Tipi di Dato I: Vettori in Linguaggio C

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2016-17

La presente dispensa introduce all'uso dei vettori in linguaggio C. Verrà posta particolare enfasi ai legami tra vettori e puntatori, e allo sviluppo di programmi con vettori guidato da asserzioni logiche.

1 Vettori

Tutti i linguaggi di programmazione forniscono la possibilità di trattare dati complessi. Tutti i linguaggi imperativi (anche il più primitivo, il FORTRAN) offrono la possibilità di usare *vettori* (o *array*), in quanto un array è sostanzialmente un'astrazione della memoria: al programmatore in linguaggio macchina, infatti, la memoria appare come un unico enorme array in cui ciascun elemento è riferibile mediante il suo indirizzo (o puntatore).

Un vettore è una struttura dati che contiene un certo numero di elementi, ognuno dei quali è accessibile mediante un indice, che esprime la posizione che l'elemento ha all'interno dell'array. Da un punto di vista astratto, è come avessimo un numero di variabili che può cambiare in diverse esecuzioni del programma, e il cui nome possa essere calcolato durante l'esecuzione[1].

In C, una variabile vettore viene definita con una dichiarazione nella forma:

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

dove `T` è il tipo degli elementi dell'array, `a` è il nome dell'array, e `K` è una costante numerica nota al momento della compilazione del programma. Ad esempio `int a[100];` dichiara un vettore di 100 interi. Dopo la dichiarazione, nello spazio di memoria della funzione o del programma sarà allocato spazio per 100 interi. Con `char s[4]={'r','o','m','a'};` si dichiara un vettore di 4 caratteri, il cui stato iniziale è $s[0] =$ 'r', $s[1] =$ 'o', $s[2] =$ 'm', $s[3] =$ 'a'. Con la dichiarazione `int b[1000]={0};` si inizializzano a 0 tutti gli elementi del vettore `b`.

[1] questa osservazione è dovuta a E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976. Lettura che consiglio a tutti coloro che amano analizzare le finezze nascoste nei problemi semplici.

Ogni elemento dell'array sarà accessibile mediante l'indice, usando l'operatore `[ ]`: ad esempio, il quarto elemento dell'array `a` sarà acceduto con `a[3]`: infatti i 100 elementi dell'array `a` sono numerati da 0 a 99. Sarà utile sapere anche che gli elementi di un array vengono memorizzati in celle contigue di memoria. La Fig. 1 esemplifica graficamente l'effetto della dichiarazione di un array `int a[100]={0};`.

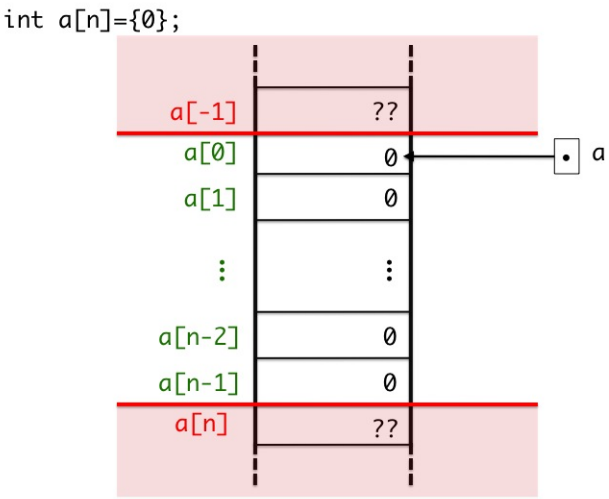


Figura 1: Effetto della dichiarazione di un vettore.

Infine è doveroso ricordare che i compilatori C più moderni ammettono la definizione di vettori con un numero di elementi *non prevedibile* al momento della compilazione. Detto in altri termini, la maggior parte dei compilatori moderni (incluso il `gcc`) non pone più il vincolo che il numero degli elementi sia una costante.

Un aspetto abbastanza importante da tenere presente è che, come sempre, il programma eseguibile ottenuto dalla compilazione di un programma C non offre *nessuna protezione* contro accessi incoerenti ai dati. Sempre riferendoci al nostro esempio dell'array `a` di 100 interi, osserviamo che mentre linguaggi "prudenti" genererebbero del codice che di fronte ad accessi come `a[-1]` o `a[100]` causerebbero l'arresto del programma in esecuzione con un'eccezione tipo `ArrayOutOfBounds`, i compilatori C generano codice che non batte ciglio e prosegue l'esecuzione. Questo atteggiamento del C aiuta i neofiti (ma a volte anche gli esperti) a scrivere codice che produce effetti assolutamente inaspettati, con errori difficili da scoprire. Quindi, evitate di sconfinare nella zona rossa!

1.1 Lo Strano Caso del Dr. Array e Mr. Pointer

Da un punto di vista tecnico, per il C, il tipo di un array, `T[]` è semplicemente un altro nome per il tipo `T*`. Osservate che il tipo `T[]` non contiene informazione relativa al numero di elementi. Ricordate però che in una dichiarazione di una

variabile vettore dovete specificare il numero di elementi come in `int a[100];` o `char b[n];`. L'effetto di tale dichiarazione è l'allocazione automatica anche di 100 (o n) celle di memoria, cosa che non avviene per effetto di una dichiarazione del tipo `int* a;`. Dopo la dichiarazione, il C tratterà la variabile `a` come un `int*` che *punta al primo elemento dell'array*: l'usuale scrittura con cui si accede a un elemento di un array, ad esempio `a[3]` per accedere al quarto elemento, altro non è che *zucchero sintattico* per `*(a+3)`, dove `+` va interpretato nell'*aritmetica dei puntatori*.

