Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Vettori in C
Aritmetica dei Puntatori

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **21**, 10 maggio 2022

Lezione 21a:

I vettori in Linguaggio C

I vettori

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, può essere vista come 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 un vettore

Classicamente, in C, un vettore **statico** è definito come segue:

```
T a[K];
```

dove `T` è il **tipo** degli elementi del vettore e `K` una **costante**, eventualmente simbolica definita con una `#define`. Gli elementi dell'array `a` sono numerati da 0 a `K-1`.

I vettori **statici** si possono inizializzare usando i simboli {...}. Ad esempio:

```
char s[4]={'r','o','m','a'};
```

dichiara un vettore di caratteri `s`, tale che `s[0]='r'`, `s[1]='o'`, `s[2]='m'`, `s[3]='a'`.

Con la dichiarazione:

```
int a[100]={0};
```

si dichiara il vettore `a` di 100 interi, in cui **tutti gli elementi dell'array** sono inizializzati a 0.

Memorizzazione di un vettore

Gli elementi di un'array, sono memorizzati in celle contigue di memoria. Ecco l'effetto della dichiarazione `int a[n]={0};`:

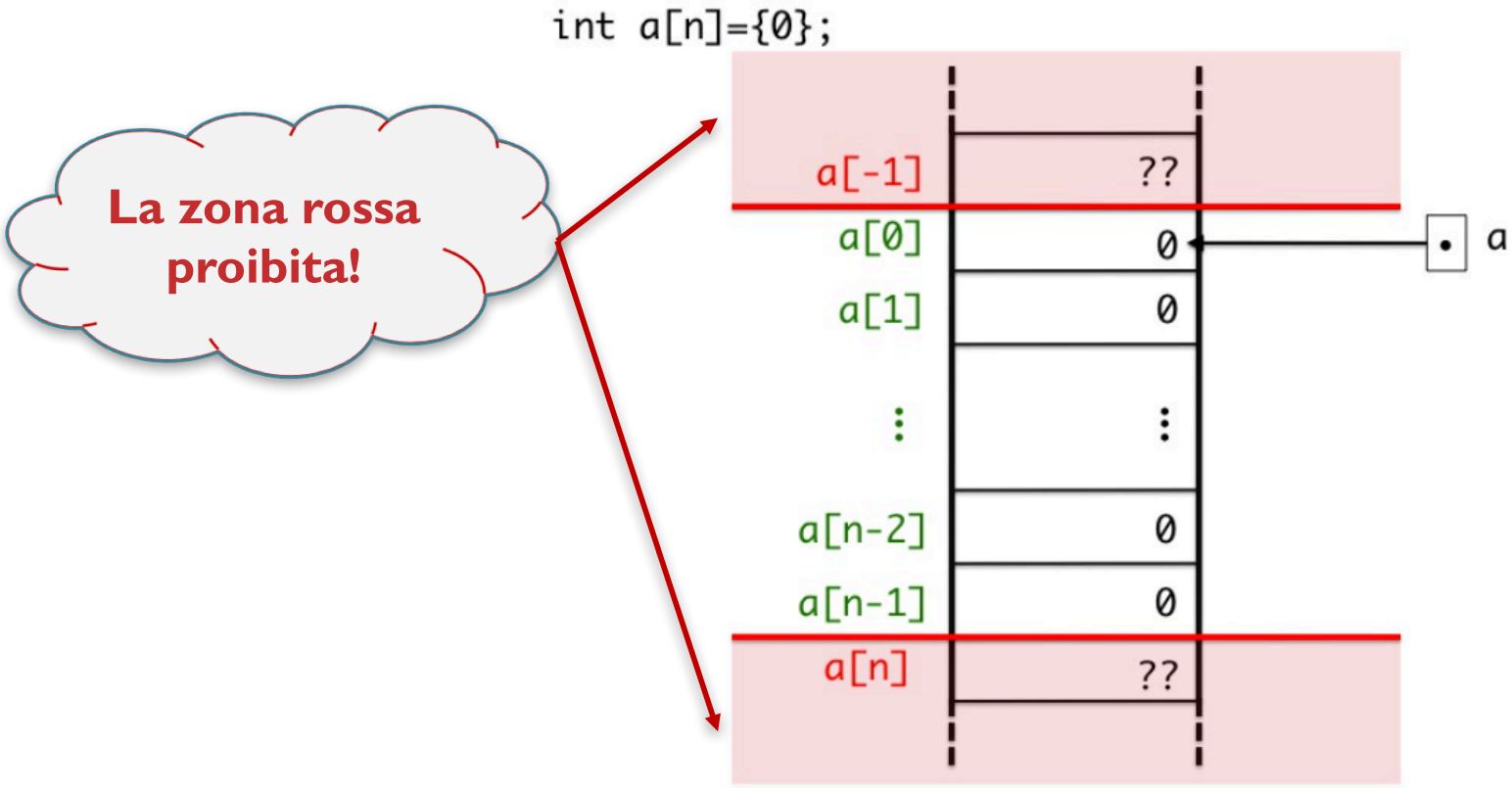


Figura 1: Effetto della dichiarazione di un vettore.

Lo strano caso di Dr. Array & Mr. Pointer

Visto che **i vettori** sono un'astrazione della memoria del calcolatore, in C essi **sono** visti **semplicemente** come **un puntatore**.

Dopo la dichiarazione `int a[n]={0};` la variabile `a` è semplicemente un `int*`.

Attenzione! la dichiarazione `int* b;` definisce una variabile compatibile per tipo con `a` **ma non alloca memoria**!

In C, le parentesi `[]` sono un **operatore di de-referenziazione**. Più precisamente, la scrittura:

```
a[i];
```

è del tutto equivalente a:

```
*(a+i);
```

