Commento al Laboratorio n. 3

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

L'esercizio propone un problema verifica/selezione su una matrice. Il numero di righe e colonne letto da file dalla funzione `leggiMatrice` è ritornato come parametro by reference *(si usano quindi i puntatori)*, come pure la base e l'altezza del rettangolo riconosciuto dalla funzione `riconosciRegione`. Un rettangolo viene riconosciuto da una qualunque casella nera (la prima in alto a sinistra), in quanto appena riconosciuto, tutte le caselle del rettangolo vengono modificate (valore 2). La strategia, pur se leggermente meno efficiente di quella descritta nella soluzione dell'esercizio 1 del lab. 2, viene presentata per completezza e perché potrebbe consentire di estendere il problema al caso di rettangoli adiacenti. Il problema è di ottimizzazione e richiede cha siano enumerate tutte le regioni. Si mantengono i valori della regione che massimizza base, area e altezza e li si aggiorna ogni volta che viene individuata una nuova regione.

Esercizio n.2: **Puntatori e rappresentazione dati**

Premessa: la memoria contiene byte che, in opportuni gruppi, codificano dei dati (es. numeri `float`, `double` o `long double`). I singoli byte possono essere facilmente "osservati", considerandoli come numeri senza segno da 8 bit (tipo `unsigned char`), mediante puntatori (servono operazioni di cast o di passaggio intermedio a `void *`). *L'`unsigned` è opportuno quando si vogliono evitare i numeri negativi, ad esempio per vedere un byte come un intero nell'intervallo 0..255. Per accedere ai dati si può usare, quasi in modo indifferente, la notazione a puntatore, con aritmetica dei puntatori, oppure la notazione vettoriale con parentesi quadre e indici.*

Non si tratta dell'unica soluzione, si potrebbe usare il tipo `char`, ma occorerebbe tener conto dei numeri negativi, oppure si potrebbero trattare word, ad esempio di 4 byte, viste come `int` o `unsigned int`.

1) **Determinare endianness**

Per determinare se il sistema utilizzi *Little o Big Endian*, è sufficiente analizzare i byte di un dato per il quale sia possibile riconoscere la differenza tra il byte più significativo (MSB) e il byte meno significativo (LSB). Una strategia semplice, tra le altre, è quella di utilizzare una variabile di tipo `int` contenente il valore 1: in tale variabile l'unico byte diverso da 0 sarà il meno significativo (LSB). La funzione `checkBigEndian` utilizza una variabile intera (`int test`) a cui viene assegnato 1. Per osservare il byte meno significativo, si usa una variabile puntatore a `char` (`pchar`) a cui si assegna (con cast) l'indirizzo (il puntatore) a `test`. Il `char` puntato da `pchar` è quello di indirizzo basso (più piccolo): il sistema è quindi Big Endian se tale byte è 0.

2) **Determinare le dimensioni**

Si tratta semplicemente di applicare l'operatore `sizeof`. Si noti che, per il tipo `long double`, la dimensione potrebbe essere superiore a quella effettivamente utilizzata (in pratica è un caso di allineamento/padding).

Il numero di byte indicato dall'operatore `sizeof` può quindi risultare sovradimensionato in più casi, nei quali, ad esempio, l'elaboratore proposto lavori su "extended precision" da 80 bit. Se ci si limitasse quindi al `size` determinato con `sizeof`, non si individuerebbero correttamente, ad esempio, bit di segno ed esponente nella codifica Little Endian.

Un modo pragmatico di affrontare tale problema, una volta noto l'elaboratore e il software usato, consiste nel determinare empiricamente, ad esempio usando la funzione `stampaMemoria` (punto 4), la codifica effettivamente usata e usare questa informazione, in modo opportuno nella fase di stampa.
Il programma proposto mostra una funzione che automatizza il riconoscimento del padding, limitato alle architetture Little Endian (quasi tutte): la funzione `dimensioneConPadding`, ricevuta la dimensione totale (padding incluso) determinata con `sizeof`, calcola la dimensione netta (padding escluso). A tale scopo si "cerca" il bit di segno, facendo in modo che sia l'unico a differire, in due variabili con valori opposti: la funzione usa due variabili contenenti valori opposti (tali che differiscano per il solo segno) e itera su tali variabili, a partire dal byte meno significativo, alla ricerca del byte contenente il bit di segno (tutti gli altri sono identici). *Attenzione*, si sconsiglia di cercare il bit di segno partendo dal padding (indirizzi più alti) in quanto i bit di padding potrebbero essere arbitrari (non è garantito che contengano zeri e/o che siano uguali nei due dati).

3) **Acquisire i dati.**
Si usa una sola `scanf`, per la variabile `af`, assegnando poi il valore acquisito (come richiesto) alle variabili `ad` e `ald`. Si potrebbero acquisire tre valori diversi per le tre variabili. Si faccia tuttavia attenzione al fatto che su alcuni elaboratori, i compilatori (specie open source) potrebbero non gestire correttamente il tipo `long double` in input (si tratta di un problema di compatibilità del software con un dato processore).

4) **Stampare il contenuto della memoria**
Pur se ***non richiesta***, si è realizzata la funzione `stampaMemoria`, che visualizza (in formato esadecimale, con formato %02x, variante del formato esadecimale %x, tale da stampare almeno 2 byte, aggiungendo, se serve, uno o due zeri per riempire entrambe cifre) il contenuto della memoria associata a una variabile (tutta, incluso eventuale spazio per allineamento/padding). Visualizzare il contenuto della memoria è in definitiva il modo migliore per capire e/o verificare la codifica applicata. La funzione osserva il dato come se si trattasse di un vettore di `unsigned char`, visualizzato in due modi:

- un byte alla volta, per indirizzi crescenti, visualizzando per ogni byte sia l'indirizzo che il contenuto
- tutti i byte contigui su una sola riga, dal più al meno significativo (si noti l'uso dell'informazione bigEndian).

