l'elemento presente nella coda da meno tempo.

Successivamente, definisco delle *funzioni* e dei *metodi* per creare e gestire le code. Ho scelto di utilizzare anche i metodi perché mi sembravano più leggibili e per variare rispetto alle funzioni normali delle liste. Le funzioni e i metodi definiti includono:

- `newQueue`, che inizializza una nuova coda.
- `newQueueElement`, che inizializza un nuovo elemento di una coda.
- `enqueue`, un metodo che inserisce un elemento nella coda.
- `dequeue`, un metodo che rimuove un elemento dalla coda.
- `isEmpty`, un metodo che restituisce *true* o *false* a seconda se la coda contiene o meno elementi.
- `bottom`, un metodo che restituisce l'elemento presente da più tempo nella coda.
- `top`, un metodo che restituisce l'elemento presente da meno tempo nella coda.

Tutte queste funzioni hanno complessità di *O(1)*.

Come per le liste, questo codice mi consente di gestire efficacemente le code all'interno del mio progetto, garantendo un controllo completo sulle operazioni eseguite e adattando la struttura alle specifiche del mio problema.

3 – Spiegazione delle funzioni

3.1 – Spiegazione delle funzioni ausiliarie

Durante lo sviluppo di questo progetto, mi sono trovato più volte a dover affrontare operazioni che, nel lungo termine, avrebbero aumentato la complessità e diluito il codice. Per risolvere questa situazione, ho deciso di creare un file `brickProject/controllaMattoncini.go` contenente funzioni ausiliarie utili per eseguire specifiche operazioni.

La funzione `controllaScatola` serve per verificare se un mattoncino è presente nella scatola. Utilizza il costrutto `if exists`, il che le conferisce una complessità *O(1)*.

Per quanto riguarda `controllaTavolo`, ho riscontrato alcune difficoltà: essa itera prima sulla mappa `tavolo` per controllare tutte le file e, per ciascun nome di fila, esegue un *ciclo di controllo* interno. Attualmente, la complessità di questa funzione è *O(n*m*k)*, dove *n* è la lunghezza del nome della stringa, *m* è la lunghezza del nome del mattoncino da cercare e *k* è il numero delle file. Tuttavia, va considerato che nel caso peggiore tutti i mattoncini potrebbero collegarsi tra loro formando un'unica fila. In questo caso, la complessità della funzione sarebbe *O(n*m)*, con *n* che rappresenta il numero di mattoncini e *m* la lunghezza del nome del mattoncino da cercare (in quanto il parametro *k* si annullerebbe, essendoci una sola fila).

Nella funzione ho utilizzato `strings.Contains`. Tuttavia, anche dopo aver esaminato il codice sorgente di questa funzione, posso solo stimarne la complessità come *O(n*m)*, dove *n* è la lunghezza della stringa in cui si effettua la ricerca e *m* è la lunghezza della stringa da cercare.

La funzione `trovaFilaSulTavolo` è essenzialmente una replica di `controllaTavolo`, ma restituisce il *nome* della fila e la lista effettiva dei mattoncini. La complessità rimane identica a quella di `controllaTavolo`.

Infine, `verificaFila` è una funzione che verifica se una concatenazione di mattoncini è valida per formare una fila. La sua complessità è *O(n)*, dove *n* è il numero di mattoncini da controllare; nel caso peggiore, *n* è uguale al numero totale di mattoncini nella scatola. La funzione utilizza uno *switch case* per controllare il segno di due mattoncini adiacenti e valutare i loro bordi in base a quelli.

3.2 – Spiegazione di inserisciMattoncino

La funzione `inserisciMattoncino` consente di aggiungere un mattoncino alla scatola. Viene invocata con il comando `m alpha beta sigma`, dove i tre parametri sono tutti di tipo *string,* e inserisce un mattoncino denominato `sigma`, con bordo sinistro `alpha` e bordo destro `beta`. Se il mattoncino supera i controlli di legittimità, viene inserito nella mappa `scatola` e le forme dei suoi bordi vengono aggiunte alla mappa `forme`. Questa funzione ha complessità *O(1)* perché all'interno utilizza la funzione `controllaScatola` e esegue tre inserimenti in una *mappa*.

3.3 – Spiegazione di stampaMattoncino

La funzione `stampaMattoncino` consente di stampare a schermo un mattoncino secondo il formato specificato:

`sigma: alpha, beta.`

Viene invocata con il comando `s sigma`. Può stampare un mattoncino presente sia nella scatola sia in una fila disposta sul tavolo. Nel secondo caso, che rappresenta il caso peggiore, la complessità massima della funzione è *O(n*m*k)*, dove:

- *n* è la lunghezza del nome della fila in cui è presente il mattoncino,
- *m* è la lunghezza del nome del mattoncino da stampare,
- *k* è il numero di file presenti sul tavolo.

