Commento al Laboratorio n. 1

Esercizio n.1: **Valutazione di espressioni regolari**

La soluzione proposta si propone di scostarsi il meno possibile dalla manipolazione di stringhe fatta con funzioni di libreria quali `strlen`, `strcmp` e `strstr`. Anche se non richiesto dalle specifiche, si presenta un main che opera con la stessa filosofia della `strstr`: si cerca `regexp` in `src`, se la si trova si identifica una stringa `exp` che inizia con `regexp`, la si stampa, poi si riparte da una stringa `src` che inizia dal carattere successivo a quello iniziale della stringa `exp`. In questo modo si possono cercare `regexp` eventualmente sovrapposte. Per evitare la sovrapposizione si può cercare nella stringa `src` che inizia dal carattere di `exp` distante da quello iniziale di tante posizioni quanto la lunghezza di `regexp`.
La funzione richiesta `cercaRegexp` si ispira alla `strstr` e si basa su un'iterazione che scandisce la stringa `src`, partendo da ognuno dei caratteri di questa (eccetto gli ultimi che sono pari alla lunghezza di `regexp`) a confrontare la relativa sottostringa con `regexp` tramite la funzione `regexpCmp`.

La funzione `regexpCmp`, simile alla `strcmp`, fa un confronto tra stringa ed espressione regolare. Lo schema è simile alla `strcmp`, con applicazione del principio dei quantificatori in verifica, atto a trovare un'eventuale differenza tra un carattere e un metacarattere (chiamando la funzione `regexpCmpChar`). A differenza della `strcmp`, il risultato è binario (vero/falso) in quanto non interessa in questo caso un confronto con esito maggiore/minore.

La funzione `regexpCmpChar` confronta carattere e metacarattere e ritorna, oltre a un risultato logico `res` (carattere e metacarattere corrispondono), l'indicazione `n` di quanti caratteri effettivi occupi un metacarattere (parentesi e/o altri simboli inclusi). Essa riceve un carattere e un puntatore all'inizio di un'espressione regolare (o a una sua sottostringa). Basandosi su un costrutto `switch-case`, tratta in modo opportuno il metacarattere, riconosce la corrispondenza (o meno) al carattere da verificare, ed infine conta (e ritorna, nel parametro per riferimento/puntatore `np`, il numero di caratteri occupati). Il numero di caratteri occupati potrà essere utilizzato dal programma chiamante sia nel caso di conteggio dei metacaratteri (ignorando l'esito del confronto) sia nel caso di confronto tra stringa ed espressione regolare, al fine di aggiornare l'inizio di una sottostringa di espressione regolare, per passare al prossimo metacarattere.

La funzione richiesta `regexpLen` si ispira alla `strlen` e conta i metacaratteri in un'espressione regolare. Essa utilizza la funzione `regexpCmpChar`,

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: **Azienda di trasporti – ordinamento**
Struttura dati: la `struct` per l'esercizio 2 (tipo `tabella_t`) viene estesa a comprendere un campo per il criterio di ordinamento corrente, il cui valore è selezionato in un tipo `chiaveOrdinamento` definito per enumerazione (`NESSUNO`, `DATA`, `CODICE`, `PARTENZA`, `ARRIVO`). Si tratta a tutti gli effetti di una `struct` wrapper (cfr. *Puntatori e strutture dati dinamiche 5.3*) che contiene informazioni non necessariamente omogenee tra loro.

Si osservi la tabella utilizzata nell'esercizio 2 era una `struct`, uno dei cui campi era un vettore. La `struct` veniva passata alle diverse funzioni by value, quindi ricopiata, con evidente costo in termini sia di tempo, sia di memoria. Si propongono, in aggiunta alla soluzione-base, 2 soluzioni rese più efficienti dai passaggi by reference:

1. si disaccoppiano le informazioni raggruppate nella `struct` (vettore, sua dimensione e criterio di ordinamento), passandole quando necessario separatamente come parametri. Poiché il passaggio come parametro di un vettore è sostanzialmente *by reference* si evita la ricopiatura
2. la `struct` viene passata non *by value*, bensì emulando il passaggio *by reference* ricopiando il puntatore alla `struct` stessa.

Menu: invece di utilizzare un tipo definito per enumerazione per i comandi, nella soluzione proposta si utilizzano esplicitamente gli interi come indici di un vettore di stringhe che li contiene.

Ordinamento: si utilizza l'insertion sort per realizzare un ordinamento stabile. La funzione di confronto `confrontaVoci` tiene conto della chiave di ordinamento richiesta per identificare i campi da confrontare e le funzioni di confronto (`comparaData`, `comparaOra` o `strcmp`).

Ricerca: per decidere se applicare una ricerca lineare o dicotomica si testa secondo quale chiave è eventualmente ordinato il vettore. Anche se non richiesto dalle specifiche, si realizza una funzione di ricerca per codice, per la quale sia la ricerca lineare, sia quella dicotomica sono standard.
La ricerca parziale è fatta per prefissi della stringa, quindi a partire dal primo carattere, mediante la funzione `strncmp`. Essa richiede di conoscere il numero di caratteri per il confronto. Al tal scopo si ricorre alla funzione `strlen`.
La ricerca in base a una stazione di partenza può trovare più di un risultato. Se è richiesto di elencarli tutti:

- nel caso di utilizzo della ricerca dicotomica, si trova una qualunque delle stazioni di partenza corrispondenti, dalla quale (se trovata) si reperiscono le altre mediante determinazione di un intervallo: se presenti, altre stazioni di partenza corrispondenti al criterio sono adiacenti, nel vettore ordinato, alla stazione di partenza trovata
- nel caso di ricerca lineare, si itera su tutto il vettore, stampando il risultato non appena trovato la stazione di partenza o parte di essa.

Esercizio n.4: **Azienda di trasporti - multiordinamento**
Si modifica la soluzione dell'esercizio precedente introducendo nella `struct` wrapper 4 vettori `logC`, `logD`, `logP`, `logA` di puntatori a elementi di tipo `voce_t` che serviranno a mantenere gli ordinamenti per codice, data/ora, partenza e arrivo. Il campo della chiave secondo cui è correntemente ordinato il vettore non serve più e viene eliminato. I 4 vettori di puntatori sono allocati, inizializzati e ordinati nella funzione `leggiTabella`.
Nella funzione `stampa` si accede ai dati del singolo elemento mediante il puntatore contenuto in uno dei 4 vettori a seconda che si voglia una stampa ordinata per codice, data/ora, partenza o arrivo.
La funzione `ordinaStabile` è un insertion sort che sposta i puntatori in funzione del confronto effettuato tra i dati cui essi puntano.
Le funzioni di ricerca dicotomica per codice e per cognome operano sui vettori di puntatori `logC` e `logP` per sfruttare l'ordinamento. Per la ricerca dicotomica per stazione di partenza parziale valgono le considerazioni fatte per l'esercizio 3.

Per questo esercizio si presentano una soluzione-base con la `struct` wrapper passata by value e una più efficiente dove il passaggio è by reference, evitando così la fase di ricopiatura. Non si presenta la soluzione in cui le informazioni sono passate singolarmente e non all'interno di una `struct` wrapper in quanto esse sono alquanto numerose e il loro passaggio singolarmente renderebbe il codice poco leggibile.