Tipi di Dato II: Stringhe e Vettori Bidimensionali

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2016-17

La presente dispensa introduce a problemi legati alle sequenze di caratteri (*stringhe*) e ai vettori bidimensionali, più comunemente noti come *matrici*.

1 Stringhe

Una stringa è essenzialmente un array di caratteri. Tuttavia esistono una serie di piccole peculiarità da conoscere.

Innanzitutto, una costante stringa può essere denotata tra i doppi apici `""`. Ad esempio, in C è legittima una dichiarazione del tipo: `char w[15]="paperino";` che è equivalente alla dichiarazione `char w[15]={'p','a','p','e','r','i','n','o','\0'}`: come vedete, come array di caratteri, una stringa ha un carattere *terminatore*, il carattere `'\0'` che viene automaticamente inserito quando una stringa viene indicata tra doppi apici. Il carattere `'\0'` è anche il valore carattere che valuta a `false`, e quindi è comodo per impostare la condizione di fine stringa nelle guardie dei cicli.

Potete sempre, avendo a che fare con parole, usare gli array di caratteri. Tuttavia, ci sono valide ragioni per usare le stringhe:

1. La libreria `string.h` contiene varie funzioni di utilità (vedremo nel seguito l'implementazione di alcune) che possono aiutare a scrivere alcuni programmi in modo veloce (se ne avete bisogno consultate un qualsiasi manuale di C).

2. C'è uno specifico formato di stampa e di lettura per le stringhe (come alcuni di voi sanno `scanf("%c",&c)` può risultare poco gestibile). Se `w` è una variabile stringa, posso stamparne il contenuto con `printf("%s",w)` e leggere una stringa con l'istruzione `scanf("%s",w)`. Osservate che **non** occorre scrivere in questo caso `&w`: infatti `w` è già un pointer, più precisamente un `char *`. Ricordo ancora che una stringa ha sempre un carattere in più (il terminatore) rispetto a quelli che voi esplicitamente scrivete.

3. Scrivendo funzioni sulle stringhe, in generale non serve passare parametri lunghezza, perchè la fine stringa viene riconosciuta grazie al terminatore.

Di converso, ci sono alcune cose a cui stare attenti:

1. se caricate una stringa "a mano", cioè carattere per carattere, ricordatevi di inserire alla fine il carattere terminatore. Altrimenti la maggior parte delle funzioni sulle stringhe non funzioneranno correttamente. Ad esempio, la `printf("%s",w)` continuerà a stampare il contenuto di tutte le celle di memoria che incontra (trasformandole in caratteri), fino a che non trova una cella che, casualmente contiene il codice ASCII 0 (cioè una cella di memoria che contiene il carattere '\0').

2. Essendo le stringhe vettori, *posso* fare l'assegnamento fra variabili stringa, ma questo produce semplicemente uno spostamento di puntatori. Per modificare il *contenuto* di una variabile, esiste una funzione di libreria per questo (`strcopy`), oppure occorre assegnare due stringhe carattere per carattere.

Fatte queste precisazioni nella prossima sezione ci divertiremo a scrivere qualche programmino sulle stringhe, vedendo possibili implementazioni di alcune tipiche funzioni di libreria.

1.1 Ordine Lessicografico

In alcune note precedenti, si è fatto un gran parlare di ordine lessicografico. L'ordine definito sulle stringhe è esattamente l'ordine lessicografico, e viene calcolato usando la funzione `strcmp`) che al solito torna -1 se la prima stringa è maggiore della seconda, 0 se sono uguali e 1 se la seconda è maggiore della prima. Eccovi una possibile implementazione (nello stile dei maestri, Kernigham e Ritchie ☺):

```
int myStrCmp(char* s, char* t){
   /* PREC: s!=t */
    while (*s==*t){
         if (!*s) return 0;
         s++; t++;
      }
    if (!s || *s<*t) return 1;
    return -1;
   }
```

Osservate che se dentro al ciclo, `*s` è il carattere terminatore (e quindi `!*s` valuta a `true`), allora anche `*t=='\0'` e quindi le due stringhe sono uguali. Attenzione, che la procedura vista sopra, assume che le due stringhe siano *diverse*, cioè occupino spazi di memoria diversi (altrimenti succede un bel pateracchio! cosa?). Volendo essere prudenti e farla funzionare anche su chiamate del tipo `myStrCmp(s, s)` oppure `myStrCmp(s, t)` dove `s` e `t` sono alias, forse è meglio iniziare con l'istruzione `if (s==t) return 0` (osservate, che in questo caso, confronto i *puntatori* e non i caratteri). Al solito, più prosaica la versione che usa le notazioni a vettore, piuttosto

che quella basata sull'aritmetica dei puntatori. In questo caso, però non si pone il problema che `s` e `t` siano alias (perché?):

```
int myStrCmp(char s[], char t[]){

   for (i=0; ; i++){
       if (s[i]=='\0'){
          if (t[i]=='\0') return 0;
                      else return 1;
          }
       if (t[i]=='\0') return -1;
       if (s[i]<t[i]) return 1;
       if (s[i]>t[i]) return -1;
      }
   }
```

1.2 Prefisso e Sottostringa

Poniamoci ora altri due tipici problemi. Il primo consiste nel determinare se una stringa è prefisso di un'altra. Torna 1 se è vero, e 0 altrimenti. Osservate che il corpo del `while` è il comando vuoto, e il suo scopo è semplicemente quello di trovare la più lunga sequenza comune tra `s` e `t`: se questa sequenza include tutta la stringa `s` viene tornato 1.

```
   int prefix(char* s, char* t)}{

      while (s && t && *s++==*t++);
      if (*s) return 1;
      return 0;
   }
```