Detto questo, risulta chiaro che scritture come `a[-1]` sono perfettamente legittime, e hanno una *precisa semantica*, anche se logicamente sono tipicamente scorrette. Tale possibilità, infatti, genera errori subdoli (tipicamente si osservano variabili che cambiano valore senza motivo apparente – tenete conto che nel codice probabilmente non c'è scritto `a[-1]`, ma `a[E]` con `E` espressione intera che dipende da variabili).

Perché allora il compilatore C è così permissivo? Prima di tutto, osserviamo che il compilatore non può *staticamente* (cioè senza eseguire il programma, o se preferite il termine tecnico al *compile-time*) prevedere tutti i valori che l'espressione `E` potrebbe assumere in ogni possibile esecuzione. Quindi, per intercettare errori di questo tipo durante l'esecuzione (*run-time*), il compilatore deve *generare del codice* (non scritto dal programmatore) che protegge gli accessi al vettore. Ad esempio, una banale istruzione del tipo:

$$\texttt{a[}E_1\texttt{]=}E_2\texttt{;}$$

dovrebbe essere tradotta come se il programmatore avesse scritto:

$$\texttt{e=}E_1\texttt{; if (e>=0 \&\& e<aLun) a[e]=}E_2\texttt{; else throw(ArrayOutOfBounds);}$$

dove `e` è una variabile fresca, `aLun` deve essere correttamente valorizzata con la lunghezza del vettore `a` e la funzione `trhow` blocca l'esecuzione del programma informando l'utente che si è verificato un indirizzamento scorretto di un array. Quindi ogni accesso ad un elemento di un array costerebbe (almeno) un paio di controlli. Uno degli obiettivi del C, viceversa, è quello di permettere a programmatori esperti di scrivere codice molto efficiente.

C'è anche un secondo motivo: il C è stato storicamente (ed è tuttora) usato per scrivere importanti porzioni di sistemi operativi (tutti i sistemi operativi di famiglia Unix, quindi ad esempio Linux, Mac OS X, etc.) e dovendo sviluppare tale tipo di applicazioni è necessario un linguaggio che permetta all'occorrenza di avere pieno controllo della memoria e delle altre risorse della macchina.

1.2 Stampa di un Vettore

Cominciamo (Fig. 2 e Fig. 3) con lo scrivere una banale funzione che scrive a video gli elementi di un vettore di caratteri. La semplicità dell'esempio ci permette di concentrarci sul protocollo standard con cui un vettore si passa come parametro e ritornare sulla questione vettori e puntatori. Per complicare le idee al lettore (ma nel

```
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

```
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

suo bene ☺), abbiamo intenzionalmente incrociato nelle due procedure notazioni stile vettore con notazioni stile puntatore. Sappiate che sono perfettamente equivalenti!

La scrittura `char a[]` (con il vuoto tra le parentesi quadre) è permessa solo nei parametri e non nella dichiarazione di un vettore. Indica un vettore con un *numero ignoto di elementi*. Se così non fosse, la funzione `printv` dovrebbe essere riscritta per ogni differente dimensione del vettore (questa scelta disgraziata è stata fatta per esempio dal Pascal e da altri linguaggi di famiglia Algol). Nelle funzioni che trattano un vettore, usualmente il numero degli elementi n del vettore `a`, viene passato a sua volta nel secondo parametro `n` (intero).

Come sempre, una piccola nota di linguaggio C: quando l'operatore `x++` viene applicato dentro un'espressione, il valore della variabile `x` (necessario alla valutazione dell'espressione) viene valutato *prima* di eseguire l'incremento che modifica `x`. In Fig. 2, tutto funziona: nel comando: `printf("%1c",*(a++));` prima si valuta la variabile puntatore `a` poi si va a prendere il contenuto della cella di memoria puntata da `a` e solo a quel punto si esegue l'incremento su `a` che sposta in avanti il pointer all'interno del vettore. Se si volesse che l'incremento venga fatto *prima* della valutazione, è possibile usare l'operatore `++` in forma *prefissa*, cioè scrivendo `++a`. In questo caso, sarebbe scorretto, perché ci farebbe perdere il primo elemento e andrebbe a stampare il contenuto di una cella nella "zona rossa" oltre i confini del vettore.

Il chiamante chiamerà le funzioni sopra viste, con la chiamata `printv(a,n);`, dove `a` è il nome del vettore e `n` è la variabile che memorizza la sua lunghezza. Ricordate, infine, che quello che viene passato è *sempre l'indirizzo base del vettore*, e quindi i vettori vengono *sempre* passati per indirizzo, senza ricopiare il valore del vettore, o allocare memoria per memorizzare il vettore nell'activation record della funzione. Osservate inoltre che, mentre le modifiche a una generica cella `a[i]` si ripercuote sul chiamante, le modifiche al valore del puntatore `a` non influenzano il chiamante perché il puntatore è passato per valore. Quindi l'operazione `a++` all'interno della funzione `printv` non fa perdere il riferimento alla base del vettore al chiamante.