3.4 – Spiegazione di disponiFila

La funzione `disponiFila` consente di disporre una fila sul tavolo e viene invocata con il comando `d ±σ1 ±σ2 . . . ±σn`. È composta da due parti: una di controllo e una di effettiva disposizione. Innanzitutto, verifica se i mattoncini esistono e se formano una fila. Una volta superati i controlli, la fila viene creata con la funzione `creaFila` (complessità *O(n)*), in cui vengono creati i nodi corrispondenti ai mattoncini, aggiunti alla lista e successivamente rimossi dalla mappa `scatola`, poiché una volta disposti sul tavolo non sono più presenti nella scatola.

In `disponiFila`, cosi come in altre funzioni, ho utilizzato la funzione `strings.Split`, che restituisce una *slice* di stringhe a partire da una stringa, separandola in determinati punti in cui è presente un carattere separatore specifico. Ciò semplifica notevolmente l'ottenimento dei mattoncini. Dopo aver controllato il codice open source della funzione, ho dedotto grossolanamente che il costo massimo possibile è $O(n^2)$, con *n* uguale alla lunghezza della stringa di input. La complessità del resto della funzione è $O(m^2)$, con *m* uguale alla lunghezza della slice risultante da `strings.Split`. Tuttavia, *m* è sempre minore di *n*, poiché almeno un carattere separatore viene trovato (altrimenti l'input non sarebbe valido). Di conseguenza, la complessità massima è $O(n^2)$, con *n* uguale alla lunghezza della stringa di input.

3.5 – Spiegazione di stampaFila

La funzione `stampaFila` viene invocata con il comando `S sigma`, e permette di stampare a schermo la fila contenente il mattoncino chiamato `sigma`. Il formato di stampa è il seguente:

```
(
γ1
γ2
..
.
γn
)
```

dove `γn` è un mattoncino. Prima viene effettuato un controllo per verificare se il mattoncino `sigma` esiste, sia nella scatola sia disposto in una fila sul tavolo. Successivamente si scorre la lista per stamparla. La complessità è *O(n*m*k)*, con *n* che rappresenta la lunghezza del nome della fila, *m* la lunghezza del nome del mattoncino e *k* il numero di file disposte sul tavolo. Nella funzione viene utilizzata anche `printList`, che ha complessità *O(n)*, con *n* uguale alla lunghezza della lista.

Tuttavia, poiché il nome della fila viene ottenuto concatenando tutti i nomi dei mattoncini che la compongono, se i nomi dei mattoncini sono di lunghezza unitaria (quindi formati da un solo carattere), allora la lunghezza della lista sarà uguale alla lunghezza del nome della fila. In tutti gli altri casi, la lunghezza del nome è maggiore della lunghezza della lista. Questo è il motivo per cui la complessità è *O(n*m*k)* e non *O(n)*.

3.6 – Spiegazione di eliminaFila

La funzione `eliminaFila` viene invocata con il comando `e sigma` e permette di rimuovere la fila contenente il mattoncino di nome `sigma` dal tavolo e di reinserire i mattoncini che la compongono all'interno della scatola. La funzione controlla se il mattoncino esiste, sia nella scatola sia in una fila sul tavolo, e poi procede a reinserire nella mappa `scatola` il mattoncino con i suoi bordi, e nella mappa `forme` le forme dei bordi. Infine, elimina dalla mappa `tavolo` la fila specificata. La complessità di questa funzione è, come per `stampaFila`, *O(n*m*k)*, con *n* uguale alla lunghezza del nome della fila, *m* uguale alla lunghezza del nome del mattoncino e *k* uguale al numero di file disposte sul tavolo. I motivi sono gli stessi di `stampaFila` (sezione 3.5).

3.7 – Spiegazione di sottostringaMassima

La funzione `sottostringaMassima` è una funzione che permette di trovare la sequenza di caratteri in comune più lunga tra due stringhe, e viene invocata con il comando `m sigma tau`, dove `sigma` e `tau` sono di tipo *string*. Un dettaglio importante è che i caratteri in comune devono essere nello stesso ordine in entrambe le stringhe, ma non devono per forza essere contigui. Ad esempio, la sottostringa massima tra *casa* e *cosa* è `csa`.