cioè "**salta `i` posizioni dall'indirizzo base di `a`**": ne consegue che espressioni come `a[-1]` oppure `a[100]` sono perfettamente **lecite**, anche se **probabilmente errate**.

Uno sguardo oltre il C

In molti Linguaggi di programmazione moderni (ad esempio **Java**), **riferire un elemento esterno al dominio degli indici di un array causa un errore**, chiamato `ArrayIndexOutOfBounds`.

Per garantire questo comportamento, **il compilatore genera codice** (non esplicitamente scritto dal programmatore) che controlla ogni accesso. Significa che ad esempio un'istruzione tipo `a[e1]=e2` viene tradotta come fosse:

```
e=e1;
if (e<0 || e>K) throw ArrayOutOfBounds
else a[e]=e2;
```

Ovviamente in C, il linguaggio **non offre nessuna protezione** predefinita e si suppone che il programmatore eviti accessi incoerenti.

Attenzione: può causare errori difficili da scoprire! Come variabili non nominate dal programma che cambiano valore!

Primo programma con array: stampa

Quando si passa un parametro array, essenzialmente **si passa un puntatore** alla base del vettore, **non si ricopia l'array**!

Passando il parametro, **non si deve allocare memoria**!
Si usa la notazione: **`int a[]`** (senza numeri tra parentesi) oppure quella equivalente **`int *a`**.

Il vettore può essere scandito usando gli indici o la cosiddetta **aritmetica dei puntatori**.

```
void printv(char a[], int n){

  for (int i=0; i<n; i++)
     printf("%1c",*(a++));
  printf("\n");
  }
```

Figura 2: Stampa di un vettore – I

Qui uso notazione vettore per il parametro e aritmetica dei puntatori per scorrere a.

```
void printv(char* a, int n){

  for (int i=0; i<n; i++)
     printf("%1c",a[i]);
  printf("\n");
 }
```

Figura 3: Stampa di un vettore – II

Qui uso la notazione a puntatori per il parametro e gli indici per scorrere a.

Flessibilità dell'approccio C

Abbiamo visto che il C non offre protezione contro accessi errati a un array. Ma quali sono i **vantaggi**?