Questo modo di procedere ha vari vantaggi, per esempio il fatto che potete usare `printv` per stampare anche porzioni del vettore. Per esempio, avendo $0 \leq inf \leq$

$sup \leq n$, la chiamata `printv(&a[inf],sup-inf)` stamperà tutti gli elementi del vettore $a[j]$ con $inf \leq j < sup$. Osservate, che essendo il parametro formale `a` un pointer (in entrambi i casi(!) a dispetto delle scritture), non fa differenza se passo un puntatore alla base del vettore o un puntatore a un elemento che sta nel mezzo. Va da sé che una funzione opportunamente parametrizzata `printv(char a[], int inf, int sup)` (inizializzando `i` al valore di `inf` nel `for`) sarebbe da preferirsi per tale scopo, per un motivo di leggibilità del codice, anche se apparirebbe un po' prolissa all'occhio esperto del mio amico immaginario, il Vero Programmatore C.

1.3 Esercizi e Spunti di Riflessione

1. Scrivere una funzione C che stampa un vettore di caratteri in ordine inverso. Dare la versione iterativa e quella ricorsiva.
2. Scrivere una funzione C che calcola la somma degli elementi di un vettore di interi. Dare la versione iterativa e quella ricorsiva.
3. Scrivere una funzione C che stampa un sottoinsieme di elementi di un vettore di interi a "passo d", partendo da un certo indice inf (stampa perciò $a[inf + kd]$ al variare di k e finchè $inf + kd < n$, dove n è la lunghezza del vettore. Dare una versione iterativa e una ricorsiva.
4. Come nell'esercizio precedente, interpretando però il vettore come circolare ($a[n]$ viene interpretato come $a[0]$). Ovviamente bisogna fare un solo giro (e continuare finchè non si ritorna "nei pressi" dell'indice inf di partenza). Dare una versione iterativa e una ricorsiva.
5. Scrivere una funzione C, di prototipo `int palindromo(int a[], int n)` che restituisce 1 se il vettore a è palindromo, cioè se non fa differenza leggerlo da destra a sinistra o da sinistra a destra.
6. Scrivere una funzione C, di prototipo `void maxLoc(int a[], int n, int b[], int* l)` che carica nel vettore `b` i massimi locali del vettore `a`. Un certo elemento $a[i]$ è un massimo locale se $a[i] \geq a[i-1]$ & $a[i] \geq a[i+1]$ (attenzione che $a[0]$ è massimo locale se $a[0] \geq a[1]$ e $a[n-1]$ è massimo locale se $a[n-1] \geq a[n-2]$).

2 Programmi su Vettori con Asserzioni Logiche

In questa sezione vedremo la soluzione di qualche tipico problema sui vettori, approfondendo l'uso di asserzioni logiche nella soluzione di problemi sugli array; cercheremo di far vedere (nonostante si tratti di problemi molto semplici, la cui soluzione a molti sembrerà immediata) come le asserzioni logiche non solo servano a dimostrare la correttezza di un programma, ma siano uno strumento di guida al suo sviluppo.

Il lettore annoiato (a suo rischio e pericolo) può saltare ai problemi più impegnativi nelle prossime sezioni.

Notazioni Usiamo le seguenti convenzioni: x indica il valore di una variabile `x`. Quando dobbiamo occuparci della conservazione degli invarianti indicheremo con x' il valore che la variabile assumerà dopo l'esecuzione di una sequenza di comandi.

Le asserzioni e predicati logici in genere saranno indicati con lettere greche $\varphi, \psi, \phi, \ldots$ La scrittura $\varphi[x]$ vuole stigmatizzare che φ dipende da una variabile x, e può essere comoda quando vogliamo sostituire qualche valore a x.

Infine indicheremo con g l'asserzione logica associata alla guardia del ciclo che si sta esaminando.

2.1 Minimo di un Vettore

Il problema è molto semplice e consiste nel restituire il valore minimo di un vettore. Dato un vettore a lungo n, la funzione dovrà calcolare un valore min in modo che sia soddisfatta la seguente asserzione finale:

$$\varphi[a, n, min] = \forall j : 0 \leq j < n . min \leq a[j]$$

L'idea per risolvere questo problema è molto semplice ed è comune a molti programmi sui vettori: scandire il vettore da sinistra a destra memorizzando il minimo calcolato fino al punto in cui è arrivata la scansione del vettore, in modo da soddisfare ad ogni passo un *indebolimento* dell'asserzione finale, $\varphi[a, i, min]$, con $0 \leq i < n$. Più in dettaglio, si introduce una variabile `min` che viene inizializzata al valore di `a[0]` (che in effetti è anche il minimo del sottovettore contenente solo l'elemento `a[0]`). Si scandisce il vettore usando un indice `i`, avendo l'accortezza di riaggiornare la variabile `min` ogni qualvolta si trova un elemento `a[i]` che contiene un valore minore di quello già memorizzato in `min`. Forse è più semplice vedere il codice in Fig. 4, corredato dell'opportuno invariante $\varphi[a, i, min]$.

È relativamente facile verificare che l'invariante sia verificato all'entrata del ciclo. Infatti, esiste un solo j, tale che $0 \leq j < 1$, e la variabile `min` vale esattamente `a[0]`. Altrettanto evidente è che l'invariante e la negazione della guardia ($i = n$) implicano l'asserzione finale. $n - i$ è una funzione di terminazione, in quanto decresce ad ogni iterazione e assume valori positivi se la guardia del ciclo è verificata ($i < n$).

