Informatica Generale
Programmazione C

Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 7, 7 aprile 2020

Las Vegas

La vignetta di Biani

Precisazioni su vettori e puntatori

```c
#include <stdio.h>

int main(){
    char a[5]="amor";
    char *b;
    char  c[];

    b=a;
    c=b;

    for(int i=0; i<5; i++) printf("%lc\n", *a++);

    for(int i=0; i<5; i++) printf("%lc\n", *b++);

    return 0;
}
```

Non compila: scrittura ammessa solo nei parametri di una funzione

Scorre correttamente il vettore a

Non compila: non si può alterare la base di un vettore

Lezione 7a:

Soluzione Esonero 2016

Combinazioni n oggetti presi k a k

Correzione Primo Esonero, Esercizio 2

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2015-16

Esercizio 2: *Considerare le combinazioni di n oggetti presi k a k. Supponiamo di rappresentare una di tali combinazioni con un vettore di k interi compresi tra* 1 *ed n memorizzati nel vettore in ordine* crescente.

Punto 1 *Scrivere una funzione C* `int nextCbn(int c[], int n, int k)` *che presa in input una combinazione c di k interi tra* 1 *ed n, modifica c in modo che rappresenti la prossima combinazione nell'ordine lessicografico e torna 1. Se la combinazione c in ingresso è l'ultima,* `nextCbn` *torna 0.*

Questo problema è un adattamento di un famoso problema (e soluzione): il problema della **prossima permutazione** (vedi tra i "piccoli classici", la dispensa **C1**.

Analisi del problema

Ovviamente, occorre prima capire il problema. Scriviamo, ad esempio, le 10 combinazioni di numeri da 1 a 5, presi 3 a 3, nell'ordine lessicografico: [1,2,3], [1,2,4], [1,2,5], [1,3,4], [1,3,5], [1,4,5], [2,3,4], [2,3,5], [2,4,5], [3,4,5]. Osservate che, finchè possibile, è sufficientemente incrementare l'ultimo elemento, fino alla combinazione $[1, 2, 5]$. A quel punto, occorre incrementare il 2, ma la prossima combinazione semplicemente contiene, da quel punto in poi, numeri consecutivi in ordine crescente, ed è infatti $[1, 3, 4]$. Facciamo un ultimo esempio. Arrivati a $[1, 4, 5]$ non è possibile nè incrementare il 5, nè il 4, perchè il 5 è già presente nella combinazione. A questo punto è possibile incrementare solo l'1, continuando, riempendo ancora a destra con numeri consecutivi, ottenendo $[2, 3, 4]$. Ovviamente, giunti a $3, 4, 5$, non c'è nessun numero che si possa incrementare, e quindi non ci sono combinazioni successive e la nostra funzione tornerà 0.

Cambia il secondo elemento

Cambia il primo elemento

Soluzione: dove incrementare?

L'algoritmo, che ovviamente dipende dal fatto che rappresentiamo le combinazioni con i numeri in ordine crescente, consiste nel trovare, *partendo da destra*, il primo indice in cui è possibile incrementare. Si incrementa quel numero di 1 e poi si posizionano i suoi successivi, uno alla volta alla sua destra. Per fare questo lavoro, è conveniente definire una funzione `void firstCmb(int* c, int l, int q)` che carica nel vettore c, l elementi consecutivi a partire dal valore q come in Fig. 1.

```
void firstCmb(int* v, int l, int q) {
     int i;
     for(i=0; i<l; i++) v[i]=q++;
}
```

Figura 1: Funzione che sistema la parte destra della combinazione

Come riconoscere il punto dove incrementare? Chiaramente per l'ultimo elemento in posizione $k-1$, deve essere $c[k-1] < n$. Per un elemento interno in posizione $i < k-1$, deve essere $c[i] < c[i+1]-1$. Quindi non appena si è trovato il punto giusto i, si sistema la coda della combinazione, invocando `firstCmb(&c[i],k-i,c[i]+1)`, che incrementa $c[i]$ di 1 e riempie la coda con numeri consecutivi nelle $k-i$ posizioni rimanenti (senza tale funzione è comunque sufficiente un facile ciclo `for`).

