Commento al Laboratorio n. 2

Esercizio n.1: **Individuazione di regioni**

L'esercizio propone un problema verifica/selezione su una matrice. Visto che non è ancora stato trattato il passaggio di parametri by reference, i valori di ritorno delle funzioni `leggiMatrice` e `riconosciRegione` sono 2 `struct` che contengono rispettivamente il numero di righe e colonne letto da file (`dim`) e la quaterna base, altezza, ascissa e ordinata dell'estremo superiore sinistro del rettangolo riconosciuto (`rett`).

Il problema è di ottimizzazione e richiede cha siano enumerate tutte le regioni. La matrice viene scandita casella per casella e la funzione `riconosciRegione` riconosce un rettangolo verificando unicamente che l'angolo in alto a sinistra sia una casella nera senza un'altra casella nera a sinistra o sopra. Per il riconoscimento dei rettangoli si sfrutta l'assunzione che i rettangoli siano corretti, cioè che non ci siano forme diverse nella matrice. Ci si limita quindi a misurare i due lati adiacenti al vertice in alto a sinistra di un rettangolo. Una volta che una regione è stata riconosciuta si aggiornano le informazioni relative alla regione a massima altezza, massima base e massima area.

Esercizio n.2: **Azienda di trasporti**

Il problema è essenzialmente di strutture dati in cui memorizzare le informazioni lette da file per visualizzarle su richiesta, in quanto l'elaborazione di dette informazioni è minima. Si noti che esiste un sottoproblema specifico, la gestione delle date, che si potrà incontrare anche in esercizi successivi e che quindi vale la pena di trattare appositamente.

Gestione delle date: le date possono essere gestite come `struct` o come intero (che fa in modo di associare a ogni giorno un numero intero diverso, tale da rispettare i criteri di confronto). Si noti che le date (e più in generale il tempo) sono informazioni composte (aggregati) di giorno, mese e anno; nel caso specifico a queste si aggiungono anche ora, minuti e secondi. **Si tratta in effetti di un problema di codifica**. Per operazioni (aritmetiche o di confronto) su date è possibile:

- realizzare funzioni che in modo esplicito gestiscano le singole componenti,
- oppure convertire una data in giorni (un tempo in secondi), come valore intero e lavorare su questo.

Indipendentemente dal modo di lavorare, una data può essere rappresentata come `struct`, come stringa o come un unico numero. La scelta dipende spesso dal tipo di operazioni che è necessario effettuare:

- se è sufficiente ricordare e visualizzare una data, una stringa può essere la soluzione più semplice. Una `struct` è più versatile se è necessario visualizzare campi in modo separato o in formati diversi a seconda dei casi.
- se occorre effettuare operazioni, si sconsiglia la stringa, mentre può essere equivalente la rappresentazione (compatta) come un unico intero, oppure come `struct`. Si tenga presente che:
  - se si effettuano solo confronti tra date, non è necessario garantire la contiguità tra gli interi che rappresentano le date (non è necessario, ad esempio, che il 1 luglio sia rappresentato dal numero corrispondente al 30 giugno + 1), è sufficiente garantire la relazione di ordine. Ad esempio, `dataInt = aa*10000+mm*100+gg`
  - se si vuole effettuare aritmetica (ad esempio poter aggiungere o togliere un intero arbitrario a una data), allora è necessario garantire la contiguità delle codifiche intere (tenendo conto dei giorni in ogni mese e degli anni bisestili. Occorre inoltre spesso una funzione di decodifica.

Si propongono 2 soluzioni: entrambe rappresentano la data come `struct`, ma la prima (`confrontaDate1`) realizza i confronti operando sui campi della `struct`, la seconda (`confrontaDate2`) trasforma le informazioni dei campi in un intero.

Strutture dati: per la rappresentazione interna dei dati si usa come tabella una `struct` che contiene il numero di voci `n_voci` nel file di log e un vettore `log` di `struct` tipo `voce_t` avente campi stringa per `codice`, `partenza`, `destinazione`, `data_str`, `orap_str`, `orad_str`, `ritardo`, `data`, `p` e `d`. La data e le ore di partenza e arrivo sono lette come stringhe e poi le diverse informazioni che esse contengono sono estratte e memorizzate in `data`, `p` e `d`, apposite `struct` di tipo `data_t` e `ora_t`. La tabella viene acquisita nella funzione `leggiTabella`. La gestione dei comandi è un problema di menu (fatto con utilizzo di un tipo `enum`, si veda ad esempio "Dal problema al programma" 4.4.1).
La selezione e stampa dei dati richiesti è un problema di filtro dati, che comprende in questo caso la richiesta di informazioni aggiuntive (coppia di date, partenza, destinazione, codice). La funzione `selezionaDati` comprende quindi:

- l'acquisizione delle informazioni aggiuntive a seconda del comando
- l'iterazione sugli elementi della tabella con stampa del campo richiesto, eventualmente calcolandone il valore come nel caso del ritardo totale.

