

Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

*Vettori e matrici dinamici in C
Allocazione dinamica di Memoria*

Corso di Laurea in **Informatica**, III anno



SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 20, 13 maggio 2021

Lezione 20a:

Vettori variabili

Il problema del minimo intero libero

Esercizio 2: Considerare il problema di trovare il minimo intero non presente in un vettore di interi non negativi distinti. Ad esempio, nel vettore:

$\{3, 8, 4, 5, 11, 15, 9, 2, 6, 0, 1\}$

il minimo intero non presente è il 7.

Punto 1 Scrivere una funzione `C` di prototipo `int minFree(int v[], int n)` che restituisca il minimo intero non presente in `v`. La funzione `minFree` non deve modificare il vettore `v` e non può allocare strutture dati di dimensione dipendente da `n`. Valutare, anche informalmente, la complessità della funzione.

Non potendo allocare memoria, né modificare memoria, le uniche soluzioni possibili sembrano essere l'**iterazione** di una procedura di **ricerca** o di **minimo**:

- cercare 0, 1, 2, ... finché la ricerca non fallisce;
- calcolare i minimi m successivi (cioè maggiori del minimo precedente p_m) del vettore e arrestarsi quando $m > p_m + 1$.

Entrambe queste procedure hanno “chiaramente” costo $\mathcal{O}(n^2)$.

Per il **principio dei buchi di piccionaia**, il risultato è in $[0, n]$!

Soluzione con ricerca iterata

```
int trova(int a[], int n, int x, int *p){
    for (int i=0; i<n; i++)
        if (a[i]==x) {*p = i; return 1;}
    return 0;
}

int minFree(int a[], int n){
    int p, i=-1;
    while (trova(a, n, ++i, &p);
    return i;
}
```

È sempre una buona idea definire **funzioni ausiliarie** (che risolvono sottoproblemi) e farle il più generali possibile.

In questo caso, `trova` ha un prototipo molto generale e restituisce anche l'indice in cui si trova l'elemento `x`. Noi non usiamo questa informazione.

Osservate che tutto viene fatto nella valutazione della guardia del `while`.

Altre soluzioni divertenti

```
int minFreeNonAnnidata(int a[], int n){
    int i=0;
    int j=0;
    while (i<n && j<n)
        if (a[i]==j){i=0; j++;}
        else i++;
    return j;
}
```

Questa funzione fa grossomodo le stesse cose della precedente ma con un unico **while**.

Osservate che una volta che j è stato trovato si cerca il prossimo j e si rimette i a 0.

Quando un certo j non si trova, si esce perché i diventa n .

Probabilmente, ogni funzione con cicli annidati si può scrivere con un unico ciclo (non annidato, **Esercizio**)

Allocando memoria...

Punto 2: Osservando che il minimo intero non presente in v deve necessariamente essere un numero nell'intervallo $[0, n]$, dove n è la lunghezza del vettore, fornire una funzione `int minFreeLin(int v[], int n)` di complessità lineare in

n che risolve lo stesso problema avendo la libertà di allocare un vettore di opportuna lunghezza locale alla funzione `minFreeLin`.

Per risolvere questo esercizio dobbiamo introdurre un nuovo concetto: i **vettori variabili**. È possibile, dentro una funzione dichiarare un vettore con **un numero di elementi che dipende** ad esempio dal **parametro** di una funzione.

Non è C standard, ma ormai accettato da tutti i principali compilatori C (in particolare anche dal gcc da diversi anni).

Attenzione: tale vettore viene **allocato nel record di attivazione della funzione** e quindi de-allocato quando la funzione smette di eseguire!

Soluzione lineare con vettore

Possiamo allocare un vettore p di lunghezza n e codificare

$$\text{con: } \begin{cases} p[i] = 1 & \text{se } i \text{ occorre in } v \\ p[i] = 0 & \text{se } i \text{ non occorre in } v \end{cases}$$

Prima di cominciare non sappiamo nulla, e quindi il vettore p va azzerato. Poi scorriamo v e mettiamo $p[v[i]]$ a 1 per ogni i .