5) **Stampa dei bit, suddividendo le 3 parti: segno, esponente, mantissa**
Si noti (come specificato nel testo) che è sufficiente visualizzare i bit così come sono, NON è necessario interpretarli o decodificarli (non serve, ad esempio, gestire il fatto che il primo 1 della mantissa in alcuni casi venga nascosto e in altri sia esplicito).
Il problema principale consiste nel fatto che nè le memorie RAM, nè il linguaggio C consentono di indirizzare e *leggere/scrivere direttamente i bit*, per cui occorre un lavoro a due livelli: un primo livello che gestisce un ***vettore di byte*** e un secondo che riconosce/visualizza i ***bit all'interno di un byte***. Si propongono, a tale scopo, due soluzioni per la funzione `stampaCodifica`[1]

[1] si propone, per gestire la scelta tra `stampaCodifica1` e `stampaCodifica2`, una opportuna `#define` che associ il nome `stampaCodifica`, nella chiamata

- `stampaCodifica1`: la funzione sfrutta un vettore di `char`, usati ognuno per rappresentare un solo bit (è quindi sufficiente il tipo `char`, sottoutilizzato per i soli valori 0 e 1). A partire dal byte meno significativo, vengono analizzati i byte e per ognuno si generano gli 8 bit, proponendo tre strategie alternative[2]
    - `separaBit1`, ricevuti il byte, il vettore destinazione e l'indice da cui iniziare a scrivere i bit, decodifica il byte con una strategia simile ai problemi di codifica in *"Dal problema al programma", Cap. 4*. I bit di un byte vengono generati osservando se il numero, diviso iterativamente per 2, sia pari o dispari (mediante resto della divisione per 2)
    - `separaBit2` usa una strategia simile alla precedente[3], ma cambia gli operatori usati: la divisione per 2 viene fatta utilizzando l'operatore shift (>>) che, spostando dutti i bit a destra, realizza divisioni per potenze di 2 (in questo caso shift di una posizione, quindi divisione per 2). Per determinare il valore del bit meno significativo, anzichè l'operatore resto (`%2`) si isola il bit mediante un'operazione logica: *AND bit-a-bit ("bitwise")* tra gli 8 bit del byte osservato e la "maschera" `0x01` (`00000001` binario), contenente tutti 0 eccetto il bit meno significativo: tale AND ha due soli risultati possibili: `0x00` o `0x01` (gli interi `0 o 1`), che indicheranno, rispettivamente, 0 o 1 come bit meno significativo
    - `separaBit3` continua a usare operatori di shift e AND bit-a-bit, ma cambia strategia: anzichè muovere i bit nel byte, per portarli nella posizione meno significativa, facendo sempre AND con la stessa maschera, si tiene il byte così come è e si muove la maschera, che assumerà i valori (binari) `00000001, 00000010, …, 10000000`. In questo caso il risultato dell'operazione AND sarà 0, per indicare bit a 0, oppure una potenza di 2 (valore diverso da 0) per indicare bit a 1.

  Una volta accumulati i bit (al massimo 128) nel vettore `vb`, questi andranno visualizzati in tre sezioni: segno, esponente e mantissa. In pratica, l'unica grandezza che differenzia `float`, `double` e `long double` (a parte la dimensione complessiva, eventualmente corretta per il padding) è la dimensione dell'esponente. A tale scopo nella soluzione proposta si usa una variabile `nExpBits`, assegnata usando la funzione `nBitEsponente` (questa, a partire dalla dimensione (`size`), usa un costrutto `switch` per selezionare una tra le 3 possibili dimensioni dell'esponente).

- `stampaCodifica2`: la funzione evita il vettore `vb` di `char`, usato per accumulare i bit prima di stamparli, ma percorre direttamente i bit, come se fossero in un vettore virtuale di bit (mentre in realtà si tratta di una matrice di bit, in cui ogni riga corrisponde a un byte e le colonne sono 8), cui si accede mediante la funzione `estraiBit`. Questa funzione, ricevuto il vettore di byte e l'indice

---

fatta nel `main`, a una delle due. Ovviamente la scelta non può essere fatta in esecuzione, ma per cambiare occorre ricompilare il programma

[2] Anche in questo caso si utilizza una `#define` per la scelta della funzione.

[3] Questa strategia è di solito preferita dai programmatori in quanto gli operatori shift e logici sono più veloci rispetto agli operatori artmetici di divisione e resto.

(`iBit`) del bit desiderato (es. bit 30, inizio dell'esponente, di un `float`), calcola dove trovare tale bit, determinandone l'indice del byte (`iByte = iBit/8`) nel vettore e l'indice del bit all'interno del byte (`iBit%8`). L'indice `iByte` serve a selezionare la casella del vettore di byte, mentre `iBit%8` serve a costruire la maschera adatta a isolare/riconoscere il bit selezionato.

OSSERVAZIONE: mentre la funzione `stampaCodifica1` è strutturata su due cicli annidati, uno sui byte e un secondo sui bit all'interno di un byte, la funzione `stampaCodifica2` fa una sola iterazione su tutti i bit, risalendo tuttavia al byte e al bit all'interno di questo, mediante opportune operazioni (divisione e resto della divisione).