Soluzione

Alternativamente, è sufficiente osservare che nella zona "non incrementabile", $c[i] = n - k + i + 1$ (idea non mia, ma emersa in un compito e che rispetto alla mia soluzione, consente di non trattare a parte l'ultimo elemento).

Se non si trova nessun punto in cui sia possibile incrementare, si torna 0 senza fare nulla. Il risultato in Fig. 2.

```
int nextCmb(int c[], int n, int k){
  /* PREC: c vettore di k elementi,
   * contiene numeri distinti tra 1 ed n,
   * c ordinato crescente
   */

   for (int j=k-1; j>=0; j--)
       if (c[j]<n-k+j+1){
          firstCmb(&c[j],k-j,c[j]+1);
          return 1;
       }
   return 0;
   }
```

Osservare la parametrizzazione di ***firstCmb***

Figura 2: Funzione `nextCmb`

Stampare tutte le combinazioni

Punto 2: *Usare la funzione* `nextCmb` *del punto precedente per scrivere una funzione iterativa C* `allCbn(int n, int k)` *che stampa in ordine crescente* tutte *le combinazioni di interi da* 1 *a n presi k a k [*OSSERVAZIONE*: non occorre memorizzare le combinazioni!].*

Questo esercizio è un calcio di rigore al novantesimo! E fa portare a casa i 3 punti. Avendo la funzione `nextCmb`, è sufficiente caricare un vettore c la prima combinazione (osservare che qui posso riutilizzare `firstCmb`), e poi iterare `nextCmb` su c finchè tale funzione non torna 0. Ovviamente, ad ogni ciclo si stampa c. Il risultato in Fig. 3, dove viene usata una funzione `printCmb` per ottenere una stampa opportuna di una combinazione.

```
void allCmb(int n, int k){
   int i;
   int v[k];

   firstCmb(v,k,1);
   printV(v,k);
   while (nextCmb(v,n,k)) printCmb(v,k);
}
```

Figura 3: Funzione `allCmb`

Generazione ricorsiva

Punto 3: *Scrivere una funzione C ricorsiva* `allCbnRec(int n, int k)` *che usa lo schema ricorsivo del programma che calcola i coefficienti binomiali per stampare tutte le combinazioni di interi da* 1 *a* n *presi* k *a* k*.[omissis lungo suggerimento]*

Osserviamo il programma **ricorsivo** per i coefficienti binomiali (per una procedura più efficiente, vedere la dispensa **D5** sui vettori, che genera le righe del Triangolo di Tartaglia):

```
int cBin(int n, int k){
    if (n==k || k==0) return 1;
    return cBin(n-1, k) + cBin(n-1, k-1);
}
```

Osservate come si può **interpretare**: le combinazioni di n oggetti presi k a k sono quelle **che non contengono il primo** (e quindi devo disporre n-1 oggetti su k posti) e quelle **che contengono il primo** (e quindi devo poi considerare tutti i modi di disporre gli n-1 altri oggetti nei k-1 posti rimanenti)

Generazione ricorsiva

Ogni discesa ai casi base del programma precedente, corrisponde a una combinazione ben precisa. Occorre mantenere **informazione** su quale sia la **combinazione in costruzione**.

Molto astrattamente (ma in modo codificabile in alcuni linguaggi di programmazione) stiamo definendo la seguente equazione ricorsiva (immaginiamo di generare i sottoinsiemi di cardinalità k di un insieme di n elementi):

$$\mathcal{P}_k(X \cup \{a\}) = \{Y \cup \{a\} \mid Y \in \mathcal{P}_{k-1}(X)\} \cup \mathcal{P}_k(X)$$

Riassumendo, dovrò scrivere una funzione ausiliaria `allCbnRecAux(int n, int k,int c[], int q, int j, int l)` in cui c è la combinazione in costruzione, q è il primo numero da sistemare nel sottoalbero ricorsivo corrente, j è il punto del vettore in cui sono arrivato, l è la lunghezza originaria delle combinazioni in costruzione. (in realtà j è sempre $l - k$ e potrebbe essere evitato, magari passando il pointer al primo punto "libero" del vettore c).