La funzione sottostringa, invece verifica se `s` è sottostringa di `t` e torna, in caso affermativo, l'indice del carattere di `t` in cui comincia la copia di `s` contenuta in `t`. Viceversa torna -1. A ben pensare, una sottostringa altro non è che un prefisso a partire da un certo indice. Ecco come sfruttare questa idea, semplice ed efficace:

```
   int mySubStr(char* s, char* t){
       int i=0;

       while (t)
            if (prefix(s,t++)) return i;
       return -1;
   }
```

1.3 Trasformazione di una Stringa in Numero

Chiudiamo questo piccolo viaggio all'interno del mondo delle stringhe, con un altro piccolo problemino. Data una stringa di *cifre* costruire il numero intero rappresentato. Per risolvere questo problema, sfruttiamo il fatto che i caratteri che rappresentano cifre, hanno codifiche adiacenti, cioè se n 'e la codifica di `'0'`, allora `'1'` è codificato con $n+1$, `'2'` con $n+2$ e così via fino a `'9'` con $n+9$. Se la stringa contiene cifre, la funzione non è specificata e può quindi avere un comportamento *imprevedibile*. L'idea del programma seguente, è che se ho letto il numero n leggendo le prime k cifre, e c è la $k+1$–esima cifra, leggendo $k+1$ cifre leggo il numero $n*10+c$.

```
int myAtoi(char* s){
    int v=0;

    while (s) {
          v = v * 10 + s - '0';
          s++;
          }
    return v;
}
```

1.4 Esercizi e Spunti di Riflessione

1. Modificare la funzione `myStrCmp` in modo che valuti l'ordine lessicografico tra due stringhe considerando uguali una lettera minuscola e la sua versione maiuscola. [SUGG: consultare un manuale `C` e aiutarsi con funzioni di libreria sui caratteri tipo `islower`, `isupper`, `toupper`, `tolower`. Usare queste funzioni mette al riparo da possibili diverse codifiche che i caratteri hanno su macchine diverse].
2. Modificare la funzione `myAtoi` in modo che tenga conto di un eventuale segno iniziale.
3. Modificare la funzione `myAtoi` in modo che accetti un secondo parametro, `b` che rappresenta la base in cui è rappresentato il numero. Provare a farla funzionare anche per valori di $b > 10$ (almeno fino a 16, aiutandosi con funzioni tipo `isxdigit`).
4. Scrivere una funzione `myAtof` che legge una stringa che rappresenta un numero con il punto decimale e produce in output un valore di tipo `double`.
5. Scrivere una funzione `int myStrcspn(char* s, char* t)` che stabilisce la lunghezza del segmento iniziale di `s` che non contiene caratteri di `t`.
6. Scrivere una funzione `char* myStrpbrk(char* s, char* t)` che individua la prima occorrenza in `s` di un carattere di `t`, restituendo un puntatore a tale carattere. Se nessuno dei caratteri di `t` occorre in `s`, viene restituito `NULL`.

2 Matrici

Molti problemi hanno una struttura inerentemente *bidimensionale*. Pensate, ad esempio, alla soluzione di sistemi lineari dove i coefficienti sono naturalmente descritti da una *matrice*, cioè un dato in cui ciascun singolo elemento è identificato da due indici. Oppure pensate al gioco del Tris (o Filetto, o Tic-Tac-Toe), o al Forza4, o agli Scacchi. O ancora, pensate alla tabella dei voti in una classe. In tutti i linguaggi di programmazione è possibile definire *vettori a più dimensioni*. Noi, ci limiteremo a vedere problemi che si risolvono usando vettori a due dimensioni, le già nominate matrici. Analogamente a un vettore, una matrice si dichiara con:

```
T a[M][N];
```

dove `T` è un tipo, `a` è il nome della matrice, ed `M` ed `N` sono due costanti intere positive. Quella che viene dichiarata è una struttura bidimensionale con `M` righe ed `N` colonne. Nonostante la natura bidimensionale della matrice, gli elementi vengono allocati nella memoria *monodimensionale* del calcolatore, come descritto in Fig. 1. Nella figura, vengono anche affrontati in modo implicito alcuni interessanti problemi. In Fig. 1, la matrice è di tipo `int[][]` e dopo essere stata definita, è anche stata caricata con il seguente codice:

```
k=0;
for (int i=0; i<M; i++)
    for (int j=0; j<N; j++) a[i][j]=i*N+j;
```

in modo che il contenuto di ciascuna cella di memoria corrisponda al posto che tale cella avrebbe nell'ordine definito da una scansione *per righe* della matrice. Osservate, che tale posto è lo stesso che la cella ha all'interno della struttura monodimensionale che memorizza la matrice.

Osserviamo innanzitutto che il tipo di `a`, oltre ovviamente ad essere `T[][]`, è equivalente a `T**`, cioè il tipo di un *puntatore a un puntatore di* `T`. La cosa non dovrebbe sorprendere: l'operatore `[]` si applica a dei puntatori e per poterlo applicare due volte, il risultato della prima applicazione deve essere *ancora* un puntatore. `a[0]` è una scrittura leggitima, e significa `a+0`, cioè `a`, e, come nei vettori, rappresenta l'inizio dell'area di memoria dove è memorizzata la matrice. Osservate anche cosa significa `a[1]`: ancora una volta significa `a+1`, ma il numero di celle di memoria da sommare all'indirizzo di `a` *non è* `sizeOf(T)`, bensì `sizeOf(T)*N`: è come se gli elementi `a[0]`, `a[1]`, ..., `a[M-1]` fossero dei vettori di lunghezza `N` contenenti elementi di tipo `T`.