Esercizio n.3: **Occorrenze di parole**
Analisi: si tratta di un problema di selezione, applicato a un problema di elaborazione testi. Il problema di elaborazione testi consiste nel riconoscimento delle parole all'interno di un testo (il file `testo.txt`). Il problema di selezione consiste nel verificare, per ogni parola, se essa include una delle sequenze presenti nel file `sequenze.txt`. Siccome si tratta di cercare all'interno di un file dati corrispondenti a chiavi lette da un altro file, si pone il problema di decidere se ed eventualmente quali dati immagazzinare in vettori, nonché come organizzare le iterazioni sui dati. In linea di principio, sono possibili due tipi di iterazioni:

1) iterazione esterna sulle sequenze: per ogni sequenza iterare su tutte le parole (del file `testo.txt`) selezionando quelle corrispondenti alla sequenza
2) iterazione esterna sulle parole (del file `testo.txt`): per ogni parola, iterare sulle sequenze, determinando quelle che eventualmente corrispondono alla parola corrente.

Detto `np` il numero di parole e `ns` il numero di sequenze, entrambe le strategie hanno complessità *O(ns*np)*. In mancanza di altri vincoli o criteri di scelta, entrambe sono valide. Nel problema proposto, tuttavia, l'output viene fatto raggruppando i dati per sequenze (e non per parole), il che può privilegiare strategie del tipo 1). Va poi considerato l'eventuale utilizzo di vettori (per immagazzinare sequenze, testo su cui cercare e/o parole di questo testo), ed il tutto va inoltre visto alla luce della opportunità di intercalare o meno input, elaborazione ed output.

Si propongono per completezza tre tra le varie soluzioni possibili:

a) una prima, che evita l'utilizzo di vettori, grazie all'integrazione completa di input, elaborazioni ed output, con iterazioni di tipo 1). Tale soluzione richiede tuttavia letture multiple dello stesso file (`testo.txt`), il che è in genere sconsigliato visto il costo superiore (in termini di tempo) dell'accesso a file rispetto all'accesso a dati interni (in memoria). La soluzione viene presentata per completezza e per la sua semplicità
b) la seconda soluzione segue lo schema iterativo 1) della prima, cioè iterazione esterna sulle sequenze e iterazione interna sulle parole. La lettura multipla del file `testo.txt` viene evitata scorporando l'input di tale file, che viene fatto come prima operazione, immagazzinando l'elenco delle parole in un vettore (`parole`, una matrice di caratteri).

Anziché percorrere più volte il file `testo.txt`, si percorrerà quindi più volte il vettore `parole`. Questa soluzione deve affrontare il problema del dimensionamento della matrice `parole`. Infatti, mentre è nota la lunghezza massima di una parola, non si sa quante siano (al massimo) le parole nel testo. Siccome non si utilizzano ancora soluzioni con allocazione dinamica di vettori, occorre in questo caso sovradimensionare, in modo opportuno il numero di righe (di parole) nella matrice. Nella soluzione proposta si utilizza la costante `MAX_NP`, definita come 100

c) la terza soluzione esplora l'idea di scorporare l'output rispetto alle elaborazioni, e quindi di memorizzare in modo opportuno gli elenchi di parole corrispondenti a ogni sequenza. A tale scopo si utilizza un vettore di sequenze, viste ora come tipi `struct indice_t`, contenenti, per ogni sequenza, la stringa, nonché un vettore di parole (e della relativa posizione nel testo), per il quale è necessario, anche in questo caso, prevedere un sovradimensionamento.

Completamento delle specifiche: viste le considerazioni precedenti, per le soluzioni b) e c) è necessario sovradimensionare il numero di parole (in totale oppure per ogni sequenza). La soluzione b) prevede una costante `MAX_NP`. Per la soluzione c) le specifiche indicano un numero massimo di parole inizianti con un dato prefisso (`MAX_MATCH`) da considerare. Eventuali parole in eccesso vengono scartate.

Riconoscimento delle parole: si tratta di un sotto-problema di elaborazione testi (oppure di un semplice problema di codifica), per il quale non è possibile limitarsi all'input standard del C, ma sono necessarie alcune elaborazioni, quali il riconoscimento di inizio e fine delle parole, nonché i confronti che ignorino la differenza tra maiuscole e minuscole. Occorre elaborare le stringhe a livello di singoli caratteri. Si propongono due soluzioni, una (nelle soluzioni a e b) che effettua input a caratteri, una seconda (nella soluzione c) che effettua input per righe, elaborando successivamente le righe di testo, carattere per carattere, sul vettore in sui sono state acquisite .