Generazione ricorsiva

Il risultato in Fig. 4. Osservare che se $k = 0$, la funzione `firstCmb` non fa nulla (infatti ho 0 elementi da sistemare). Osservare anche che la seconda chiamata sovrascriverà la casella j di c, per cui non occorre preoccuparsi di annullare gli effetti delle assegnazioni (tipo `c[j]=q;`).

```
void allCmbAux(int n, int k, int c[], int q, int j, int l){

  if (k==0 || n==k){
     firstCmb(&c[j],l-j,q);
     printCmb(c,l);
     return;
   }
   c[j]=q;
   allCmbAux(n-1,k-1,c,q+1,j+1,l);
   allCmbAux(n-1,k,c,q+1,j,l);
}

void allCmbRec(int n, int k){
  int c[k];
  allCmbAux(n,k,c,1,0,k);
}
```

Figura 4: Funzione `allCmbRec`

Lezione 7b:

Matrici

I vettori pluridimensionali

Molti problemi hanno una natura bidimensionale: pensate a memorizzare lo stato di un qualsiasi gioco da tavolo (esempi: dama, scacchi, filetto, forza4…) oppure tabelle, o ancora sistemi di equazioni…

Tutti i linguaggi imperativi (anche il più primitivo, il FORTRAN) offrono la possibilità di usare **vettori** (o **array**) che sono un'**astrazione della memoria della macchina**: la memoria di un calcolatore, infatti, è un unico enorme array in cui ciascun elemento **è riferibile mediante il suo indirizzo** (o puntatore).

Da un punto di vista astratto, è come avere **un numero di variabili che può cambiare in diverse esecuzioni** del programma, e il cui **nome** possa essere `**calcolato**' a tempo di esecuzione.

Definizione di una matrice

Classicamente, in C, una matrice **statica** si definisce con:

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

dove `T` è il **tipo** degli elementi del vettore, `M` è il numero di righe ed `N` è il numero di colonne. `M` e `N` sono **costanti**, eventualmente simboliche definite con una `#define`. Al solito, le righe sono numerate da 0 a `M`-1 e le colonne da 0 a `N`-1.

Gli elementi vengono allocati nella memoria **monodimensionale** del calcolatore.

Memorizzazione di una matrice

Gli elementi di un'array, sono memorizzati in celle contigue di memoria. Ecco l'effetto della dichiarazione `int a[M][N]`, e poi caricata col codice sotto.