Infine è necessario verificare che l'invariante venga mantenuto dalla sequenza di comandi contenuta nel corpo del ciclo (state attenti che questi comprendono l'incremento della variabile `i` nell'intestazione del `for`). Dimostriamo che la proprietà invariante si conserva, cioè $\varphi[a, i, min] \wedge g \Rightarrow \varphi[a, i', min']$. Avremo:

$$\begin{aligned} i' &= i + 1 \\ min' &= \begin{cases} min & \text{se } a[i] \geq min \\ a[i] & \text{se } a[i] < min \end{cases} \end{aligned}$$

```
int minimo(int a[], int n){
 /* PREC: #a = n > 0
  * POST: min,  forall i in [0,n). min<=a[i]
  */
  int min=a[0];

  for (int i=1; i < n; i++)
  /* INV: forall j in [0,n). min<=a[j] */
     if (a[i] < min) min=a[i];

  return min;
 }
```

Figura 4: Calcolo del minimo di un vettore.

Dobbiamo ovviamente procedere distinguendo i due possibili casi. Nel primo abbiamo che $min = min'$ e $(\forall j : 0 \leq j < i.min \leq a[j]) \wedge (a[i] \geq min)$ implicano immediatamente $\forall j : 0 \leq j < i'.min' \leq a[j]$. Con poco più sforzo abbiamo che $(\forall j : 0 \leq j < i.min \leq a[j]) \wedge (a[i] < min)$ implica che $\forall j : 0 \leq j < i'.a[i] \leq a[j]$ (per transività di $\leq$) e quindi visto che in questo caso $min' = a[i]$, $\forall j : 0 \leq j < i'.min' \leq a[j]$. In entrambi i casi $\varphi[a, i, min] \wedge g \Rightarrow \varphi[a, i', min']$.

2.2 Uguaglianza di Due Vettori

Vediamo ora un problema simile, cioè scrivere una funzione che restituisce 1 se due vettori sono uguali, cioè se contengono gli stessi elementi nelle stesse posizioni. I parametri di ingresso saranno i due vettori a e b e la loro lunghezza n, che supponiamo uguale (chiaramente questa proprietà fa parte delle precondizioni).

Cominciamo con lo scrivere due formule logiche $\varphi[a, b, n]$ e $\psi[a, b, n]$ che esprimono formalmente il fatto che i vettori a e b sono rispettivamente, uguali e diversi:

$$\begin{aligned} \varphi[a, b, n] &\equiv \forall j : 0 \leq j < n.a[j] = b[j] \\ \psi[a, b, n] &\equiv \exists j : 0 \leq j < n.a[j] \neq b[j] \end{aligned}$$

Osservate che n può essere tranquillamente 0: i vettori di lunghezza zero sono tutti uguali e questo viene catturato dalle formule φ e ψ: infatti una proposizione del tipo $\forall x \in X.P(x)$ è sempre vera se X è vuoto, mentre una proposizione del tipo $\exists x \in X.P(x)$ è sempre falsa se X è vuoto. Osservate inoltre che $\varphi = \neg\psi$. Indicando infine con e il valore ritornato dalla funzione possiamo scrivere l'asserzione finale che deve soddisfare la funzione:

$$(e \wedge \varphi) \vee (\neg e \wedge \psi)$$

Il fatto che i vettori a e b soddisfino φ, può chiaramente essere stabilito solo verificando che *tutte* le uguaglianze prescritte dal quantificatore universale (su un

```
int uguali(int b[], int a[], int n){
 /* PREC: #a = #b = n >=0
  * POST: 1 if forall i in [0,n). a[i]==b[i]
  *    or 0 if exists i in [0,n). a[i]!=a[i]
  */

  int i=0;
  while (i<n && a[i]==b[i]) i++;
  /* INV: forall j in [0,i) a[j]==a[j] */

  return (i==n);
}
```

Figura 5: Verifica dell'uguaglianza di due vettori.

dominio finito!) siano soddisfatte. Quindi, l'asserzione finale suggerisce che è necessario verificare (uno alla volta) che gli elementi dei due vettori siano a due a due uguali, continuando fino a quando l'indebolimento $\varphi[a, b, i]$ dell'asserzione finale (positiva) sia soddisfatta nella porzione di vettore analizzata. $\varphi[a, b, i]$ sarà anche la nostra proprietà invariante. La verifica avrà termine quando ha luogo una delle due seguenti circostanze:

- ho dimostrato $\varphi[a, b, n]$, avendo scandito completamente i due vettori;
- ho dimostrato $\psi[a, b, n]$, avendo trovato una coppia di elementi diversi, cioè $a[i] \neq b[i] \wedge i < n$.

Quindi la guardia g del ciclo, altri non può essere che $a[i] = b[i] \wedge i < n$. Se esco dal ciclo perché $i = n$ allora significa che ho provato $\varphi[a, b, n]$, mentre se esco perché $a[i] \neq b[i]$ significa che ho provato $\psi[a, b, n]$. Osservate che nel secondo caso, necessariamente, $i < n$. Il codice, ancora una volta molto semplice, si scrive da solo in Fig. 5.

La verifica che $\varphi[a, b, i] \wedge g \Rightarrow \varphi[a', b', i']$ è ancora una volta molto semplice. Chiarito che $a' = a$ e $b = b'$ e $i' = i + 1$ (i vettori non vengono modificati), dovrebbe essere di per se evidente che $(\forall j : 0 \leq j < i.a[j] = b[j]) \wedge (a[i] = b[i])$ implica $(\forall j : 0 \leq j < i + 1.a[j] = b[j]) \equiv \varphi[a', b', i']$.