Struttura dati: Oltre a variabili scalari e vettori di caratteri per singole stringhe, la soluzione b) prevede la matrice di caratteri `parole`, per acquisire le parole contenute nel testo, mentre nella soluzione c) il contenuto del file `sequenze.txt` viene memorizzato in un dizionario implementato come vettore non ordinato `indice`, di dimensione `N`, di strutture di tipo `indice_t` ciascuna contenente un prefisso `incipit` di al massimo `MAX_INC` caratteri, la sua lunghezza `lung_inc`, il numero di parole `n_match` con quel prefisso e il vettore `match` delle parole e delle loro posizioni. Quest'ultimo, di dimensione `MAX_MATCH`, contiene strutture di tipo `match_t` ciascuna contenente una parola `str` di al massimo `MAX_PAROLA` caratteri e un intero `pos` che ne identifica la posizione nel testo. La funzione `leggiSequenze` legge dal file il numero di prefissi dizionario e li memorizza uno alla volta nel vettore di `struct indice`.

Algoritmo:
Il `main` si occupa delle dichiarazioni e delle chiamate delle funzioni di primo livello

Soluzione a): il `main` apre e legge direttamente il file delle sequenze, per o`gequenze`ni sequenza chiama la funzione `cercaParola`, che legge il file `testo.txt`, acquisendo e riconoscendo le parole, mediante la funzione `leggiParola`

Soluzione b): lo schema è simile alla soluzione a), cui si aggiunge la funzione `leggiParole`, chiamata inizialmente dal `main`, per caricare, nella matrice parole, le parole contenute in `testo.txt`

Soluzione c): visto lo schema di esecuzione diverso dai precedenti, il `main` chiama `leggiSequenze`, per acquisire le sequenze da ricercare, successivamente itera sulla lettura, riga per riga del file `testo.txt`, con contestuale chiamata alla funzione `processaRiga`. La funzione `processaRiga` per ogni carattere della riga:

- se è alfanumerico (funzione `isalnum`):
  - aggiorna la posizione nel testo della nuova parola
  - identifica la parola corrente e la sua lunghezza mediante la funzione `trovaParola`
  - poi ricerca nel dizionario delle sequenze se c'è una sequenza inclusa nella parola corrente mediante la funzione `cercaSeq` e avanza l'indice nella riga di tante lettere quanto è lunga la parola corrente (-1 per compensare l'incremento a seguire del `for`).

La funzione `cercaSeq` ricerca con costo *O(n)* se nel dizionario c'è una sequenza inclusa nella parola corrente. Se si trova corrispondenza, e non si eccede il numero massimo di match `MAX_MATCH`, la parola e la sua posizione vengono memorizzate nel dizionario in corrispondenza della sequenza.

In tutte le soluzioni si usa la funzione `stristr` che, indipendentemente da maiuscole e minuscole, verifica se una stringa è inclusa nell'altra. La stampa, per le soluzioni a) e b) non corrisponde esattamente all'esempio, in quanto si riporta un messaggio anche nel caso di sequenze senza corrispondenza. Si potrebbe modificare in modo semplice l'output stampando la sequenza quando si trova la prima corrispondenza, anziché per ogni sequenza cercata.

Esercizio n.4: **Valutazione di algoritmi di ordinamento**

Si introducono le variabili `cnti`, `cntj` e `cntswap` per conteggiare le iterazioni del ciclo esterno, del ciclo interno e il numero di scambi. Nel caso di insertion e shell sort più che il numero di scambi si conteggia il numero di assegnazioni, escludendo quelle in cui si riassegna lo stesso valore. Nello shell sort si esplicita anche il numero di iterazioni del `for` esterno in funzione di `h` (calcolabile a priori come `h*(n/h -1) + n%h`). Si osservino i seguenti comportamenti:

- nell'insertion sort il numero di confronti (pari al numero di passi del ciclo interno) varia tra un caso migliore (0 per la IV sequenza che è già ordinata in ordine ascendente) e un caso peggiore (n*(n-1)/2 = 30*29/2 =435) (0 per la V sequenza che è già ordinata ma in ordine discendente). La complessità asintotica di caso peggiore è quindi $O(n^2)$
- nel selection sort il numero di confronti (pari al numero di passi del ciclo interno) è fisso e corrisponde al caso peggiore n*(n-1)/2 = 30*29/2 =435. La complessità asintotica di caso peggiore è quindi $\Theta(n^2)$, mentre per gli scambi è O(n)
- si osservi che nello shell sort il numero di passi del ciclo interno è considerevolmente ridotto rispetto a quanto avviene nell'insertion sort.