In generale quindi, l'indirizzo di un elemento `a[i][j]` sarà $a + (i \times N + j) \times$ `sizeOf(T)`. Questa formula è usata dal codice generato dal compilatore per indirizzare le celle di memoria di una matrice. Siccome è quindi necessario al compilatore conoscere il numero delle colonne N (che ovviamente influenza la *lunghezza di*

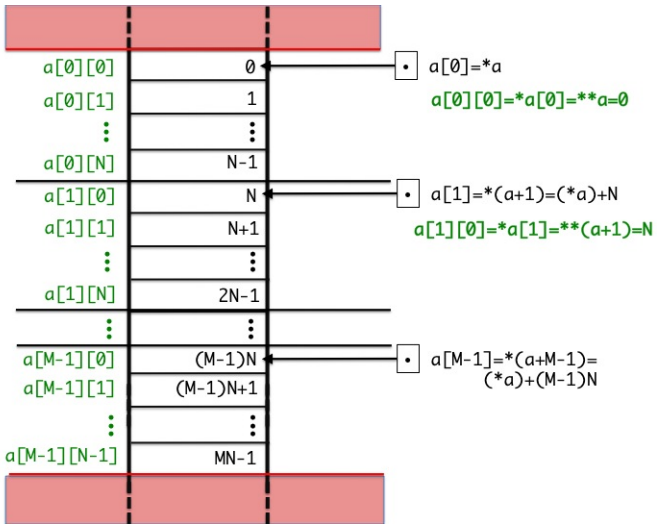

Figura 1: Effetto della dichiarazione di una matrice.

ciascuna riga) per riferire correttamente un elemento della matrice, tale valore va dichiarato quando una matrice viene passata come parametro. Questo, oggettivamente, è un fastidio che sarebbe stato meglio evitare, ma se i progettisti del C non sono riusciti a risparmiarcelo, evidentemente si tratta di un male necessario.

La Fig. 2 mostra, viceversa, come siano numerati gli elementi della matrice, da un punto di vista *logico*. La matrice in figura è una matrice 7×6: gli indici riga vanno da 0 a 6 e gli indici colonna da 0 a 5, coerentemente con quanto avviene nei vettori. Osservate che gli elementi di una colonna sono caratterizzati dal fatto di avere *lo stesso indice colonna*, e che gli elementi di una riga sono caratterizzati dal fatto di avere *lo stesso indice riga*, come ci si poteva aspettare.

Per cominciare a vedere come usualmente si percorre una matrice, vediamo la stampa di una matrice di interi, in modo che appaia con una struttura analoga a quella vista in Fig. 2. Per ottenere tale effetto, è necessario stampare un carattere di accapo alla fine della stampa di ciascuna riga. Per allineare le righe, vengono riservati a ciascun intero 4 caratteri. Questo permetterà di avere stampe ordinate, quando gli interi della matrice hanno un numero di cifre diverse (gli interi vengono allineati a destra). Ovviamente, riservando 4 spazi, la stampa sarà gradevole solo se i valori sono compresi tra -999 e 999.

```
void stampaMatrice(int a[][N], int m, int n){
   for (int i=0; i<M; i++){
       for (int j=0; j<N; j++)
            printf("%4d",a[i][j]);
       printf("\n");
   } /* end for */
}
```

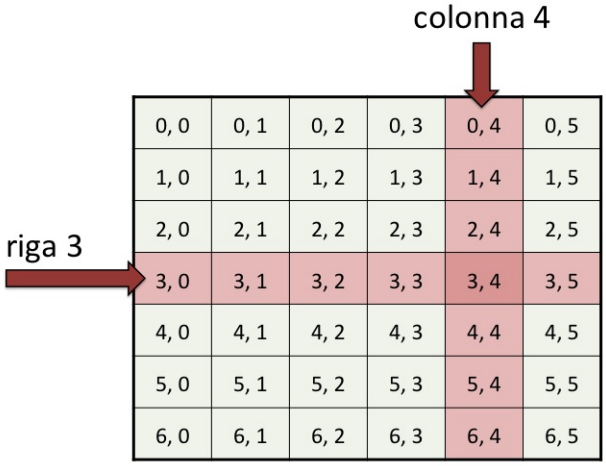

Figura 2: Matrici: righe e colonne.

Dopo questa breve introduzione, vedremo qualche problema più interessante. In Sezione 2.1, la verifica se una matrice rappresenta un quadrato magico ci permetterà di familiarizzare su come percorrere righe, colonne e diagonali. Dopo aver discusso un altro modo per definire le matrici, in Sezione 3.2, la costruzione di un quadrato magico di ordine dispari, ci permetterà di vedere come sia possibile percorrere una matrice in sequenze più bizzarre.

2.1 Verifica di un Quadrato Magico

Una matrice quadrata di lato n è un *quadrato magico* se contiene *tutti e soli* i numeri da 1 a n^2 e la somma di tutte le righe, tutte le colonne e delle due diagonali principali è costante. Siccome sappiamo dalla formula di Gauss che la somma dei primi n numeri è data da $\frac{n(n+1)}{2}$, ed essendoci n righe ed n colonne, la somma di tutti i numeri dentro la matrice sarà $\frac{n^2(n^2+1)}{2}$ e di conseguenza la *costante magica* deve essere necessariamente $\frac{n^2(n^2+1)}{2n} = \frac{n(n^2+1)}{2}$. Ad esempio, per $n = 3$, la costante magica sarà 15, per $n = 4$ sarà 34, e per $n = 5$ sarà 65. In Fig. 2.1 sono scritti 3 bellissimi esempi di quadrati magici[1]. Occorrerà verificare che la matrice contenga tutti gli elementi da 1 a n^2 una sola volta e sarà poi necessario scorrere ciascuna riga, colonna, e le due diagonali principali e verificare che la somma degli elementi sia sempre uguale alla costante magica (usando dei vettori, è possibile calcolare tutte le somme di colonne, righe e diagonali in una sola scansione della matrice).

[1]il secondo appare in una stampa di Duhrer. Era tipico nel Rinascimento, disseminare amenità matematiche nelle opere d'arte, e ci sono diversi quadrati magici in molti quadri dell'epoca.