Infine, scorriamo p e il primo indice i tale che $p[i]=0$ è il risultato cercato.

```
int minFreeLin(int a[], int n){
    int p[n+1];
    azzer(a, p, n+1);

    for (int i=0; i<n; i++)
        if (a[i]<=n) p[a[i]]=1;

    for (int i=0; i<n+1; i++)
        if (!p[i]) return i;
}
```

il minimo intero libero sta in $[0, n]$

*occorre fare attenzione **a non uscire dai limiti** di p !*

si torna l'indice del primo 0

Alcune osservazioni

Limitazioni sui
vettori variabili

1. la dichiarazione `int p[n]={0}`; non viene accettata dai compilatori con vettori di lunghezza *variabile*. Non ho tolto punti per questo “errore”. Comunque, poco male, possiamo invocare una funzione `azzera` (meglio di azzerare il vettore in loco con un ciclo `for` per motivi di semplicità del codice – anche qui questioni che non modificano il punteggio ottenuto).

2. Più importante, ricordarsi che prima di settare p_{v_i} ad 1 è necessario assicurarsi che $v_i \leq n$. Attenzione sempre alla *zona proibita*!

Errore Interessante: Al solito, alcuni errori sono particolarmente interessanti. Qualcuno, noncurante dei saggi consigli della traccia, ha voluto calcolare il valore $m = \max_i v_i$ e poi definire il vettore p con m elementi. Questo evita il controllo $v_i \leq n$, ma cosa accade se il vettore v , contenesse ad esempio i valori $\{0, 1, 2, 3, 2^{64} - 1\}$?

Sempre meglio
scomporre in funzioni

Sappiamo che il
risultato è in $[0, n]$

Potendo modificare il vettore?

Punto 3: *Non potendo allocare memoria come al punto 1, ma potendo modificare il vettore, in quale altro modo è possibile accelerare la funzione scritta al punto 1? [non è richiesto codice per questo punto]*

Qui occorre semplicemente dire che una volta riordinato il vettore (investendo $n \log n$ passi), il minimo intero non presente è il primo indice i per cui $v_i > i$, oppure n se per ogni $i \in [0, n). v_i = i$. Chiaramente $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$.

A rigore, *mergeSort* non va tuttavia bene (qualcuno lo ha osservato!) perchè non è un algoritmo *in place* e necessita di allocare memoria (un vettore ausiliario) durante la fase di fusione (onore a chi ha fatto questa osservazione, ma niente punizioni per chi ha detto di usare *mergeSort*).

I raffinati hanno poi cercato l'intero mancante con una ricerca binaria. Finezza che non cambia la complessità $n \log n$. Ma come si fa ad adattare la ricerca binaria per trovare il minimo numero mancante? Un delicato esercizio di antipasto, assolutamente da fare, per preparare il cervello alla soluzione del Punto 4.

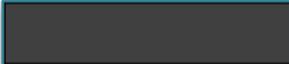
Soluzione Divide et Impera

Punto 4: *Adattando opportunamente l'algoritmo quickSort (in particolare la funzione partiziona), scrivere una funzione C basata su divide et impera ricorsiva di complessità lineare che non alloca memoria, ma modifica v, senza necessariamente riordinarlo tutto. SUGG: scrivere una funzione ausiliaria ricorsiva di prototipo:*

```
int minFreeAux(int v[], int inf, int sup)
```

che viene chiamata rispettando la preconditione che il minimo numero libero sia nell'intervallo $[inf, sup]$.

 Supponete ora di conoscere un indice s del vettore per cui tutti gli elementi a sinistra di s siano minori di tutti gli elementi a destra di s . Chiamando $S = \{v_i \mid inf \leq i < s\}$ e $D = \{v_i \mid s \leq i < sup\}$ avremo che per ogni $x \in S$ e $y \in D$, necessariamente $x < y$. Se avete suddiviso il vettore seguendo la tecnica di partizionamento di *quickSort*, conoscete anche il valore p del perno che soddisfa alla condizione $x \leq p < y$, ancora con $x \in S$ e $y \in D$.

Inoltre, ragionando sulle cardinalità di S e D sapete se m sta in S o in D : infatti se $|S| < p - inf$, allora $m \in [inf, inf + s]$, altrimenti $m \in [inf + s, sup]$. 