Forse la cosa più simpatica di questo esempio, è come viene restituito il valore finale: l'espressione `i==n` pur essendo, almeno in principio, un'espressione logica, può essere usata in C in luogo di un valore intero.

2.3 Ricerca Lineare (con Sentinella)

Questo problema consiste nel cercare un elemento x in un vettore a, e in caso affermativo restituire al chiamante un indice i per cui $x = a[i]$. Come vedremo dalle

```
int ricerca(int a[], int x, int n, int* w){
 /* PREC: #a = n >=0
  * POST: 1 if exists i in [0,n). a[i]==x, *w=i
  *       0 if forall i in [0,n). a[i]!=x
  */
  int i=0;
  while (i<n && a[i]!=x) i++;
  /* INV: forall j in [0,i) a[j]!=x */

 *w=i;
 return (i==n);
}
```

Figura 6: Ricerca lineare in un vettore.

asserzioni logiche, questo problema è fortemente imparentato coi precedenti, ma ci dà lo spunto di discutere un aspetto tecnico e una piccola finezza.

È comodo scrivere funzioni come quella in Fig. 6 che restituiscono come risultato l'esito della ricerca (1 se l'elemento è stato trovato e 0 altrimenti) e comunicano al chiamante l'indice occupato dall'elemento attraverso un parametro passato per indirizzo. Si potrebbe in effetti tornare come risultato direttamente l'indice, ma dovremmo ritornare un valore particolare per comunicare l'insuccesso. Sarebbe comodo ed elegante poter tornare un valore che viene valutato come `false` nel caso in cui l'elemento non sia trovato, ma 0 non si può usare a tale scopo, perchè 0, essendo un indice del vettore, è un possibile risultato positivo della funzione.

La funzione che abbiamo scritto può essere usata come nel seguente frammento:

```
    if (ricerca(a,x,n,&w)) a[w]++;
         else a[n]=x;
```

che incrementa la casella del vettore che contiene x, se presente nel vettore, oppure inserisce x alla fine della parte di vettore utilizzata.

Molto brevemente, la correttezza del programma in Fig. 6 si può stabilire scegliendo come invariante per il ciclo:

$$\varphi[a, i, x] \equiv \forall j : 0 \leq j < i.a[j] \neq x$$

e come asserzione finale:

$$(\neg e \wedge \varphi[a, n, x]) \vee (e \wedge \exists j : 0 \leq j < n.a[j] = x \wedge {}^*w = j)$$

Lasciamo al lettore il piacere di impratichirsi con asserzioni finali, invarianti e quant'altro. Dovrà semplicemente adattare i ragionamenti del problema precedente. Osserviamo che la specifica della funzione non dice nulla sul valore ${}^*w$ nel caso in cui la ricerca dia esito negativo.

Viceversa, occupiamoci di vedere un piccolissimo trucchetto che, senza modificare il comportamento asintotico del programma, *dimezza il numero dei confronti*. Si tratta di una sciocchezza, ma l'applicazione sistematica di questo genere di furbizie, può modificare in modo considerevole il comportamento dei programmi. La morale è: ogni treno impiega un tempo lineare nel numero dei chilometri che deve percorrere, ma fa molta differenza fare Roma-Venezia in 7 ore o in 4 ore!

Supponiamo di sapere che l'elemento x sicuramente occorra in a. Allora potremo non preoccuparci della condizione $i < n$ nella guardia, perchè si uscirà *sempre* dal ciclo grazie al fatto che *esiste sempre* un certo indice i_0 tale che $a[i_0] = x$. Ma allora perché non mettere noi x in a, ad esempio in posizione n? Aggiungiamo quindi l'istruzione `a[n]=x;` prima di entrare nel ciclo. Così facendo, non serve nemmeno modificare l'espressione con cui si torna il valore, in quanto se siamo usciti dal ciclo con $i = n$ significa che il ciclo è terminato per effetto dell'elemento fittizio posto in coda al vettore.

L'unica controindicazione è che, a rigore, la casella $a[n]$ appartiene alla zona rossa, e quindi a rigore è proibita e potrebbe, in generale, essere allocata per altre variabili. Tuttavia, è bene ricordare, che usualmente gli array vengono *sovradimensionati*. Essendo (tradizionalmente) memoria allocata *staticamente*, si prevede all'inizio un *numero massimo di elementi*, maggiore di quello che effettivamente si usano durante l'esecuzione del programma.

2.4 Crivello di Eratostene

I numeri primi si definiscono come quei numeri naturali che sono divisibili solo per 1 e per sé stessi. Per rendere più gradevoli gli enunciati di numerosi teoremi (pimo tra tutti l'unicità della scomposizione in fattori primi) è conveniente eliminare il numero 1 dall'esclusivo club dei numeri primi.