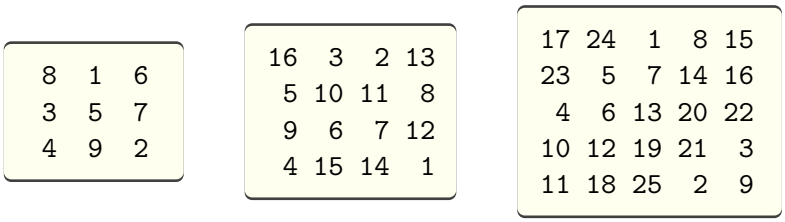

Figura 3: Quadrati magici di ordine 3, 4 e 5.

Per risolvere il primo problema, sarà conveniente usare un vettore `x` di n^2 interi che verranno usati come booleani. Il vettore viene inizialmente azzerato. Ogni volta che si incontra un certo valore k nella matrice, verrà verificato il valore di $x[k]$. Se $x[k]$ è diverso da 0, significa che k è già presente nella matrice e quindi non possiamo essere di fronte a un quadrato magico, altrimenti $x[k]$ viene posto ad 1 per marcare il fatto di aver incontrato il valore k. Osservate che `x` è un vettore definito e allocato dentro la funzione con un numero *variabile* (cioè dipendente dai parametri della funzione) di elementi. Questi vettori non appartengono al C standard, ma sono ormai usabili praticamente con ogni compilatore.

```
int verificaQuadratoMagico(int q[][N], int n){
   int x[n*n];
   int km=(n (n*n +1)) / 2;

   for (int i=0; i<n*n; i++) x[i]=0;

   /* verifica che q contenga tutti e soli i numeri in [1,n^2] */
   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++){
         if (a[i][j]<1 || a[i][j]>n*n) return 0;
         if (x[a[i][j]-1]) return 0;
           else x[a[i][j]-1]=1;
      }
   /* verifica che somme righe/colonne diano la costante magica */
   for (int i=0; i<n; i++){
      if (sommaR(q, n, i)!=km) return 0;
      if (sommaC(q, n, i)!=km) return 0;
   }
   /* verifica somma diagonali principali */
   if (sommaDA(q, n, n, n-1)!=km) return 0;
   if (sommaDD(q, n, n, 0)!=km) return 0;

   /* se abbiamo superato tutti i controlli ...*/
   return 1;
}
```

Figura 4: Verifica se una matrice quadrata sia un quadrato magico.

Per mantenere la complessità del codice ragionevole è conveniente scrivere il codice supponendo che il nostro esecutore sappia eseguire operazioni come sommare tutti gli elementi di una riga o di una colonna. A questo proposito può essere interessante una nota storica: uno slogan dei programmatori SmallTalk (il primo linguaggio Object–Oriented) era *"write stupid methods!"* che tradotto nel nostro contesto suonerebbe "scrivi funzioni stupide": se il loro codice supera un certo numero di righe, forse parte dei loro obiettivi va ottenuto invocando altre funzioni.

Tra l'altro, queste operazioni, potrebbero essere *riusate da altri programmi* (per questo nella parametrizzazione togliamo l'ipotesi che la matrice sia quadrata). Il risultato è rappresentato in Figura 2.1. Il codice delle funzioni `sommaR` e `sommaC` è riportato qui sotto e non dovrebbe richiedere ulteriori spiegazioni.

```
int sommaR(int a[][N],int n,int r){
    int s=0;
    for (int i=0; i<n; i++)
        s+=a[r][i];
    return s;
}
```

```
int sommaC(int a[][N],int m,int r){
    int s=0;
    for (int i=0; i<m; i++)
        s+=a[i][c];
    return s;
}
```

Le diagonali meritano un piccolo studio. Guardando la matrice da sinistra a destra, ci sono due tipi di diagonale. Le diagonali "ascendenti" (vedi Fig. 5) sono caratterizzate dal fatto che *la somma degli indici* degli elementi che stanno su una stessa diagonale *è costante*. Queste diagonali possono essere numerate con un numero che va da 0 a $m + n - 2$ che rappresenta tale somma di indici. La diagonale principale ascendente è la diagonale numerata $m - 1$.

Le diagonali "discendenti" (vedi Fig. 6) sono caratterizzate dal fatto che *la differenza degli indici* degli elementi che stanno su una stessa diagonale *è costante*. Queste diagonali possono essere numerate con un numero che va da $n - 1$ a $-m + 1$ che rappresenta tale differenza di indici. La diagonale principale dicendente è la diagonale numerata 0.

Scriviamo due procedure che calcolano rispettivamente la somma di una diagonale ascendente e discendente, ricevendo come parametri di input le dimensioni della matrice e il numero della diagonale come sopra descritto. L'idea è quella di posizionarsi sul punto d'inizio della diagonale e poi percorrere la diagonale (nella direzione in figura) finché non si esce dalla matrice. Le diagonali discendenti vengono percorse incrementando entrambi gli indici ad ogni passo, mentre quelle ascendenti incrementando l'indice colonna e decrementando l'indice riga.

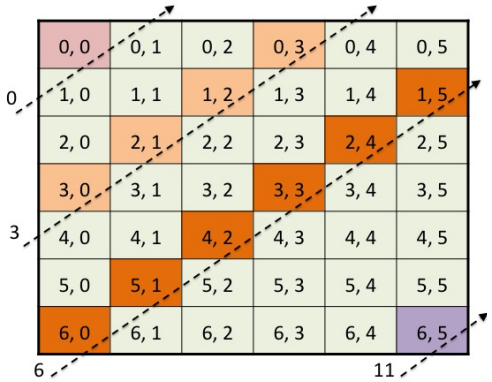

Figura 5: Diagonali "ascendenti".