Possiamo usare la funzione `printV` per stampare una **porzione di vettore** tra due indici `inf` e `sup` usando la chiamata:

```
printV(&a[inf], sup - inf)
```

Ovviamente, da un punto di vista di chiarezza, se volete ottenere questo effetto, è meglio avere una funzione più generale opportunamente parametrizzata, come segue:

```
void printV(char a[], int inf, int sup){
   for (int i=inf; i<sup; i++)
      printf("%1c", a[i]);
   printf("\n");
```

Detour 1: il ciclo for

I vettori, usualmente, si scorrono usando il costrutto di controllo `for`. Nella sua **forma generale**, in **C** ha la forma:

```
for (C1; B; C2) C3
```

con la seguente semantica:

> "*Esegui il comando* `C1`, *ripeti il comando* `{C3; C2}` *finché l'espressione* `B` *non valuta a* `False`"

Quindi il ciclo `for` visto sopra è equivalente al seguente programma while:

```
C1; while (B){C3; C2}
```

Più usualmente, i cicli `for` servono a 'contare' usando un indice e si usano quando il numero di iterazioni è noto all'ingresso del ciclo, nella forma (**galateo**):

```
for (int i=0, i<n, i++) C
```

Detour 2: L'aritmetica dei puntatori

Abbiamo visto che è possibile scorrere un vettore v senza usare gli indici, ma usando l'espressione/comando v++.

Il significato dell'operatore postfisso v++ **dipende** dal **tipo di v**, e più precisamente dalla **quantità di memoria necessaria per memorizzare dati di quel tipo**.

Ad esempio, se v fosse un vettore di caratteri, v++ incrementerebbe il pointer v di 1, se v fosse di interi, v++ incrementerebbe il pointer v di 4, mentre se v fosse un vettore di long int, v++ incrementerebbe il pointer v di 8 etc.

Esercizio: Scrivere dei programmini che dimostrano (o sconfessano) queste affermazioni.

Lezione 19b:

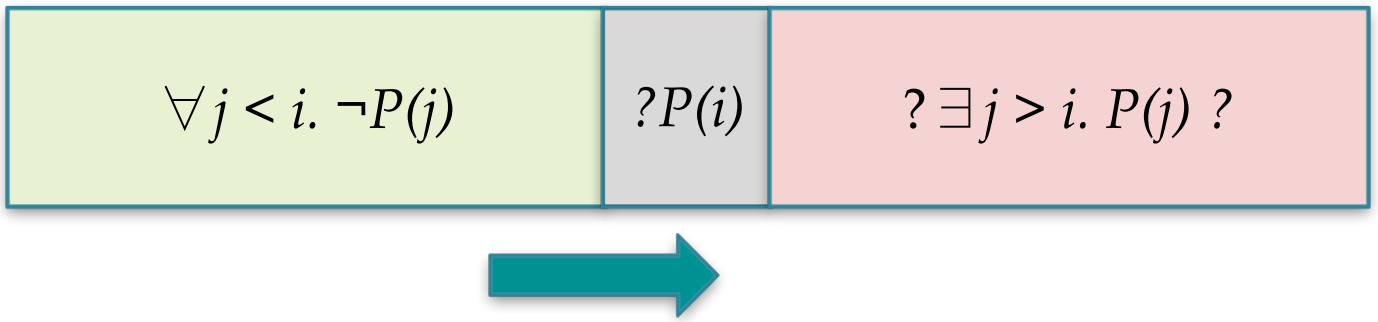
Programmi standard su vettori con asserzioni logiche

Schemi tipici di Programmi: ricerca

Immaginiamo di voler trovare un elemento $v[i]$ (se esiste) di un vettore di lunghezza n che **soddisfi** una certa proprietà $P(v[i])$ (spesso tale che $i = \min\{ j \mid P(v[j])\}$.)

Il problema può essere risolto da una scansione lineare del vettore in cui:

- l'**invariante è** $\varphi[i] = \forall j < i.\ \neg P(v[j])$ (ho escluso che esista un elemento di indice minore di i che soddisfi P)
- a ogni iterazione:
  - **esco con successo** se ho trovo i tale che $P(v[i])$
  - stabilisco $\varphi[i+1]$
- Esco **senza successo** avendo stabilito $\varphi[n]$.



Esempio: ricerca sequenziale

Si tratta del più semplice problema di ricerca. Continuo l'iterazione finché non trovo l'elemento.