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."*

Occupiamoci di scrivere un programma che implementa questo algoritmo per stampare tutti i numeri primi, fino a un certo numero n. Bisogna trovare una *rappresentazione* all'idea di scrivere e cancellare un numero. Consideriamo un array p che

```
void crivello(int p[], int n){
 /* PREC: forall i in [0,n). p[i]=1
  * POST: forall i in [0,n).
              p[i]<=>primo(i)
  */

  for (int i=2; i*i<n; i++)
     if (p[i]) {
       for (int j=i*i; j<=n; j+=2*i)
          p[j]=0;
       }
  }
```

Figura 7: Crivello di Eratostene

```
void stampaPrimi(int p[], int n){
 /* PREC: forall i in [2,n).
          p[i]<=>primo(i)
  */

  for (int i=2; i<n; i++)
   if (p[i])
     printf("%4d ",i);
   printf("\n");
 }
```

Figura 8: Stampa dei primi generati.

useremo nel seguente modo: se $p[i] = 1$ allora il numero i non è stato cancellato, mentre se $p[i] = 0$ allora il numero i è stato cancellato.

Dovrebbe essere chiaro che cominceremo la nostra procedura con p che soddisfa l'asserzione $\forall i : 0 \leq i \leq n.p[i] = 1$ (cioè tutti i numeri sono ancora dei potenziali primi), mentre l'asserzione finale, sarà chiaramente $\varphi[p, n] \equiv \forall j : 2 \leq j \leq n.(p[j] = 1) \Leftrightarrow primo(j)$.

Il crivello di Eratostene impone di cancellare tutti i numeri composti cancellando i multipli dei numeri non ancora cancellati procedendo in modo crescente. Quest'idea si basa sul fatto che se ho cancellato i multipli di un certo k allora ho cancellato anche tutti i multipli di ogni k' tale che[2] $k \mid k'$. Detto in altri termini, l'insieme dei multipli di un numero contiene strettamente l'insieme dei multipli di un suo multiplo.

Speculando sulla nostra codifica del problema, è necessario scorrere l'array p a partire dall'indice 2, garantendo, in una generica iterazione che esamina il naturale i, di aver già cancellato tutti i composti dei primi fino a i. Quindi l'asserzione logica $\varphi[a, i] \equiv \forall i : 2 \leq i \leq n.p[i] = 0 \Leftrightarrow \exists k : 2 \leq k \leq i.i \mid k$, è il naturale candidato invariante del nostro processo iterativo e informalmente afferma che sono stati cancellati tutti i numeri divisibili per numeri più piccoli di i.

Dovrebbe essere facile dimostrare che se un numero n non ha divisori minori di $\sqrt{n}$, allora non ha divisori nemmeno maggiori di $\sqrt{n}$. Infatti $n = ab \ \wedge \ a < \sqrt{n} \Rightarrow b > \sqrt{n}$. Quindi, $\varphi[a, i] \ \wedge \ i \geq \sqrt{n} \Rightarrow \varphi[a, n]$. E quindi abbiamo anche la guardia candidata, cioè $i < \sqrt{n}$. A tale scopo, è preferibile di gran lunga usare la guardia `i*i<n` in luogo di `i<sqrt(n)` per vari motivi. `sqrt` è una funzione di libreria che probabilmente approssima il valore della radice quadrata utilizzando opportuni algoritmi numerici approssimati. Un prodotto, viceversa, è praticamente un'istruzione macchina, e ci fa rimanere nel mondo dei numeri interi.

[2]La notazione $k \mid k'$ significa "k divide k'".

Rimane da scrivere un comando per cui partendo da una situazione in cui vale $\varphi[p,i]$, allora valga $\varphi[p,i+1]$. Se $p[i]=0$, non devo fare nulla, perchè i non è primo, e avendo già analizzato i suoi fattori, sono già stati cancellati anche i suoi multipli. Se $p[i]=1$ allora devo semplicemente cancellare tutti i multipli di i minori di n. Dovrò perciò scrivere un comando che garantisca l'asserzione $\psi[n,i] = \forall k : i < k < n.i \,|\, k \Rightarrow p[k] = 0$. Il modo più elegante ed efficiente è "camminare" sui multipli di i, partendo da $2i$ (è sufficiente partire da i^2, verificate personalmente!), facendo passi lunghi i (per $i > 2$, posso fare passi lunghi $2i$, evitando i pari). Lo squisito programma risultante è mostrato in Fig. 7. Per stampare i numeri primi, occorrerà stampare tutti gli indici i, tali che $p[i]=1$. A titolo di completezza, anche la funzione `stampaPrimi` è visualizzata in Fig. 8. Un programma completo, dovrà prima inizializzare il vettore `p` a tutti 1, poi invocare la funzione `void eratostene(int p, int n)` e infine invocare la funzione `stampaPrimi(p, n)`.

2.5 Coefficienti Binomiali

Ricordiamo la definizione di coefficiente binomiale ($n \geq 0, 0 \leq k \leq n$):

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Il loro uso è pervasivo, visto che il coefficiente binomiale $\binom{n}{k}$ è per esempio il coefficiente del monomio $a^k b^{n-k}$ nello sviluppo di $(a+b)^n$, oppure il numero dei sottoinsiemi di k elementi generati a partire da un insieme di n elementi. Prima di tutto, osserviamo che, volendo scrivere una funzione che calcola i coefficienti binomiali, è assolutamente sconsigliato usare la definizione data sopra per scrivere il seguente programma:

```
unsigned long int cbin(int n, int k){
   return fatt(n)/(fatt(k)*fatt(n-k));
}
```

che brilla in concisione, ma ha l'enorme difetto di invocare la funzione `fatt` (che ovviamente calcola il fattoriale) su numeri potenzialmente molti grandi, a rischio di far uscire il risultato dal campo intero, anche quando il coefficiente binomiale sarebbe un numero non molto grande. Ad esempio 90! è un numero decisamente poco gestibile, ma il numero delle cinquine al Lotto $\binom{90}{5}$ è un numero decisamente ragionevole: 43.949.268 e il programma sopra scritto non lo potrebbe calcolare.