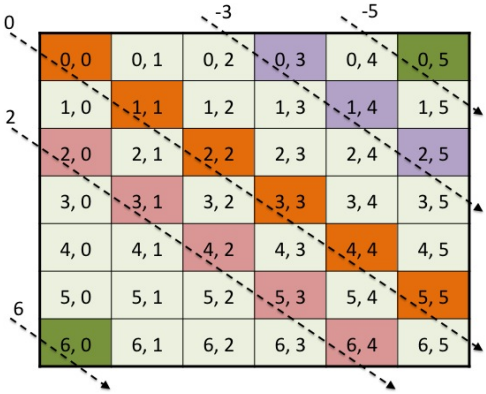

Figura 6: Diagonali "discendenti".

```
int sommaDA(int a[][N], int m,
            int n, int d){
    int i, j;
    if (d<m) {i=d; j=0;}
        else {i=m-1; j=d-m+1;}
    int s=0;
    while (i>=0 && j<n)
        s+=a[i--][j++];
    return s;
}
```

```
int sommaDD(int a[][N], int m,
            int n, int d){
    int i, j;
    if (d<0) {i=0; j=-d;}
        else {i=d; j=0;}
    int s=0;
    while (i<m && j<n)
         s+=a[i++][j++];
    return s;
}
```

2.2 Esercizi e Spunti di Riflessione

1. ♣ Onde evitare i noiosi problemi relativi al passaggio di una matrice come parametro, rappresentare una matrice di interi come un vettore di interi (di opportuna lunghezza). Scrivere due funzioni:

- `void set(int v[], int m, int n, int i, int j, int x)` che aggiorna la cella in posizione $[i, j]$ con il valore x in una matrice $m \times n$ codificata dal vettore v.
- `ing get(int v[], int m, int n, int i, int j)` che restituisce il contenuto della cella in posizione $[i, j]$ in una matrice $m \times n$ codificata dal vettore v.

Riscrivere alcuni programmi sulle matrici usando questa rappresentazione. Discutere pregi e difetti.

2. ♣ Come nell'esercizio precedente, usare un vettore per rappresentare matrici di forma particolare (per esempio matrici triangolari come il triangolo di Tartaglia), senza sprecare memoria. Per ogni particolare matrice dovrete opportunamente programmare le funzioni `set` e `get`.

3 Vettori e Matrici Dinamici

Le soluzioni agli esercizi in Sezione 2.2 permetterebbero di liberarsi della fastidiosa necessità di mettere la lunghezza delle righe nel tipo dei parametri matrice passati a una funzione. L'esercizio 2.2(2) sul Triangolo di Tartaglia, inoltre, è motivato dall'obiettivo di non sprecare memoria quando si alloca una matrice che, per esempio, è triangolare. Un ultimo aspetto da considerare è il seguente: immaginate una funzione che deve *creare* una matrice (il verbo ha un sapore più religioso che informatico, ma è di uso comune) e restituirla in qualche modo al chiamante.

I compilatori C moderni permettono di scrivere codice come quello nel riquadro, in cui una matrice (o un vettore) viene creata con un numero *variabile* di elementi, cioè dipendente dal valore di una espressione da valutare durante l'esecuzione.

```
int** creaMatrice(int r, int c){
   int a[r][c];
   return a;
}
```

Tuttavia ha da ritenersi scorretto ritornare il puntatore come risultato al chiamante. Infatti, la matrice viene allocata nel record di attivazione della funzione `creaMatrice` e viene pertanto *deallocata* contestualmente al rientro dalla funzione.

Ci sono quindi tre buoni motivi per considerare i cosìdetti *vettori dinamici*, allocati esplicitamente dal programmatore: 1. usare sempre e comunque il tipo `T**` per le matrici, senza preoccuparsi del loro numero di colonne; 2. allocare matrici con righe di lunghezza variabile (ad esempio matrici triangolari); 3. scrivere funzioni che creano vettori o matrici e restituiscono come risultato un puntatore ad essi.

Prima cosa da sapere: è possibile allocare memoria *fuori* dal record di attivazione di una funzione (e in generale, fuori dalla pila di sistema) in un'area chiamata *heap*. Il C mette a disposizione delle funzioni di libreria (le più famose `malloc` e `calloc`, per il cui uso è necessario includere le librerie standard con `#include <stdlib.h>`) che allocano una certa quantità di memoria richiesta dal programmatore. Per allocare un vettore di n elementi interi, si può operare come segue (mostriamo sia l'uso di `malloc` e di `calloc`).

```
int* creaVettore(int n){
   int *a = (int *) malloc(sizeof(int)*n);
   int *b = (int *) calloc(n,sizeof(int));
   return a;
}
```

A questo punto `a` è un puntatore che punta ad un blocco di memoria consecutiva sufficiente a memorizzare n interi. La pseudo funzione `sizeof(T)` restituisce la

quantità di memoria necessaria a memorizzare un elemento di tipo `T`[2]. Tale memoria può comodamente essere usata come fosse un vettore, riferendo gli elementi con l'usuale notazione `a[i]`. Una piccola differenza tra `calloc` e `malloc` (oltre al fatto che la prima prende due parametri e la seconda uno solo) è che `calloc` automaticamente azzera gli n elementi allocati (dovrebbe essere quindi leggermente più lenta di `malloc`, ma con le attuali architetture, raramente questa "inefficienza" è veramente rilevante).

Per le matrici, la questione è leggermente più complicata. Il fatto è che dopo aver eseguito la funzione nel riquadro sotto, `a` è semplicemente un vettore con $r \times c$ elementi e non potremo usare la usuale comoda notazione per le matrici in cui ogni elemento è individuato da due indici. Se volessimo farlo, dovremo ricorrere alle tecniche dell'Esercizio 1 in Sezione 2.2.

```
int* creaMatrice(int r, int c){
   int *a = (int *) calloc(r*c,sizeof(int));
   return a;
}
```

Esiste tuttavia una brillante soluzione: pensare una matrice come un *vettore di vettori*. Ogni elemento di un vettore sarà semplicemente un puntatore a un vettore riga. Tutto miracolosamente torna: se ho una variabile `a` di tipo `int**`, allora `a[i]` sarà di tipo `int*`, quindi (se correttamente allocato) sarà un vettore e di conseguenza `a[i][j]` sarà il j-esimo elemento del vettore riga $a[i]$.