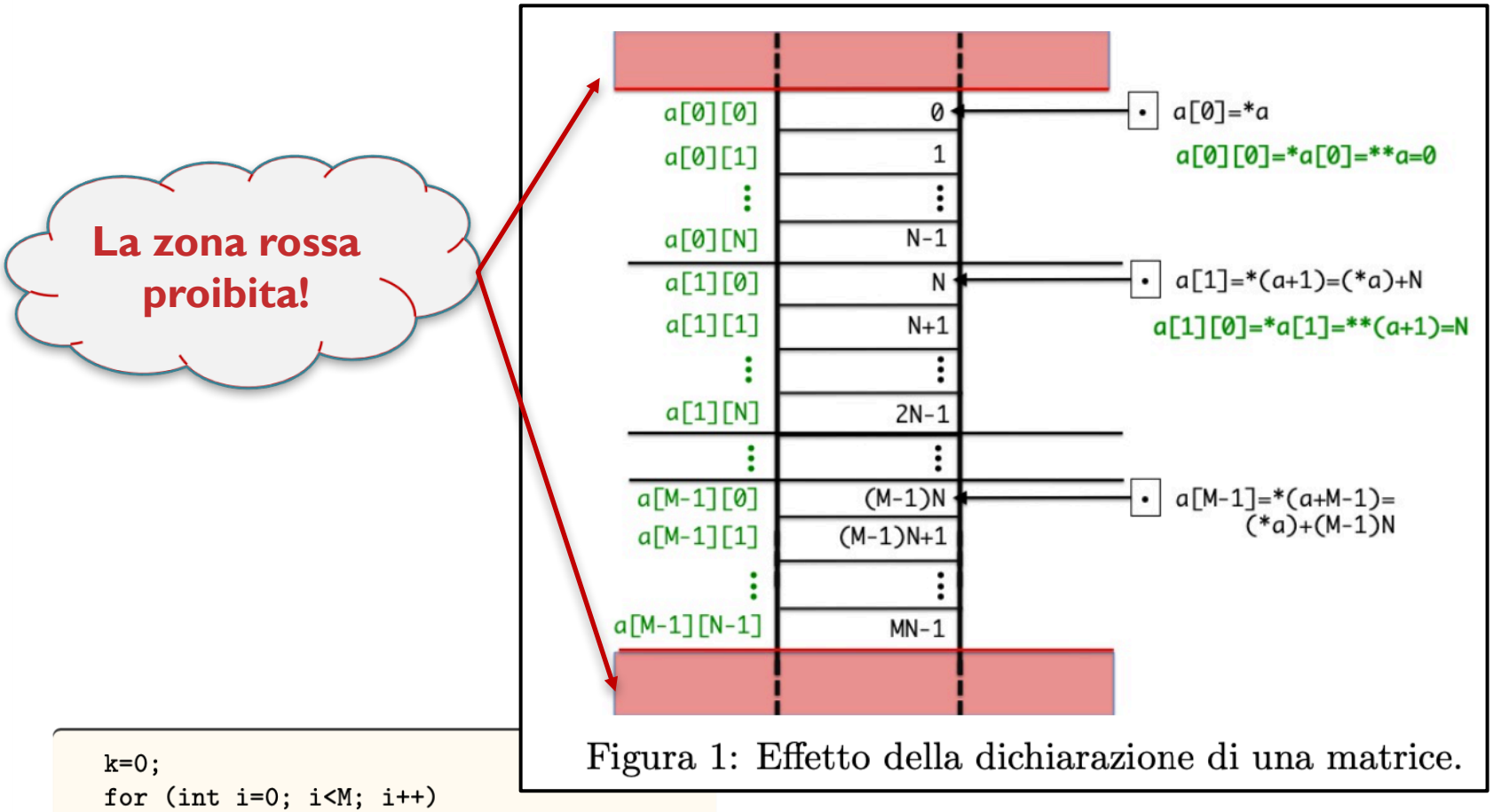


Figura 1: Effetto della dichiarazione di una matrice.

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

Attenzione: matrici e puntatori

Il tipo della matrice è `T[][]` che è equivalente a `T**`. Se osservate, è come allocare `M` vettori di lunghezza `N`. **Tuttavia…**

Che significa `a[k]`? Significa saltare `k` elementi di tipo `T[]`: quindi significa `a+k*N*sizeof(T)`! In generale, l'indirizzo di `a[i][j]` sarà calcolato come:

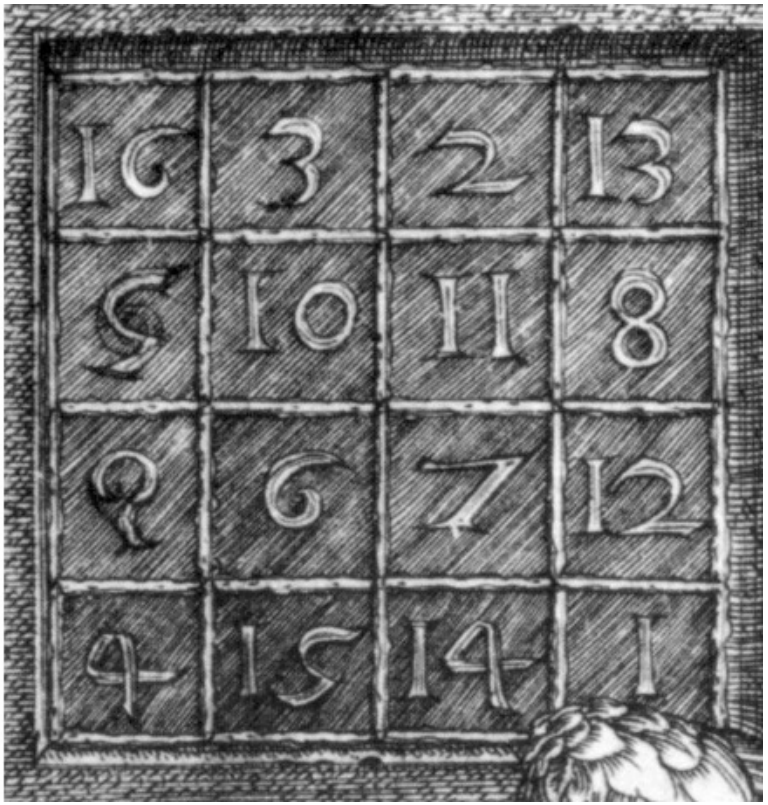
`a+(i*N+j)*sizeof(T)`

Osservate che questa operazione **necessita di conoscere la lunghezza delle righe** `N`, cioè il numero di colonne.