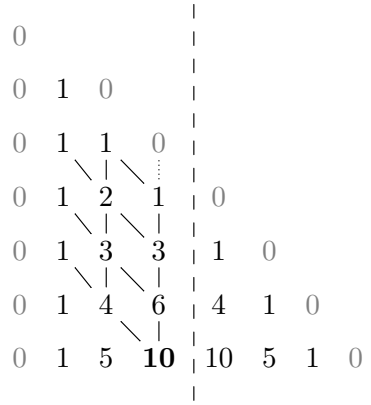
Un po' meglio va osservando che $\frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{(n-k)!}$. Per esercizio potete provare a utilizzare questa idea. Comunque, anche questa procedura causerebbe il calcolo di numeri più grandi del necessario, anche avendo cura di

fare le divisioni non appena possibile (cioè quando la divisione dà risultato intero). Inoltre, fare somme è meglio che fare prodotti.

Di conseguenza è preferibile scrivere un programma che sfrutta la seguente notevole proprietà dei coefficienti binomiali:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \text{e} \quad \binom{n}{0} = \binom{n}{n} = 1$$

Queste relazioni portano alla costruzione del cosidetto *Triangolo di Tartaglia* (o di Newton, o di Pascal... a seconda voi siate in Italia, Inghilterra o Francia ☺), che abbiamo riportato qui sotto (contornato di zeri fantasma, su cui torneremo più tardi):



Questa relazione porta ad un immediato ed elegante programma ricorsivo:

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

che purtroppo soffre della stessa sindrome (in forma più acuta) della funzione ricorsiva che calcola i numeri di fibonacci, e va a ricalcolare più volte coefficienti binomiali già calcolati. Il modo migliore di procedere per calcolare il coefficiente binomiale $\binom{n}{k}$ è quello di generare le prime n righe del triangolo di Tartaglia.

Come "dimostrato" dalla figura, inoltre, è sufficiente generare le righe fino alla posizione k. Nella figura, si vede che il coefficiente binomiale $\binom{5}{3}$ dipende solo da coefficienti binomiali $\binom{n'}{k'}$ con $n' < 5$ e $k' \leq 2$.

A questo punto non ci resta di vedere come sia possibile generare la $n+1$-esima riga del triangolo di Tartaglia, avendo a disposizione in un vettore `t` l'n-esima.

```
unsigned long int cbin(int n, int k){
   unsigned long int t[k+1];

   if (k>n-k) return cbin(n, n-k);

   for (i=1; i<=k; i++) t[i]=0;
   t[0]=1;

   for (int i=1; i<=n; i++)
       for (int j=k; j>0; j--) t[j]=t[j]+t[j-1];

   return t[k];
}
```

Figura 9: Calcolo dei coefficienti binomiali.

Saremmo tentati di usare un vettore di appoggio `s` e scrivere quanto segue (avendo il vettore `t` memorizzato la riga precedente (compreso lo zero fantasma)):

```
for (j=2; j<=k; j++) s[j]=t[j]+t[j-1]
```

Dovendo, ovviamente, poi ricopiare `s` in `t` per prepararsi all'iterazione successiva. Una furbizia tuttavia ci fa risparmiare parecchio: andando all'indietro questo problema non c'è. Infatti l'operazione `t[j]=t[j]+t[j-1]` distrugge il valore precedente di `t[j]`, ma se sto andando da destra a sinistra, questo valore non è necessario per calcolare i valori che stanno a sinistra di `t[j]`.

Un'ultima furbizia ci porta al programma in figura: ovviamente, il triangolo di Tartaglia è simmetrico, ed infatti $\binom{n}{k} = \binom{n}{n-k}$, e chiaramente, conviene applicare la procedura vista sopra per il minore tra k e $n-k$.

Un'ultima osservazione relativa alla dichiarazione dell'array `t`: tradizionalmente gli array vengono dichiarati sovradimensionati come variabili globali o al più nel main con un *numero massimo* di elementi *costante* (cioè non dipendente dall'esecuzione del programma). Il C standard non prevede dichiarazioni come quella fatta nella funzione `cbin`, ma la maggior parte dei compilatori la ammettono (compreso il `gcc`). Ci sono alcune piccole differenze con gli array "tradizionali", che dal vostro punto di vista potete ignorare: per esempio non è ammessa l'inizializzazione con la notazione a parentesi graffe {}). A differenza dei *veri* vettori dinamici, vengono comunque memorizzati nel record di attivazione della funzione. Quindi, non tornateli come risultato di una funzione!

2.6 Esercizi e Spunti di Riflessione

1. ♣ Valutare il numero di chiamate del tipo `cbin(n,0)` generate da un'esecuzione della funzione ricorsiva per i coefficienti binomiali, in funzione di n e k.

2. Scrivere una funzione C che calcola il *prodotto scalare* di due vettori di interi di uguale lunghezza. Ricordiamo che il prodotto scalare di due vettori a e b (nelle nostre notazioni) è $\sum_{i=0}^{n} a[i] \times b[i]$.