Per definire e allocare una matrice (rettangolare o anche "irregolare") $r \times c$, dovremo quindi: 1. definire una variabile `a` di tipo `T**` (Fig. 7); 2. allocare un vettore di r elementi di tipo `T*` (Fig. 8); 3. per ciascun elemento $a[i]$ del vettore allocare un vettore riga (Fig. 9).

Come la figura suggerisce non è nemmeno necessario che i vettori riga abbiano la stessa lunghezza (voi, però la dovete conoscere, per evitare sconfinamenti in zone oscure della memoria il cui accesso è proibito al vostro programma). Facciamo due piccoli esempi per familiarizzare con questo potente strumento.

3.1 Costruzione del Triangolo di Tartaglia

Cominciamo con un programma che alloca e riempie correttamente il noto Triangolo di Tartaglia (o di Newton, o di Pascal... *paese che vai, triangolo che trovi!* ☺). La funzione è mostrata in Fig. 10.

Osservate come prima cosa, il tipo della funzione: dovendo restituire al chiamante una matrice di interi, il valore di ritorno avrà tipo `int**`. Appena entrati nella

[2] osserviamo che è deprecabile allocare memoria speculando di sapere quanta memoria occupi, ad esempio, un intero: questo è un valore che può dipendere dall'architettura della macchina e può cambiare nel tempo.

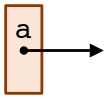

Figura 7: Dopo la dichiarazione di `a`.

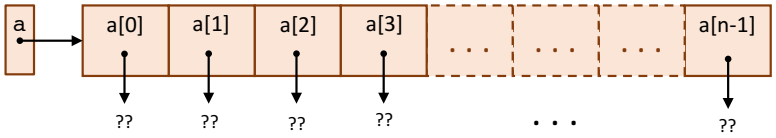

Figura 8: Dopo l'*allocazione* `a`.

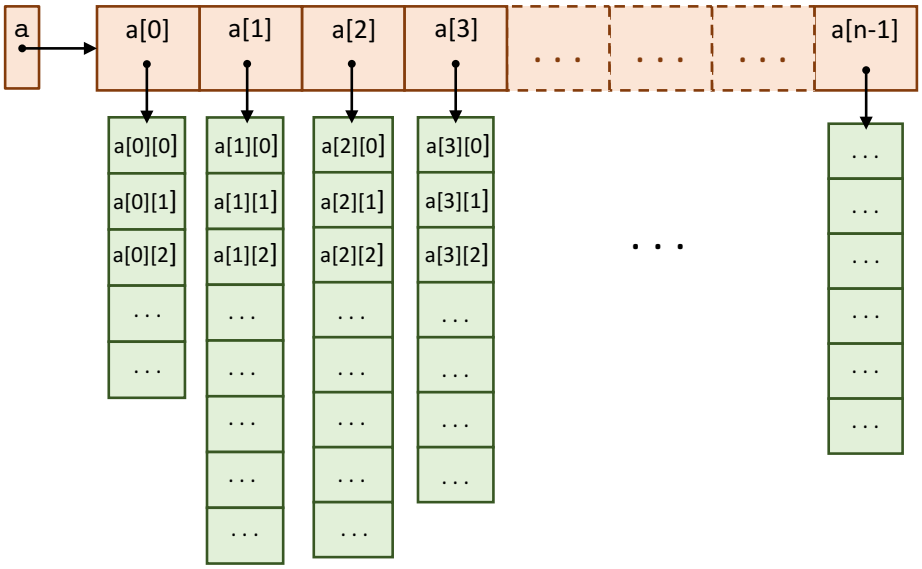

Figura 9: Dopo l'allocazione di ciascun vettore `a[i]`.

funzione, si crea il vettore di righe, che saranno tante quante richieste dal chiamante. A quel punto, si farà un ciclo che alloca ogni riga una per una. Già che ci siamo, ne approfittiamo anche per caricare i valori nella riga appena caricata. Si posizionano gli 1 nel primo e ultimo elemento della riga (questi elementi non possono essere generati seguendo la regola base di costruzione del triangolo di Tartaglia, a meno di non immaginare la matrice "incorniciata" dentro due strisce di 0, che comunque ci farebbero solo sprecare inutilmente memoria). Osservate che la riga i–esima ha lunghezza $i + 1$.

Osservate anche che, quando si mettono gli 1 in riga 0, avrete, per $i = 0$ che `t[i][0]` e `t[i][i]` sono la *stessa* cella di memoria, per la precisione quella riferibile con `t[0][0]`. In questo caso, tutto fila liscio perché entrambe le assegnazioni met-

tono un 1 in quella casella. Lo spreco di un'assegnazione è pienamente compensato dal fatto di essere sollevati dal trattamento particolare del caso 0.

A questo punto, siamo pronti per caricare la parte *interna* di una riga, e lo facciamo come usuale applicando la regola di costruzione del triangolo, che qui viene applicata nell'assegnazione `t[i][j]=t[i-1][j-1]+t[i-1][j];`. Un'altra piccola finezza è che l'espressione `t[i-1][j-1]+t[i-1][j]` è chiaramente mal definita quando i è 0. Fortunatamente[3] quando i è 0 non si entra nel ciclo `for` interno.

```
int** triangoloTartaglia(int n){

   int** t = calloc(n, sizeof(int *));

   for (int i=0; i<n; i++){
       t[i] = calloc(i+1, sizeof(int));
       t[i][0]=1; t[i][i]=1;
       for (int j=1; j<i; j++)
            t[i][j]=t[i-1][j-1]+t[i-1][j];
     }
   return t;
 }
```

Figura 10: Costruzione del Triangolo di Tartaglia.