Questo spiega perché dovendo passare una matrice come parametro, occorre indicare il numero di colonne: `T[][N] a`:

```
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 */
}
```

Verifica di un quadrato magico



Una matrice di interi $n \times n$ è un **quadrato magico** se contiene tutti e soli i numeri da 1 a n^2 e la somma su tutte le righe, tutte le colonne e le due diagonali principali è uguale (nella figura, **34**).

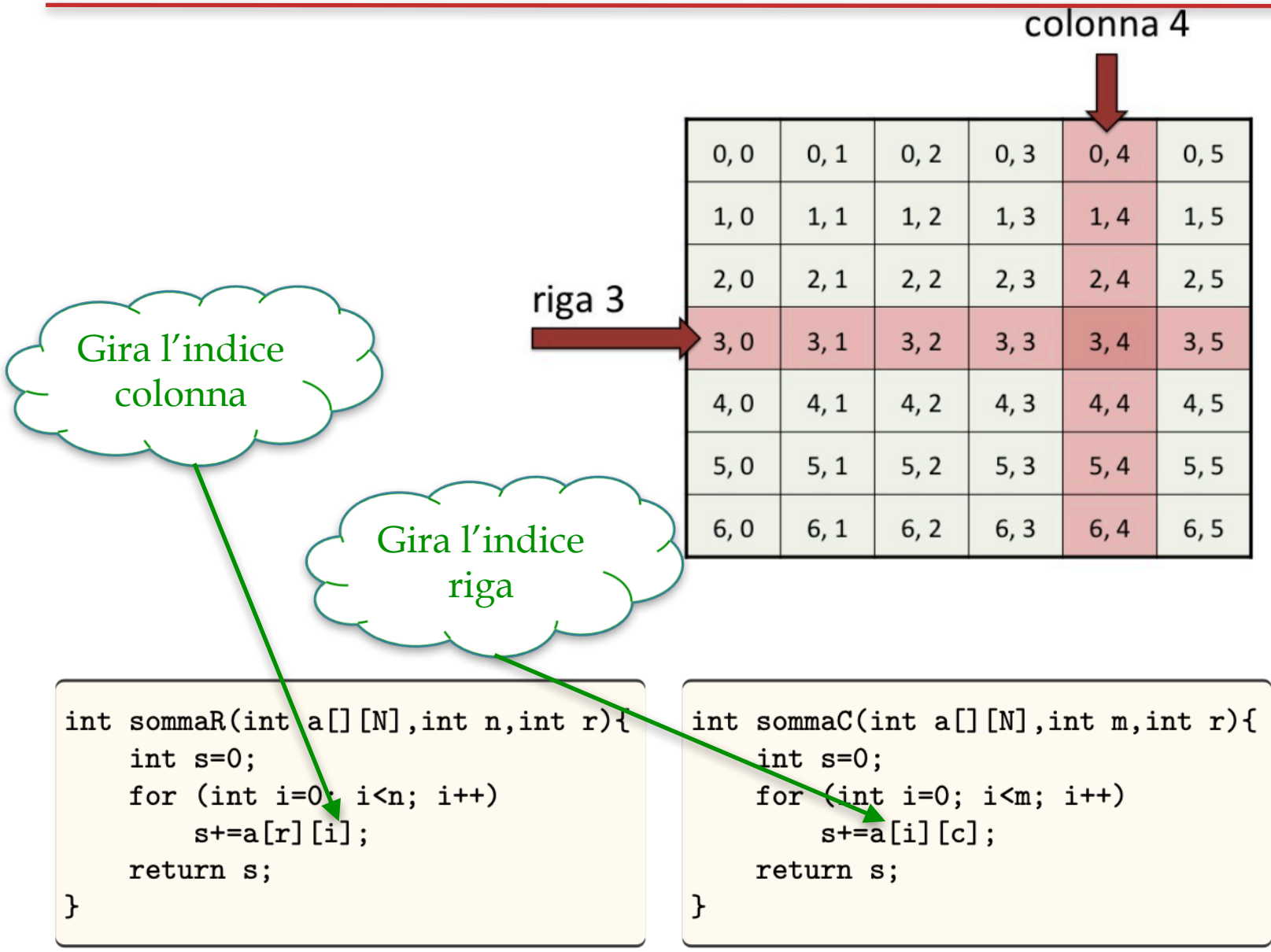
Costante magica

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.

Scopo del problema è prendere dimestichezza con righe, colonne, e diagonali. Dobbiamo infatti controllare:

1. che ci siano (1 sola volta) tutti i numeri da 1 a n^2
2. La somma delle righe e colonne sia uguale alla costante magica;
3. Le due diagonali principali.

Righe e colonne




```
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;
}
```

Diagonali 'ascendenti' e 'discendenti'

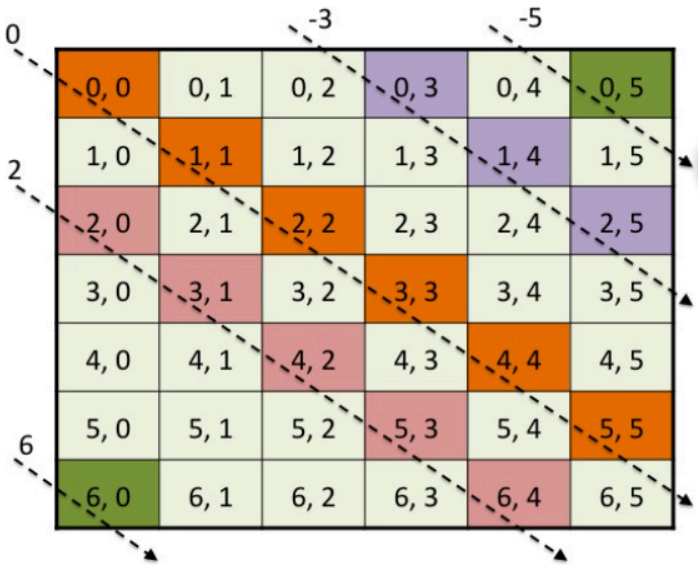


Figura 6: Diagonali "discendenti".

Ogni diagonale **discendente** ha la **differenza** degli indici costante e possono essere numerate da m-1 a $-n$+1

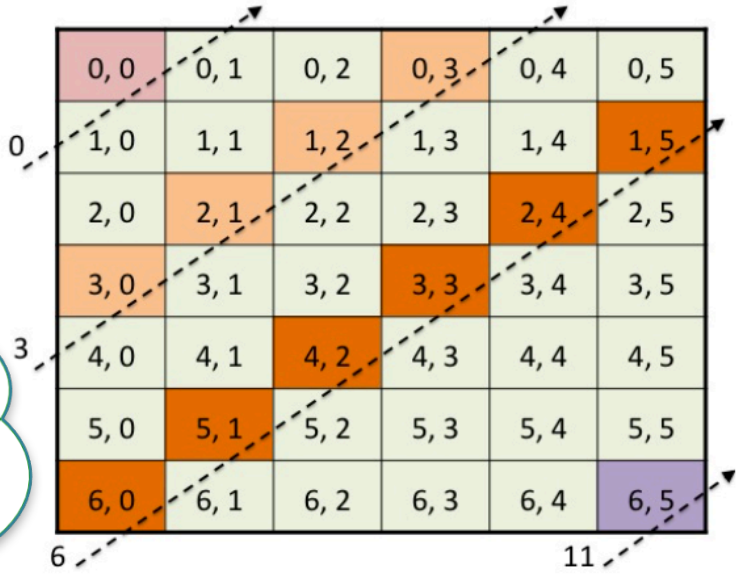


Figura 5: Diagonali "ascendenti".