Soluzione Divide et Impera

Scelta del Perno:

Resta da decidere come scegliere il perno: non occorre che sia un elemento del vettore e la scelta ideale è il punto medio $(inf + sup)/2$. Così facendo, se $|D| < (inf + sup)/2$ andrò a cercare a destra, altrimenti a sinistra, ma in quel caso allora avremo $|S| < (inf + sup)/2$. Morale: in entrambi i casi ho almeno dimezzato la dimensione della porzione da analizzare (curiosamente, in questo caso, il caso

Casi Base:

- in una chiamata con $inf + 1 = sup$, ho solo v_{inf} nella porzione di vettore da analizzare e quindi deve essere che se $v_{inf} = inf$, allora $m = inf + 1$, mentre se $v_{inf} > inf$ allora $m = inf$;
- in una chiamata con $inf = sup$ necessariamente $m = inf$.

Soluzione Divide et Impera

```
void swapV(int v[], int i, int j){
    int h = v[i];
    v[i] = v[j];
    v[j] = h;
}

int minFreeAux(int v[], int inf, int sup){
    /* PREC: f \in [inf,sup] */
    int m = (inf + sup) / 2;

    if (inf==sup) return inf;
    if (sup==inf+1)
        if (v[inf]==inf) return inf+1;
        else return inf;

    int i=inf;
    int s=sup-1;
    while (i<s){
        while (v[i]<=m) i++;
        while (v[s]>=m) s--;
        if (i<s) swapV(v,i,s);
    }
    if (i<m) return minFreeAux(v,inf,i+1);
    else return minFreeAux(v,i+1,sup);
}

int minFreeDeI(int v[], int n){
    return minFreeAux(v,0,n);
}
```

Confrontare con la versione Haskell che risolve lo stesso problema seguendo la stessa idea!

Figura 5: Funzione lineare basata su *Divide et Impera*

Soluzione Divide et Impera

Complessità:

la dimensione della porzione da analizzare (curiosamente, in questo caso, il caso pessimo è il dimezzamento bilanciato, perchè andrò sempre a scegliere la porzione più piccola per cercare il minimo intero libero). Quindi la complessità dell'algoritmo è maggiorata da $n + n/2 + n/4 + \dots = n + n(1/2 + 1/4 + \dots)$. La serie tra parentesi lasciava perplesso Zenone di Elea riflettendo sul fatto se Achille avrebbe potuto raggiungere la tartaruga, ma noi sappiamo che converge a 1 e quindi tutto a $2n$ (ovviamente si può impostare una relazione di ricorrenza).

Tutto ciò conduce alla funzione in Fig. 5, che è un curioso miscuglio tra *quickSort* e ricerca binaria.

Addendum Un arguto studente ha osservato (dopo il compito) come sia possibile scrivere una funzione lineare *senza* allocare memoria e *senza* modificare il vettore. Usando in sostanza la tecnica che risolve il punto **2**, riuscendo a codificare l'informazione nel vettore di appoggio p dentro v in modo "reversibile": essendo il vettore v di interi non negativi (assumiamo per semplicità che lo 0 non sia in v , se c'è sommiamo 1 a tutti gli elementi di v), posso porre v_i a $-v_i$ per codificare il fatto di sapere che i è un elemento di v . A quel punto il minimo intero non presente, sarà l'indice del primo elemento di v positivo! Prima di tornare, risistemiamo le cose, riportando i valori negativi al loro valore assoluto (sottraendo 1 per invertire l'operazione necessaria a sbarazzarsi dell'eventuale 0 presente).

Ma dov'è il trucco?

Lezione 6b:

Allocazione di Memoria

Generiamo un vettore...

Immaginiamo di voler risolvere il seguente problema:

“Scrivete una funzione C di prototipo

```
int* generaPrimi(int n, int* k)
```

*che genera il vettore di tutti i numeri primi minori di n, e carica nell'intero *k il numero dei primi trovati”*

Potremmo essere tentati di scrivere qualcosa del tipo:

```
int* generaPrimi(int n, int* k){  
    int p[n]; /* n è un po' troppo */  
    .. /* codice che carica p */  
    return p;  
}
```

Tuttavia sarebbe un **grave errore**. Il vettore `v` è **locale** a `generaPrimi` e viene **dealloca**to quando la funzione termina.