Non appena lo trovo registro il successo. Se arrivo in fondo ritorno l'insuccesso.

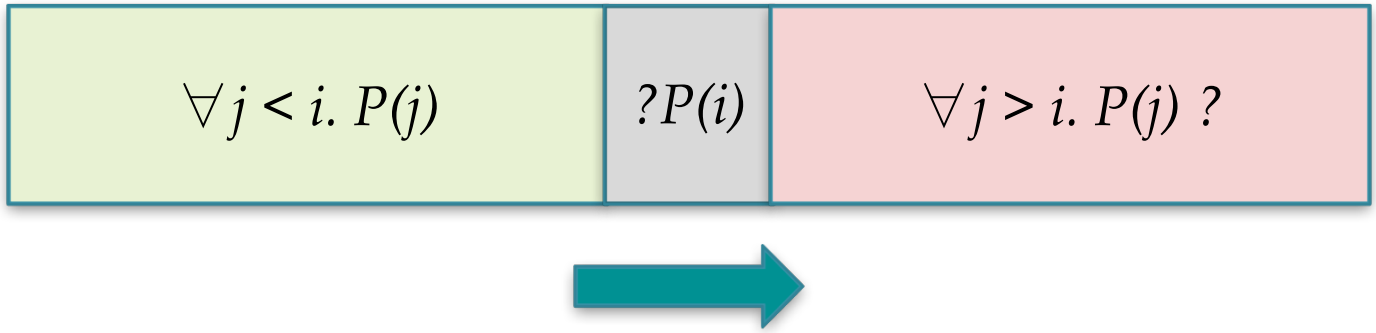
```
int ricercaSeq(int x, int a[], int n, int* j){
/* REQ: a vettore di lunghezza n
 * ENS: 1 se exists j. a[j]=x, 0 altrimenti
 * MOD: j (quando a[j]=x)
 */
   for (int i=0; i<n; i++)
    /* INV: forall j<i. a[j]!=x */
      if (a[i]==x){
            *j = i;
            return 1;
      }
    /* forall j<n. a[j]!=x */
   return 0;
}
```

Schemi tipici di Programmi: verifica

Immaginiamo di voler dimostrare che tutti gli elementi di un vettore di lunghezza n **soddisfino** una certa proprietà $P(v[i])$.

Il problema può essere risolto da una scansione lineare del vettore in cui:

- l'**invariante è** $\varphi[i] = \forall j < i.\ P(v[j])$ (ho dimostrato che tutti gli elementi fino a i soddisfano P)
- a ogni iterazione:
  - **esco con fallimento** se trovo i tale che $\neg P(v[i])$
  - stabilisco $\varphi[i+1]$
- Esco **con successo** avendo stabilito $\varphi[n]$.



Esempio: uguaglianza di due vettori

Si tratta di un tipico problema di **verifica**.

L'idea quindi è di **indebolire l'asserzione finale** e verificarla a ogni iterazione i fino all'indice $i < n$.

Se arrivo **all'iterazione n**, ho che la **verifica è conclusa** e riporto il successo.

```
int equalArrays(int a[], int b[], int n){
/* REQ: a, b vettori di lunghezza n
 * ENS: 1 se forall j. a[j]=b[j], 0 altrimenti
 * MOD: -
 */
   int b=1;
   for (int i=0; i<n && b=a[i]==b[i]; i++);
    /* INV: forall j<i. a[j]==b[i] */
   return b;
}
```

Esempio: minimo di un vettore

Questo caso è un po' un ibrido: l'idea è mantenere una proprietà $P(i, m) = \forall j<i.\ a[j] \leq a[m]$.

$P(1, m)$ è banalmente soddisfacibile con m=0.