La funzione sfrutta la tecnica della *programmazione dinamica* per risolvere questo problema di ottimizzazione. Questa tecnica consiste nel suddividere il problema generale in *sottoproblemi* più piccoli, risolvendoli uno alla volta e *memorizzandone* i risultati per evitare di ricalcolarli. Per fare ciò, viene utilizzata una *matrice* ausiliaria di dimensioni *(n+1)*(m+1)*, dove *n* e *m* sono le lunghezze delle due stringhe di input. Le prime righe e colonne vengono inizializzate a zero per facilitare i *cicli for* e per evitare errori di `index out of bounds` (oltre che indicare la sottostringa in comune tra una stringa *s* e una stringa vuota, ovvero 0).

La matrice contiene il numero minimo di passaggi necessari per trasformare una stringa nella sua sottostringa corrispondente nell'altra. La ricerca della sottostringa massima avviene attraverso due *cicli for* che confrontano i caratteri delle due stringhe. Quando i caratteri sono uguali, il valore nella cella corrispondente viene incrementato di 1 rispetto alla cella diagonale in alto a sinistra. Quando i caratteri sono diversi, il valore viene aggiornato con il massimo tra i valori delle celle adiacenti in alto e a sinistra. Il risultato finale è memorizzato nell'ultima cella in basso a destra.

Per ricostruire la sottostringa massima, si parte dalla cella finale della matrice e si controlla se il suo valore è stato ottenuto dall'incremento della cella diagonale superiore sinistra, indicando che i caratteri corrispondenti delle due stringhe sono uguali. Se è così, il carattere corrispondente viene aggiunto alla sottostringa massima e si procede alla cella diagonale superiore sinistra. Se il valore nella cella attuale è stato ottenuto dall'aggiornamento dalla cella superiore, ci si sposta in alto nella matrice; altrimenti, ci si sposta a sinistra. Si continua questo processo fino a quando non si raggiunge la cella in alto a sinistra, indicando la fine della ricostruzione della sottostringa massima. Il risultato sarà la sottostringa massima tra le due stringhe originarie.

La complessità di questa funzione è *O(n*m)*, dove *n* è la lunghezza della prima stringa e *m* è la lunghezza della seconda stringa.

3.8 – Spiegazione indiceCacofonia

La funzione `indiceCacofonia` è invocata con il comando `i sigma` e trova *l'indice di cacofonia* di una fila contenente il mattoncino di nome `sigma`. L'indice di cacofonia è definito come la *somma* delle lunghezze di tutte le sottostringhe massime tra ogni coppia di mattoncini che compongono la fila. Ad esempio, se il nome della fila è *casacosacane*, l'indice di cacofonia è

la somma della lunghezza della sottostringa massima tra *casa* e *cosa* (`csa` = 3) e tra *cosa* e *cane* (`ca` = 2), quindi l'indice di cacofonia è 5. La complessità della funzione è principalmente determinata da due parti: la parte di controllo e il calcolo effettivo dell'indice di cacofonia. La parte di controllo ha una complessità *O(n*m*k)*, dove *n* è la lunghezza del nome della fila, *m* è la lunghezza del nome del mattoncino e *k* è il numero di file. Questa parte controlla la presenza dei mattoncini necessari per calcolare l'indice di cacofonia. La parte di calcolo ha una complessità *O(s*r*l)*, dove *s* è la lunghezza della prima stringa, *r* è la lunghezza della seconda stringa e *l* è la lunghezza della lista. È importante notare che la lunghezza della lista è sempre minore o uguale alla lunghezza del nome della fila, mentre i nomi dei mattoncini possono essere considerati dello stesso ordine di grandezza. Infine, più lunghi sono i nomi dei mattoncini, più lungo sarà il nome della fila. Di conseguenza, possiamo semplificare dicendo che la complessità totale è *O(n*m*k)*, poiché la parte di calcolo è di ordine inferiore rispetto alla parte di controllo e non influisce significativamente sulla complessità complessiva della funzione.

3.9 – Spiegazione di disponiFilaMinima

La funzione `disponiFilaMinima` è invocata mediante il comando `f sigma tau` e permette di disporre sul tavolo la fila di lunghezza minima tra le due forme `sigma` e `tau`. Per implementare questa funzione, come scritto nella sezione 2.1, ho adottato un approccio basato su *grafi*, rappresentando le forme di mattoncini come *nodi* e i mattoncini stessi come *archi*. Ciò consente l'utilizzo di una BFS (*Breadth-First Search*) per visitare il grafo e trovare il percorso più breve tra due forme, ovvero il *cammino minimo tra due vertici*. Successivamente, è possibile ricostruire la sequenza di forme procedendo *a ritroso* nella mappa dei predecessori.