Allocazione di memoria

In C è possibile **allocare memoria dinamicamente**, durante l'esecuzione del programma. Tale memoria risiede in una zona di memoria della macchina **diversa dallo stack** di attivazione delle chiamate di funzione, detta **Heap**.

L'allocazione di memoria avviene attraverso chiamate a funzioni della **libreria <stdlib.h>**. Le più usate sono:

```
void* malloc(int n)
void* calloc(int k, int n)
```

Entrambe allocano un blocco di memoria di un certo numero di bytes (n nel caso di `malloc` e $k \times n$ nel caso di `calloc`) e **ritornano un puntatore** alla base di tale blocco.

Osservate che il tipo di ritorno è **void ***: si tratta del tipo puntatore che **è compatibile per assegnazione con tutti gli altri tipi puntatore**. Ciò è necessario affinché queste funzioni siano generali e possano tornare pointer di ogni tipo.

Uso di malloc e calloc

Se voglio allocare memoria per **un singolo intero** dovrò eseguire un'istruzione tipo:

```
int* m = (int *) malloc (sizeof(int))
```

Osserviamo un po' di cose:

(int *) si chiama **coercion**: malloc torna un void* ma io lo assegno a una variabile int *: in C classico questo è assolutamente normale (un puntatore è sempre un puntatore), tuttavia i compilatori moderni tendono ad avvertire il programmatore che sta facendo qualcosa di strano. Mettendo la coercion, “tranquillizziamo” il compilatore.

sizeof è una pseudofunzione che **torna il numero di byte necessario per memorizzare un dato di un certo tipo**. **Evitate sempre chiamate tipo: malloc(42)**, ma usate sempre **sizeof**: questo rende i programmi correttamente eseguibili su macchine diverse che hanno diverse dimensioni dei dati.

Allocazione di un vettore dinamico

Un vettore contenente n elementi di un certo tipo T , può essere allocato usando indifferentemente una delle seguenti istruzioni:

```
T* v = (T *) malloc (n*sizeof(T))
```

```
T* v = (T *) calloc (n, sizeof(T))
```

L'unica differenza è che **calloc** azzera la memoria allocata. Era quindi poco amata dai Veri Programmatori C, in quanto “più lenta”: al giorno d'oggi tutto ciò è (praticamente) ininfluenza sui tempi di calcolo.

A questo punto, possiamo usare v come un qualsiasi vettore, e possiamo anche usare l'operatore $[]$ per accedere ai singoli elementi.

Lezione 20b

Vettori Dinamici e Allocazione di Memoria

Esempio: Crivello di Eratostene

Scriviamo la funzione `int* generaPrimi(int n, int* k)` usando l'algoritmo del crivello di Eratostene.

Il risultato di tipo `int*` sarà **l'indirizzo base di un vettore** di numeri primi lungo `*k`.

Esiste un antichissimo metodo (forse uno dei primi algoritmi di cui si abbia conoscenza) per generare tutti i numeri primi da 1 ad n , noto come *Crivello di Eratostene*, che risale al III secolo avanti Cristo. Il metodo si può scrivere informalmente come segue:

“si scrivono tutti i numeri naturali da 1 a n . Si comincia da 2 e si cancellano tutti i suoi multipli 4, 6, 8, 10, ... fino a n . Si prende il primo numero non cancellato, il 3, e si cancellano tutti i suoi multipli 6, 9, 12, 15, ... Il prossimo numero non cancellato ora è il 5 e si cancellano i suoi multipli, e così via. Alla fine, seguendo questo procedimento, i numeri non cancellati rimasti sono tutti i numeri primi tra 1 e n .”

Esempio: Crivello di Eratostene

Useremo un vettore p con n elementi con l'idea che $p[i]=1$ se i è primo e 0 altrimenti. Dopodiché useremo p per costruire il vettore di primi.

Assumendo che il vettore p sia inizializzato con tutti 1, possiamo usare la seguente funzione per implementare l'algoritmo di Eratostene.

```
void crivello(int p[], int n){
  /* REQ: forall i\in[0,n). p[i]=1
   * ENS: forall i\in[0,n). p[i]↔i è primo
   */
  for (int i=2; i*i<n; i++)
    /* INV: forall k\in[0,n).
     * p[k]=1↔k non è divisibile per primi < i
     */
    if (p[i])
      for (int j=i*i; j<n; j+=2*i) p[j]=0;
}
```

*p[i] è primo e
canello tutti i suoi
multipli*

Generare i Primi

Osservate che uso un **vettore variabile** p (non servirà più finita la funzione) mentre **allochiamo dinamicamente** il vettore dei primi che **devo restituire come risultato**.