Ogni diagonale **ascendente** ha la **somma** degli indici costante e possono essere numerate da *0* a m+n-2

Diagonali 'ascendenti' e '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;
}
```

Occorre capire dove "comincia" una diagonale

Attenzione a non uscire dalla matrice!

Quadrato magico: programma

```
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;
}
```

costante magica

Verifica dei numeri presenti: simile al minimo intero libero con vettore

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

Utili esercizi…

… per liberarsi del fardello di passare il numero di colonne tra i parametri e per evitare di sprecare memoria per matrici non rettangolari… ma si può fare meglio…

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`.

Lezione 7c:

Matrici Dinamiche

Allocazione di una matrice dinamica

Una soluzione un po' naif sarebbe la seguente:

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

che non permetterebbe poi di usare correttamente gli indici.

La soluzione corretta consiste nell'allocare un **vettore di puntatori a vettori**, ciascuno quindi punta a un **vettore riga** della matrice.

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).

Allocazione di una matrice dinamica

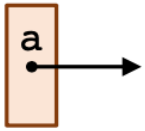


Figura 7: Dopo la dichiarazione di a.

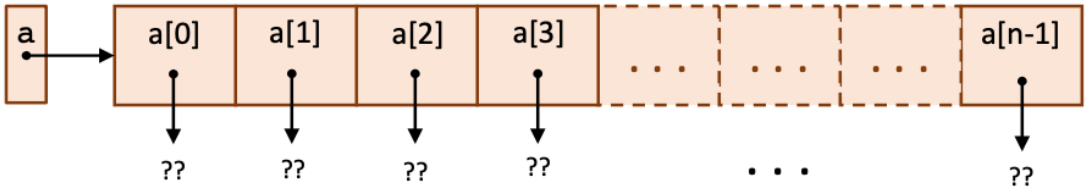


Figura 8: Dopo l'*allocazione* a.

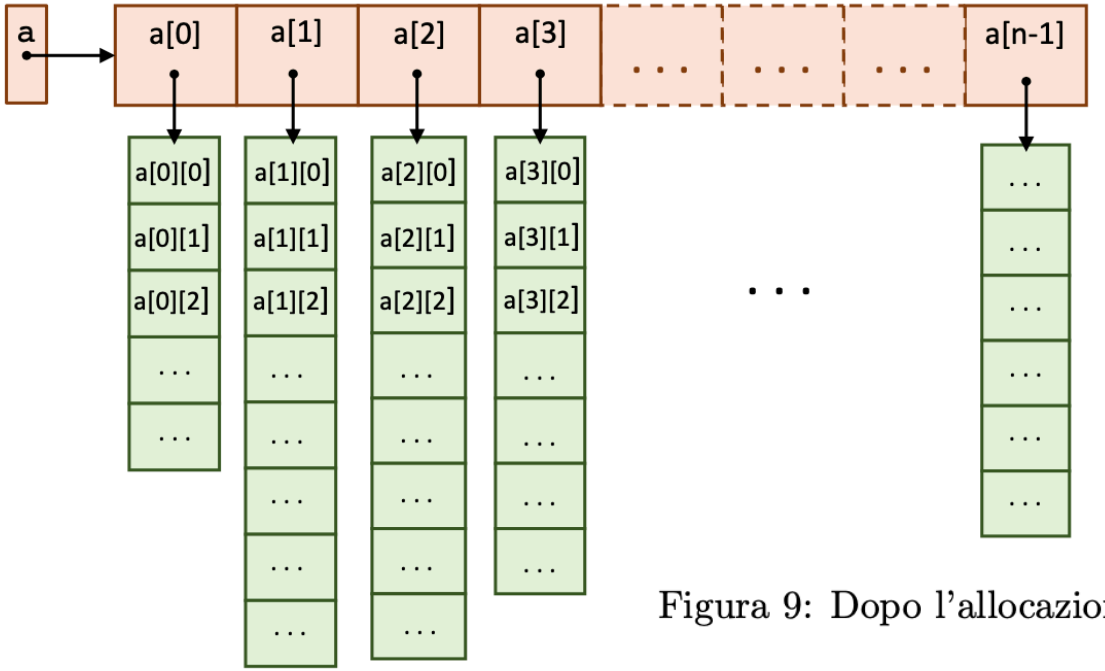
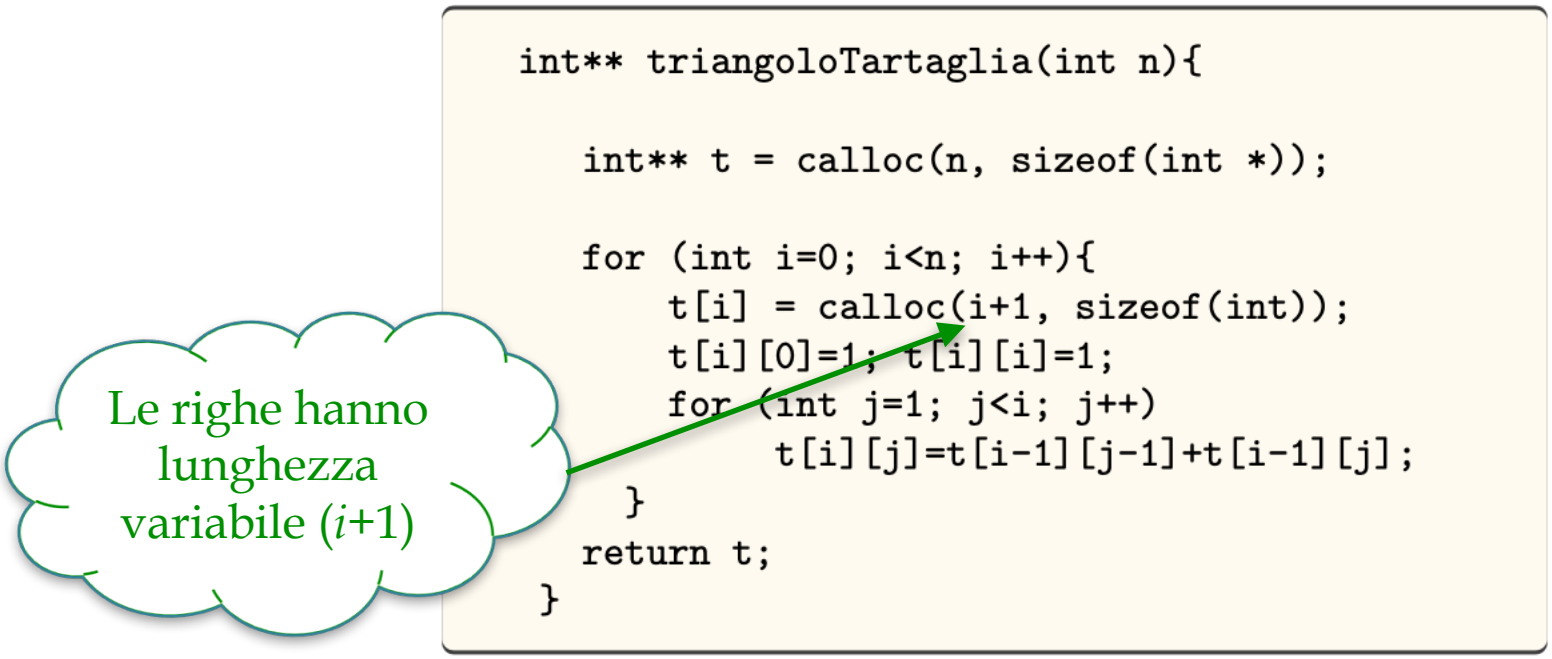


Figura 9: Dopo l'allocazione di ciascun vettore `a[i]`.

Esempio: Triangolo di Tartaglia




```
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.

Nota: Per `i == 0`, `t[i-1][j]` sarebbe mal definita, ma in quel caso non si entra nel for. In quel caso `t[i][0]` è lo stesso di `t[i][i]`: facciamo un'assegnazione "di troppo", ma **evitiamo di considerare un ulteriore caso particolare**.

Costruzione di un QM di lato dispari

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 $[i-1, j]$.

8	1	6
3	5	7
4	9	2

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Costruzione di un QM di lato 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.

```
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.

Lezione 7

That's all Folks…

…Domande?