Per implementare questa funzione, ho utilizzato diverse strutture dati. Innanzitutto, ho modellato il grafo utilizzando una *lista di adiacenza*, dove ogni forma è associata a una lista delle forme adiacenti con cui può collegarsi. In aggiunta a ciò, ho impiegato una *coda* (come descritto nella sezione 2.2.5), essenziale per l'algoritmo BFS, una *mappa string-bool* per tenere traccia dei nodi già visitati e una *mappa string-string* per tenere traccia del predecessore di ciascun vertice.

Occorre distinguere due casi principali: quando la forma iniziale coincide con la forma finale e quando non coincide. Iniziamo con il caso in cui le forme sono diverse: qui si procede normalmente, eseguendo la BFS per trovare il cammino minimo tra le due forme. Successivamente con le funzioni `ricostruisciSequenza` e `creaListaNomi` si ottiene l'elenco dei mattoncini che compongono la fila

Nel caso in cui le forme iniziali e finali coincidano, si possono avere due situazioni distinte:

1. La fila è composta da due soli mattoncini con forme speculari rispetto alla radice del grafo (la forma iniziale). In questo caso, la sequenza di forme che costituisce la fila minima sarà formata dalla radice, la forma adiacente e nuovamente la radice. Per farlo ho implementato una funzione `checkCasoSemplice` che controlla se esiste una forma adiacente alla radice che permette di disporre una fila. La parte di controlo di questa funzione serve per fare in modo che i segni dei mattoncini siano concordi (infatti se sono discordi la fila non si può formare)
2. La fila è composta da più di due mattoncini. Qui si procede *forzando* l'algoritmo a trovare una strada alternativa, rimuovendo il collegamento diretto, all'interno della lista di adiacenza, tra la forma adiacente corrente e la radice. A questo punto calcolo il cammino minimo tra ogni forma adiacente alla radice e alla radice stessa, selezionando il più corto. Infine riaggiungo il collegamento eliminato in precedenza ripristinando lo stato iniziale della lista di adiacenza. Questo approccio consente di implementare una singola funzione per trovare il cammino minimo tra due nodi diversi, rendendo il codice più *modulare* e versatile.

Questa funzione risulta essere la più costosa in termini di complessità all'interno del mio progetto, e ciò è dovuto principalmente a due fattori:

- La funzione `append`: questa funzione viene utilizzata per aggiungere un elemento a un *array*. Sebbene possa sembrare che aggiungere elementi all'array sia un'operazione senza problemi, in realtà il funzionamento di questa funzione è simile a quello che un linguaggio di livello più basso dovrebbe fare ogni volta. Se la capacità dell'array è al massimo, la funzione *allocherà* nella memoria uno spazio equivalente alla lunghezza dell'array più uno, e poi *copierà* il contenuto del vecchio array nel nuovo. Di conseguenza, se la capacità massima non è ancora stata raggiunta, la complessità della funzione `append` è *O(1)*; altrimenti è *O(n)*, dove *n* è la lunghezza dell'array da copiare.

- La funzione `contains`: questa funzione è utilizzata all'interno di un'altra funzione, `popolaListaAdiacenza`. La funzione `contains` verifica se un elemento è contenuto all'interno di un array, e quindi ha complessità *O(n)*, dove *n* è la lunghezza dell'array. Eliminando il controllo con la funzione contains all'interno di `popolaListaAdiacenza`, potrei risparmiare tempo e ridurre la complessità totale di `disponiFilaMinima` a *O(n*m)*, dove *n* è il numero di forme e *m* è il numero di mattoncini. Tuttavia, la scelta tra dare priorità al tempo o allo spazio è critica. Se si decide di dare priorità al *tempo*, si rischia di avere *duplicati* all'interno delle varie liste di adiacenza. Anche se non sarebbe un problema, poiché all'interno della funzione `bfsNormale` si tiene traccia dei nodi visitati con un elenco predecessori, si potrebbe avere spazio occupato inutilmente. D'altra parte, se si decide di dare priorità allo *spazio*, c'è il rischio che il programma abbia una complessità maggiore del previsto. Considerando che la lunghezza media di ogni lista di adiacenza è molto inferiore rispetto al numero totale di forme, ho deciso di mantenere il controllo con la funzione contains per un'esecuzione più controllata e ordinata.

```
[elia@arch-elia ~/go/src/brickProject/]$ q
```