Concludiamo l'analisi di questo piccolo problema, mostrando una funzione che stampa una matrice triangolare. Niente di particolarmente interessante, se non il fatto che in questo caso possiamo speculare sul fatto che sappiamo che la riga i–esima ha lunghezza $i + 1$. Dovessimo veramente allocare e maneggiare una matrice *irregolare* con numero variabile di elementi in ciascuna riga, forse sarebbe opportuno creare un *vettore di lunghezze* contestualmente alla matrice e passarlo tra i parametri ogni qual volta fosse necessario.

```
void printMT(int** m, int r){
    for (int i=0; i<r; i++){
       for (int j=0; j<i+1; j++)
            printf("%5d", m[i][j]);
       printf("\n");
    }
 }
```

3.2 Costruzione di un Quadrato Magico di Ordine Dispari

Un quadrato magico $n \times n$ con n dispari può essere costruito con un semplice algoritmo, che sintetizziamo nel seguito:

[3] ma il bravo programmatore è *sempre* fortunato!

Si scrive 1 nella casella centrale della prima riga. Dopo aver scritto un certo numero m nella casella $[i, j]$, si scrive $m + 1$ andando in diagonale verso l'alto e verso destra, cioè in casella $[i-1, j+1]$. Se tale casella casca fuori dalla matrice, bisogna immaginare che la prima riga o colonna (cioè quelle numerate 0) siano adiacenti all'n-esima riga o colonna (cioè quelle numerate $n-1$). Infine, se la casella $[i-1, j+1]$ è già occupata, allora $m+1$ va posto nella casella immediatamente sotto a quella occupata da m, cioè quella di coordinate $[i-1, j]$.

I quadrati magici di ordine 3 e 5 all'inizio di Sezione 2.1 sono stati ottenuti seguendo questo algoritmo. Non è un programma particolarmente difficile. Sarà sufficiente fare tutti i controlli nell'ordine in cui sono stati descritti: si calcolano gli indici della nuova posizione, se questa è fuori dai limiti della matrice se ne calcola un'altra (seguendo le regole date), si verifica se la casella è vuota (possiamo codificare questo fatto con il numero 0): in caso affermativo mettere il numero corrente, altrimenti si ricalcola un'altra volta la prossima posizione. Da osservare che, siccome la nuova posizione può dover essere ricalcolata più volte, conviene usare delle variabili temporanee `i` e `j` che rappresentano la nuova casella in considerazione (senza riaggiornare subito gli indici dell'ultima casella riempita `x` e `y`, perché da questi valori dipendono ulteriori prossimi tentativi). Il codice è mostrato in Fig. 3.2. La funzione `allocM` è in Fig. 3.2 e alloca e azzera (invocando `calloc`) una matrice.

```
int **qmD(int n){
  int** q = allocM(n,n);
  int i, j;
   /* mi posiziono al centro della prima riga */
  int x = 0;
  int y = n/2;
  int k = 1;
  do {
     q[x][y] = k++;
     i = x-1;
     j = y+1;
     if (i<0) i=n-1;
     if (j==n) j=0;
     if (q[i][j]==0){ x=i; y=j;}
        else x++;
  } while (k<=n*n);
  return q;
}
```

Figura 11: Costruzione di un Quadrato Magico di ordine dispari.

```
int **allocM(int r, int c){
    int** m = calloc(r, sizeof(int *));
    for (int i=0; i<r; i++)
         m[i] = calloc(c, sizeof(int));

    return m;
}
```

Figura 12: Allocazione e azzeramento di una matrice.

3.3 Esercizi e Spunti di Riflessione

1. (Sudoku, Homework 3, 2013) Scrivere un programma che *verifica* se una matrice di interi di lato $n^2 \times n^2$ è la soluzione di un Sudoku (questo esercizio generalizza il tradizionale Sudoku 9×9).

2. (La Sorgente d'Acqua, Homework 2, 2015) Una matrice di interi di dimensioni $m \times n$ rappresenta le quote di un rilievo topografico. In una certa coordinata è presente una sorgente d'acqua. Sapendo che l'acqua può scendere in tutte le direzioni (verticale, orizzontale, diagonale) verso punti con quota minore o uguale, ma ovviamente non può salire, determinare tutte le zone che saranno allagate. Scrivere una versione iterativa e una ricorsiva.

   Ad esempio, data la matrice a sinistra, e $[0, 2]$ come posizione della sorgente, la matrice a destra rappresenta con `2` la posizione della sorgente, con `1` le zone raggiunte dall'acqua, e con `0` quelle che non saranno allagate.

   ```
   8 1 7 4 6          0 1 2 1 0
   1 2 9 8 3          1 1 0 0 1
   7 6 1 9 2          0 0 1 0 1
   ```

3. ♣★ Una matrice si dice *parzialmente ordinata* se i vettori riga e i vettori colonna sono tutti ordinati. Formalmente se per ogni r, c, $i \leq k$ implica che $a[i][c] \leq a[k][c]$ e $a[r][i] \leq a[r][k]$. Generalizzare l'idea di ricerca binaria su una matrice parzialmente ordinata. Calcolare la complessità del vostro algoritmo con opportune relazioni di ricorrenza. Ricordate inoltre che la taglia dell'input è $m \times n$, dove m ed n al solito sono le dimensioni della matrice.

   Scrivere una versione iterativa e una ricorsiva.

4. (Matrice a Spirale, Homework 2, 2015) Scrivere un programma che legge un numero intero n e crea una matrice a *spirale* $n \times n$.
   Si mette in alto a sinistra (in posizione [0,0]) il numero 1 e poi, andando in senso *orario* verso il centro della matrice, si dispongono gli altri numeri 2,3,4, ... fino a n^2.