Avendo $P(i, m)$ posso soddisfare $P(i+1, m')$ ponendo: $m'=m$ se $a[i] \geq a[m]$ e $m'=i$ se $a[i] \leq a[m]$ (si dimostra form. per transit. di ≤).

```
int min(int a[], int n, int* minIndex){
/* REQ: a vettori di lunghezza n
 * ENS: return m | forall j. a[j]<=m
 * MOD: m = a[minIndex]
 */
   int m=0; /* P(1,m) */
   for (int i=1; i<n; i++)
    /* INV: P(i,m) */
      if (a[i]<a[m]) m=i;
   /* P(n,m) */
    *minIndex = m;
   return a[m];
}
```

Lezione 21c:

Il problema del Baricentro

Il problema del Baricentro

Esercizio 3: *Diciamo che un indice k di un vettore di interi a lungo n ($0 \leq k < n$) è il* baricentro *di a, se la somma degli elementi di a prima di k è uguale alla somma degli elementi a cominciare da k fino ad n. Formalmente:* $\sum_{i=0}^{k-1} a[i] = \sum_{i=k}^{n-1} a[i]$.

Punto 1: *Si scriva una funzione C* `int baricentro(int a[], int n, int *k)` *che restituisce* 1 *se esiste almeno un baricentro nel vettore* `a` *e 0 altrimenti. Quando torna* 1, *essa carica nel parametro* `k` *l'indice del baricentro trovato. Si valuti la complessità della soluzione.*

Indicando per semplicità $sp_k = \sum_{i=0..k-1} a[i]$ e $ss_k = \sum_{i=k..n-1} a[i]$, il problema è una **banale applicazione di ricerca** di un indice k (se esiste) tale che $sp_k = ss_k$.

Usando una **funzione ausiliaria** (è sempre buono scomporre in sottoproblemi… o quasi!)

```
int sommaVet(int a[], int inf, int sup)
```

che calcola $\sum_{i=inf..sup-1} a[i]$ permette un codice molto chiaro e di semplice scrittura.

Il problema del Baricentro – Soluzione 1

```
int sommaVet(int a[], int inf, int sup){
   int s = 0;
   for (int i=inf; i<sup; i++) s+=a[i];
   return s;
}

baricentro(int a[], int n, int *k){
   for (int i=0; i<n; i++)
      if (sommaVet(a,0,i)==sommaVet(a,i,n){
            *k=i;
            return 1;
      }
   return 0;
}
```

*Solo quando torno 1 è rilevante caricare *k*

Costano complessivamente O(n) a ogni iterazione. Totale caso pessimo: $O(n^2)$

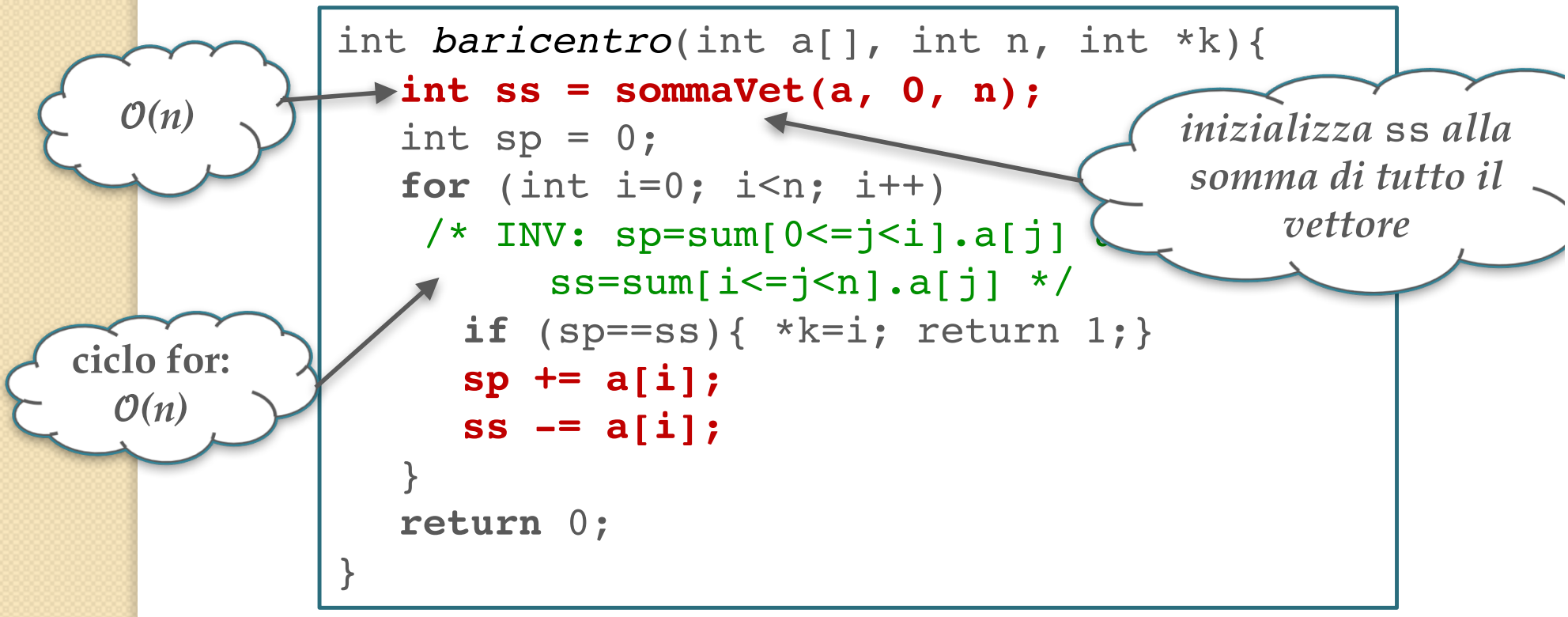
Il problema del Baricentro – Soluzione 2