Devo contare i primi per allocare v e caricare il loro numero in $*k$

```
int* generaPrimi(int n, int *k){
    int p[n+1];
    /* inizializzo p a tutti 1 */
    inizializza(p, n+1, 1);
    crivello(p, n+1);
    /* conto gli 1 in p per alloc. il ris.*/
    *k = contaUni(p, n+1);
    int *v = (int*) calloc(*k, sizeof(int));
    caricaPrimi(v, p, n+1);
    return v;
}
```

```
void caricaPrimi(int v[], int p[], int n){
    int j=0;
    for (int i=0; i<n; i++)
        if (p[i]) v[j++]=i;
}
```

Osservazione importante

Il vettore p , viene deallocato quando finisce di eseguire `generaPrimi`.

Se avessimo allocato p usando `malloc` o `calloc` **sarebbe rimasta impegnata la memoria occupata da p** , senza peraltro poterlo usare nuovamente, in quanto verrebbe **perso il riferimento a p** (locale alla funzione `generaPrimi`).

Tale memoria sarebbe diventata **garbage** (cioè spazzatura).

Attenzione, quindi. Esiste la funzione **`free(void *)`** per liberare la memoria allocata con `malloc` o `calloc`.

Lezione 20c

Matrici

e

Matrici Dinamiche

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...

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

$$\text{T } a[M][N];$$

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.

La zona rossa proibita!

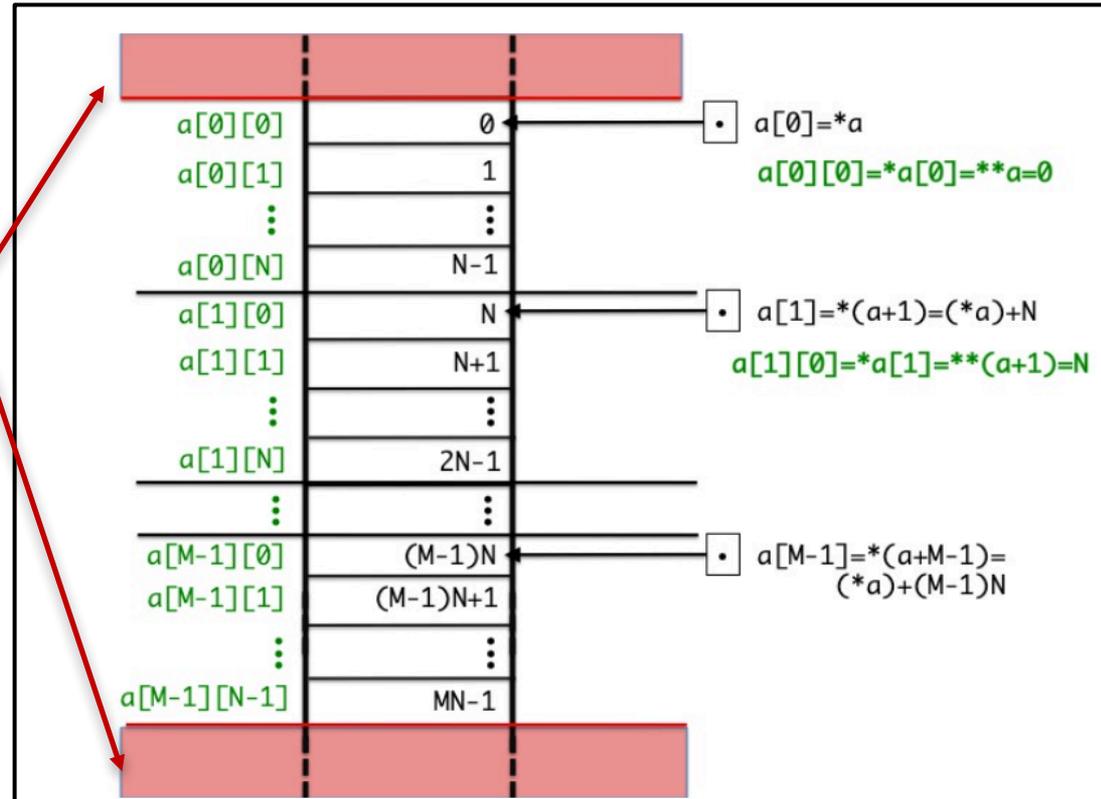


Figura 1: Effetto della dichiarazione di una matrice.

```
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 `N` delle righe**, 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 */
}
```