3. Due array a e b, di lunghezza rispettivamente n ed m sono *simili* se contengono gli stessi elementi. Formalmente se:

$$\forall i : 0 \leq i < n.\ \exists j :\ 0 \leq j < m . a[i] = b[j]\ \&$$
$$\forall i : 0 \leq i < m.\ \exists j :\ 0 \leq j < n . b[i] = a[j]$$

   Attenzione che l'array {1, 1, 3} è simile all'array {1, 3, 1}, ma anche all'array {3, 3, 3, 1}, ma non all'array {1, 2, 3}. Scrivere una funzione che verifica se due array di interi sono simili.

4. Dati due array a e b di uguale lunghezza, a è *permutazione* di b, se a contiene esattamente gli stessi elementi b, non necessariamente nelle stesse posizioni, ma eventuali elementi ripetuti devono avere lo stesso numero di occorrenze in entrambi gli array. Attenzione quindi che l'array {1, 1, 3} è permutazione dell'array {1, 3, 1}, ma non dell'array {3, 3, 1}. Scrivere una funzione che verifica se due array di interi sono permutazioni.

5. Un vettore di caratteri a è *immerso* nel vettore b se tutti i caratteri di a compaiono in b nello stesso ordine (ma non necessariamente consecutivamente). Ad esempio, la sequenza "*abba*" è immersa nella sequenza "*abracadabra*", ma non nella sequenza "*baobab*". Scrivere una funzione C che ricevendo due array di caratteri a e b e relative lunghezze restituisca 1 se a è immerso in b, -1 se b è immersa in a e 0 altrimenti.

6. Scrivere una funzione `C` che, ricevendo un array a di lunghezza n restituisce la sequenza non decrescente più lunga. Un possibile prototipo della funzione è: `void maxSeq(int a[], int n, int* i, int* l)`, dove si suppone di caricare nel parametro `i` il punto di inizio e in `l` la lunghezza della sequenza più lunga. Per esempio, avendo in input il seguente array di lunghezza 10:

   1 4 9 2 3 3 11 4 1 4

   il programma calcola come sequenza non decrescente più lunga:

   2 3 3 11

   e quindi restituisce i valori 3 (indice di inizio) e 4 (lunghezza).

7. Scrivere una funzione `C` che, dato un array `a` di interi che contiene come valori solo 0 e 1, calcola, se esiste, un indice j per cui, contemporaneamente il numero di 0 e di 1 che occorrono nel sottovettore $a[0], a[1], \ldots, a[j-1]$ è uguale rispettivamente al numero di 0 e di 1 che occorrono nel sottovettore $a[j], a[j+1], \ldots, a[n-1]$ (questo numero può anche essere 0, se il vettore

di soli 1 o di soli 0 – in tal caso sarebbe sufficiente verificare che il vettore abbia lunghezza pari). Ad esempio, dato il vettore $\{0, 0, 1, 0, 0, 1, 0, 0\}$, il programma dovrebbe calcolare il numero 4. Nel vettore $\{0, 0, 1, 1\}$ invece questa proprietà non è soddisfatta da nessun indice, in quanto o il numero di 0 o di 1 non è mai bilanciato correttamente. La funzione dovrebbe avere prototipo `int bilanc(int a[], int n, int* j)`, con l'idea che risponda 1 se tale indice esiste, caricando il suo valore in `j` e 0 altrimenti. Volendo, potete provare a scrivere una funzione ricorsiva che scandisce una sola volta il vettore, aiutandovi con qualche parametro ausiliario. E se vi sentite impavidi, provate a generalizzare il problema al caso in cui il vettore contenga solo valori compresi in un certo intervallo $[a, b]$ e la funzione deve trovare, se esiste il punto in cui sono bilanciate le occorrenze di ogni valore k, tale che $a \leq k \leq b$. In tal caso, passare tra i parametri anche i valori a e b.

8. Scrivere una funzione C, `void mink(int a[], int n, int m[], int k)` che carica nel vettore `m`, i k valori più piccoli di `a` (chiaramente richiedere $k \leq n$).

9. Scrivere una funzione C che calcola il sottovettore di somma massima. Un possibile prototipo della funzione è: `int maxSubSom(int a[], int n, int* i, int* l)`, che ritorna come risultato la somma massima trovata e carica nel parametro `i` il punto di inizio e in `l` la lunghezza della sequenza di somma massima (ovviamente questo problema è molto interessante se il vettore contiene anche numeri negativi).

10. Scrivere una funzione C di ordinamento che ordina un vettore in modo crescente supponendo che sia ordinato in ingresso in ordine decrescente. Scrivere le precondizioni e gli invarianti. Valutate la complessità del vostro algoritmo.

11. (Incastro) L'incastro è uno schema enigmistico che, partendo da due parole o frasi, ne ricava una nuova inserendo la seconda entro la prima. Può essere dunque sintetizzato dallo schema enigmistico $XX/Y = XYX$, laddove le due X stanno a indicare le parti da separare nella prima parola. Scrivere una funzione C `int incasto(char x[], int n, char y[], int m, char z[], int p)` che verifica se il vettore di caratteri `z` è un incastro tra i vettori `x` e `y`.

12. Un array di caratteri rappresenta una espressione aritmetica (le cifre sono anche caratteri, ma non confondete il numero 1 con il carattere '1'!). Scrivere una funzione C che verifica se le parentesi siano bilanciate, cioè se in ogni punto, il numero di parentesi aperte è sempre maggiore del numero delle parentesi chiuse (supporre che ci siano solo parentesi tonde eventualmente annidate). Provare inoltre a scrivere una versione ricorsiva che fa un'unica scansione dell'array.