Tuttavia… $sp_{i+1} = sp_i + a[i]$ e $ss_{i+1} = ss_i - a[i]$ e quindi **non occorre calcolare daccapo le somme ogni volta**.

Possiamo mantenere due variabili, *sp* e *ss*, uguali alla somma del prefisso e del suffisso e **mantenere cioè l'invariante**:

$$sp = sp_i \;\&\; ss = ss_i$$

ma come garantirlo all'inizio?



Baricentro: caso positivo (1)

Domanda 2: *Sotto la precondizione che gli elementi del vettore siano tutti positivi, è possibile migliorare l'efficienza della funzione (anche senza migliorare la complessità asintotica del caso pessimo)? Motivare la risposta (eventualmente con il codice di una funzione* `baricentroPos` *che sfrutta questa precondizione).*

In questo caso, si può osservare che le due sequenze $\{sp_i\}_{i=0..n-1}$ e $\{ss_i\}_{i=0..n-1}$ sono **monotone**, rispettivamente in senso **crescente** e **decrescente**.

Di conseguenza **si può uscire** dal ciclo `for` **non appena** ***sp*** **diventa maggiore di** ***ss*** riportando insuccesso ad esempio modificando la condizione del `for` come segue:

```
…
for (int i=0; i<n && sp <= ss; i++)
…
```

Non cambia la complessità asitontica, ma a volte si può risparmiare molto.

Baricentro: caso positivo (2)

Si può, tuttavia fare molto meglio (pur rimanendo lineari).

Siccome le due sequenze $\{sp_i\}_{i=0..n-1}$ e $\{ss_i\}_{i=0..n-1}$ sono monotone, fissati due indici $l < r$ ho che:

- Se $sp_l < ss_r$ allora il **baricentro**, se esiste, è nell'intervallo $(l, r]$
- Se $sp_l > ss_r$ allora il **baricentro**, se esiste, è nell'intervallo $[l, r)$

Quindi l'**invariante** $k \in [l, r]$ si mantiene nel **primo caso incrementando** l e nel **secondo caso decrementando** r.

Quindi: $r - l$ è una funzione di **terminazione** e $l < r$ la guardia.

Si può stabilire facilmente l'invariante all'ingresso del ciclo semplicemente **inizializzando** l **a** 0 ed r **a** $n-1$ perché banalmente ho che $k \in [0, n-1]$.

Come nel caso precedente, dovremmo anche mantenere la proprietà **invariante** $sp = sp_l$ & $ss = ss_r$ che stavolta si stabilisce inizialmente ponendo entrambe le variabili a 0 e poi incrementando sp quando si sposta l, e incrementando ss quando si sposta r.