   Ecco ad esempio le matrici a spirale di lato 3 e 4.

```
1   2   3          1   2   3   4
8   9   4         12  13  14   5
7   6   5         11  16  15   6
                  10   9   8   7
```

5. Una matrice quadrata `r` di dimensione $n \times n$ contenente $0, 1$ può rappresentare una *relazione binaria* R su un insieme finito $I = \{a_0, \ldots a_{n-1}\}$ di n elementi nel seguente modo: $r[i, j] = 1$ se e solo se $R(a_i, a_j)$ vale. Scrivere delle funzioni che ricevuta come parametro di ingresso la matrice `r`:

   - determinano se R è riflessiva, transitiva, simmetrica, antisimmetrica etc. (chi più ne ha, più ne metta).
   - definiamo la metrica $d : I \times I \mapsto \mathbb{N}$ come segue:

$$d(a_i, a_j) = \begin{cases} 0 & \text{se } i = j \\ 1 & \text{se } R(a_i, a_j) \\ n+1 & \text{se } \exists k.R(a_i, a_k) \wedge d(a_k, a_j) = n \\ \infty & \text{altrimenti} \end{cases}$$

     Data in input una matrice `r`, scrivere una funzione C che costruisce la matrice `d` delle minime distanze (`d` ha le stesse dimensioni di `r`). Porre a -1 la casella $d[i, j]$ se $d(a_i, a_j) = \infty$.
   - Sotto la precondizione che R sia un ordine parziale: trovare la catena più lunga; gli elementi massimali; dire se R ammette massimo o minimo.
   - Sotto la precondizione che R sia una relazione di equivalenza, determinare il numero m delle classi di equivalenza; ordinando le classi di equivalenza rispetto all'indice minimo degli elementi che vi appartengono, caricare un vettore `cEq` di m elementi, tale che $cEq[i] = k$ se l'elemento $a_i \in I$ appartiene alla k-esima classe di equivalenza.

6. Data una matrice a quadrata $n \times n$, definiamo *centro della matrice* la casella di coordinate $[\lfloor n/2 \rfloor, \lfloor n/2 \rfloor]$ se n è dispari, e il quadrato 2×2 formato dagli elementi di indice $\{[n/2 - 1, n/2 - 1], [n/2 - 1, n/2], [n/2, n/2 - 1], [n/2, n/2]\}$ se n è pari. La *cornice* C_0 è l'insieme degli elementi che stanno sul *bordo* della matrice, cioè $\{a[i, j] \mid i = 0 \vee i = n - 1 \vee j = 0 \vee j = n - 1\}$ La cornice C_{k+1} è l'insieme degli elementi adiacenti alla cornice C_k "verso" il centro, cioè $\{a[i, j] \mid \exists a[x, y] \in C_k.i = x + 1 \leq n/2 \vee i = x - 1 \geq n/2 \vee j = y + 1 \leq n/2 \vee j = y - 1 \geq n/2\}$. Il centro è la cornice $C_{n/2}$.

   • Dare una definizione induttiva di cornice partendo dal centro della matrice;

   • trovare una espressione semplice, non induttiva, dipendente da i e j per determinare a quale cornice appartenga un elemento $a[i, j]$;

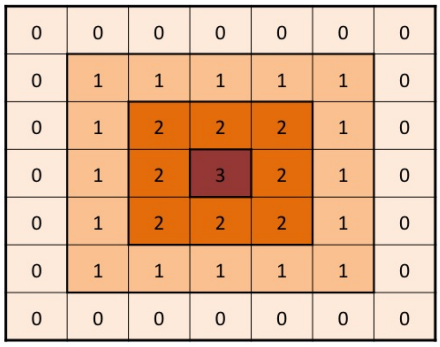

Figura 13: Cornici in una matrice di lato dispari

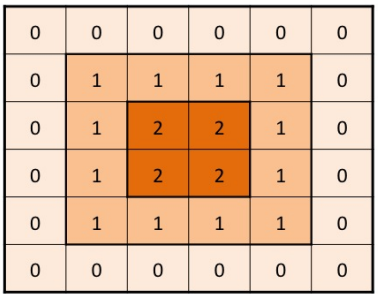

Figura 14: Cornici in una matrice di lato pari.

- scrivere una funzione che determina la cornice di somma massima;
- scrivere un funzione che determina se gli elementi di una cornice sono ordinati in modo crescente andando in senso orario e restituire l'indice dell'elemento minimo;
- diciamo che una matrice è una *piramide* se gli elementi di una cornice più esterna sono tutti minori degli elementi di una cornice più interna. Scrivere una funzione che determina se una matrice è una piramide.

7. Scrivere una funzione che presa in input una matrice $m \times n$, determina la sottomatrice quadrata di somma massima. Scrivere una versione ricorsiva e una iterativa.

8. Scrivere una funzione che presa in input una matrice $m \times n$, determina la sottomatrice quadrata più grande di somma positiva.

   Scrivere una versione ricorsiva e una iterativa.

9. Come visto in un esercizio precedente, una matrice di 0 e 1 può rappresentare una relazione binaria su un insieme finito. Data una matrice c che rappresenta la relazione di conoscenza C tra gli invitati I a una festa, definiamo l'insieme $V \subseteq I$ un *insieme di vip* se è verificata la seguente condizione: $\forall x, y \in V.C(x, y) \wedge \forall x \in I \setminus V, y \in V.C(x, y) \wedge \neg C(y, x)$ (cioè i vip si conoscono tutti tra loro, tutti i non vip conoscono i vip, mentre i vip non conoscono nessuno che non sia un vip).

   Sotto la precondizione che esista un insieme V di vip non vuoto, scrivere una funzione di *complessità lineare* che lo determina.