*int** non può essere accettato!*

Allocazione di una matrice dinamica

Una soluzione un po' naïf 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

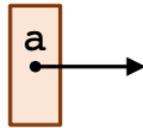


Figura 7: Dopo la dichiarazione di a.

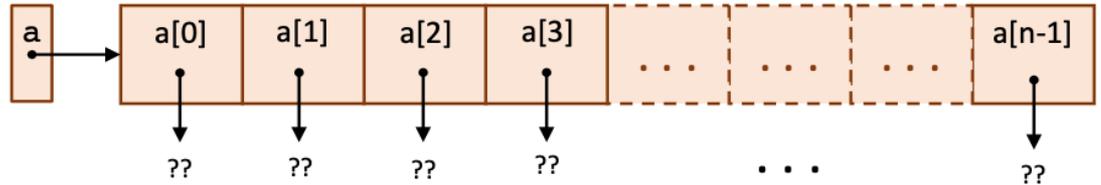


Figura 8: Dopo l'allocazione a.

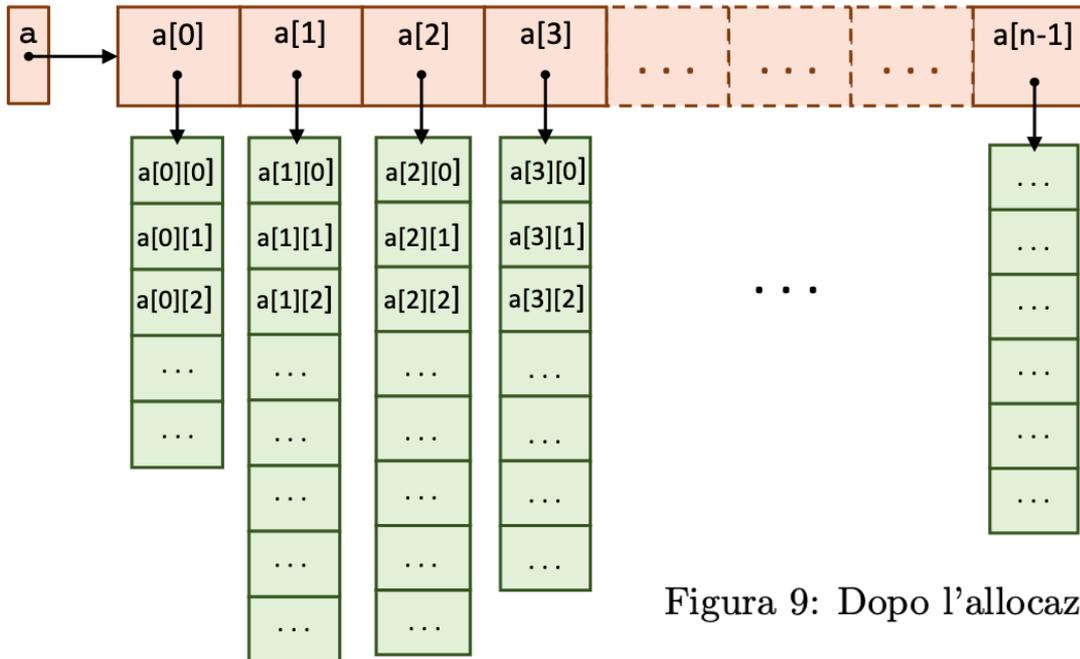


Figura 9: Dopo l'allocazione di ciascun vettore a[i].

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];
    } /* end for */
    return t;
}
```

Le righe hanno
lunghezza
variabile ($i+1$)

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 20

That's all Folks!

Grazie per l'attenzione...

...Domande?