Baricentro: caso positivo (3)

Il seguente programma fa al più una somma per ogni elemento, contro le 2 di quello precedente.

```
int baricentroPos(int a[], int n, int *k){
/* a vettore di lung. n e forall i a[i]>=0 */
   int ss = 0; int r = n-1; //ss = ss_r
   int sp = 0; int l = 0;   //sp = sp_l
   while (l < r)
    /* INV: sp=sum[0<=j<l].a[j] &&
            ss=sum[r<=j<n].a[j] &&
            k\in[l,r]
            Term: r - l
      */
      if (sp<ss) sp += a[l++];
            else ss += a[r--];:
   if (sp != ss) return 0;
   *k = l; // *k = r;
   return 1;
}
```

Notare che come spesso accade, gli indici da modificare sono quelli coinvolti nell'assegnamento

Baricentro ricorsivo (1)

Domanda 3: FACOLTATIVO: *Scrivere una funzione ricorsiva che risolve lo stesso problema con un'*unica *scansione del vettore* [**Sugg:** *scrivere una funzione ausiliaria con parametri aggiuntivi e ricordare che la scansione ricorsiva di un vettore, di fatto, percorre il vettore 2 volte, all'andata e al ritorno dalle chiamate ricorsive*].

Ovviamente uno può prendere una soluzione iterativa e trasformare i cicli in funzioni ricorsive con le tecniche standard.

Tuttavia questo porta a programmi **che scansioniamo il vettore più volte…**

La soluzione di questo problema è un virtuosismo, ma non è fine a sé stesso, perché permette di vedere tutto quello che si può fare durante una procedura ricorsiva:

1. trasmettere valori **in avanti** usando i parametri,

2. **condividere** informazione coi parametri passati per indirizzo,

3. raccogliere i valori di **ritorno** al **rientro** delle chiamate ricorsive.

Esercizio preparatorio (1)

Scriviamo una funzione ricorsiva che stampa un vettore:

```
void printVRec(char a[], int i, int n){
   if (i==n) printf("\n");
      else { printf("%1c", a[i]);
             printVRec(a, i+1, n);
           }
}
```

… oppure, una da Vero Programmatore C:

```
void printVRec(char* a, int n){
   if (!n) printf("\n");
      else { printf("%1c", *a);
             printVRec(a+1, n-1);
           }
}
```

ma che succede se **invertiamo** la **stampa** dell'elemento con la **chiamata ricorsiva**?

Esercizio preparatorio

Cosa stampa questa funzione?

```
void printVRec(char a[], int i, int n){
   if (i==n) printf("\n");
      else { printVRec(a, i+1, n);
             printf("%1c", a[i]);
           }
}
```

Stampa il **vettore rovesciato**!

Morale della favola: Una **scansione ricorsiva** di un vettore, attraversa in realtà il vettore **2 volte**, **una all'andata** delle chiamate ricorsive, e **una al ritorno** delle chiamate ricorsive.

Quindi: calcolariamo *sp* (somma dei prefissi) all'andata propagandola in avanti su un parametro ed *ss* (somma dei suffissi) al ritorno, trasmettendo il valore come risultato della funzione. Il fatto di aver trovato il baricentro viene registrato in un parametro passato per indirizzo.

Baricentro ricorsivo (2)

```
int baricentroRec(int a[], int n, int *k){
/* a vettore di lung. n */
   *k = -1;
   barRecAux(a, 0, n, 0, k);
   if (*k < 0) return 0;
   return 1;
}

int barRecAux(int a[], int i, int n,
                          int sp, int *k){
   /* REQ: sp=sp_i, 0<=i<=n
      ENS: ritorna ss_i
      MOD: *k è indice del bar. se trovato
    */
   if (i==n) return 0; // caso base: ss_n=0
   int ss = barRecAux(a, i+1, n, sp+a[i], k)
                  + a[i];
   if (*k >=0) return 42;
   if (sp == ss) *k=i;
   return ss;
}
```

osservare che la prima chiamata soddisfa REQ

passo sp_{i+1}

se *k>=0 ignoro ss

Lezione 21

That's all Folks!

Grazie per l'attenzione…

…Domande?