Commento al Laboratorio n. 4

Esercizio n.1: **Massimo Comun Divisore**

La condizione di terminazione si raggiunge o quando `a` è uguale a `b` oppure quando `b` è 0. In entrambi i casi si ritorna `a`.

Si identificano 4 casi:

1. `a` e `b` sono entrambi pari: 2 è certamente un fattore primo, quindi il massimo comun divisore di `a` e `b` è il doppio del massimo comun divisore di `a/2` e `b/2`
2. `a` è dispari e `b` è pari: il fattore primo 2 di `b` non compare certamente nel massimo comun divisore di `a` e `b`, che quindi è il massimo comun divisore di `a` e `b/2`
3. dualmente `a` è pari e `b` è dispari: il fattore primo 2 di `a` non compare certamente nel massimo comun divisore di `a` e `b`, che quindi è il massimo comun divisore di `a/2` e `b`
4. `a` e `b` sono entrambi dispari: secondo l'algoritmo di Euclide basato sulla sottrazione il massimo comun divisore di `a` e `b` è il massimo comun divisore di `a-b` e `b`, ma `a-b` è certamente pari, quindi è applicabile il caso 3. Il massimo comun divisore di `a` e `b` è quindi il massimo comun divisore di `(a-b)/2` e `b`.

Esercizio n.2: **Elemento maggioritario**

Si segue un classico approccio *divide et impera* di tipo *divide and conquer* con a=2 e b=2, quindi con 2 sottoproblemi (vettore sinistro e vettore destro), ciascuno di ampiezza metà. La condizione di terminazione si raggiunge per sottovettori unitari, dove l'elemento maggioritario è chiaramente l'unico elemento del vettore. La ricorsione scende sui 2 sottovettori sinistro e destro. Se l'elemento maggioritario ritornato da entrambi è lo stesso, questo è anche il valore di ritorno della chiamata ricorsiva. Con una scansione di costo lineare si determina il numero di occorrenze nel vettore originale di ciascuno dei 2 elementi maggioritari ritornati dalle 2 chiamate ricorsive. Se uno di questi risultati supera la metà più uno del numero di elementi del vettore, allora si è identificato l'elemento maggioritario. Se per nessuno dei due risultati vale questa condizione, l'elemento maggioritario non esiste.

Equazione alle ricorrenze:

$$T(n) = 2T(n/2) + n \qquad n>1$$
$$T(1) = 1 \qquad n=1$$

con soluzione $T(n) = O(n\log n)$

Nota 1: si poteva in alternativa ordinare il vettore e poi contare il numero di occorrenze dei suoi elementi. Per soddisfare però i vincoli di complessità e di algoritmo in loco, l'unico ordinamento accettabile era il *quicksort*, nell'assunzione di considerare la complessità di caso medio e non di caso peggiore.

Nota 2: dalla letteratura è noto l'algoritmo di majority vote di Boyer-Moore. Esso è iterativo, ha complessità lineare e non richiede locazioni di memoria aggiuntive in numero dipendente dai dati (è in loco) e quindi soddisfa i vincoli.

Questo esercizio però ha come scopo impratichirsi nella risoluzione ricorsiva di problemi, per cui è auspicabile non ricorrere alle soluzioni della nota 1 e 2.

Esercizio n.3: **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.4: **Azienda di trasporti – ordinamento**
Struttura dati: la `struct` per la tabella del Lab. 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 nel Lab. 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 in base a una stazione di partenza parziale pone il problema di poter trovare più di un risultato. Per questo:

- 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.5: **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 4.
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.