# Nota

È considerato errore qualsiasi output non richiesto dagli esercizi.

# Esercizio 1 (punti 7)

Creare i file `array.h` e `array.c` che consentano di utilizzare la seguente funzione:

```
extern void array_remove (double *arr, size_t *pn, size_t pos);
```

La funzione accetta come parametri un puntatore a un vettore di double `arr`, un puntatore a un dato di tipo `size_t` che punta ad una variabile che ne indica la dimensione `pn`, una posizione `pos`.
Se la posizione `pos` è all'interno dell'array, la funzione deve eliminare dall'array l'elemento alla posizione `pos` spostando indietro di una posizione tutti gli elementi successivi opportunamente. Se riesce ad eliminare un elemento, la funzione deve modificare la variabile contenente la dimensione decrementandola di uno, altrimenti deve lasciare la dimensione inalterata.
`arr` e `pn` non saranno mai `NULL`. La dimensione dell'array può essere 0.

# Esercizio 2 (punti 6)

Nel file `binomiale.c` implementare la definizione della funzione:

```
extern double binomiale(unsigned int n, unsigned int k);
```

La funzione accetta come parametri due numeri naturali e restituisce il coefficiente binomiale corrispondente:

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

dove il ! indica il fattoriale (si ricorda che $0! = 1$). Nel caso `n` sia 0 o `k` sia maggiore di `n` la funzione ritorna -1. Visti i valori elevati coinvolti nel calcolo si suggerisce di utilizzare il tipo double all'interno della funzione durante i calcoli.

# Esercizio 3 (punti 7)

Creare i file `read_file.h` e `read_file.c` che contengano la definizione

```
typedef unsigned char byte;
```

e consentano di utilizzare la seguente funzione:

```
extern byte *read_file(const char *szFileName, size_t *cb);
```

La funzione accetta come parametro un nome di file che deve essere aperto in lettura in modalità non tradotta (binario) e un puntatore ad una variabile di tipo `size_t` in cui si dovrà inserire il numero di byte letti dal file.
La funzione deve ritornare un puntatore ad una nuova zona di memoria (allocata dinamicamente nell'heap) contenente tutti i byte letti dal file. La dimensione del file non è nota a priori e non può essere vincolata dal codice. Se il file non può essere aperto, ad esempio perché non esiste, la funzione ritorna `NULL` e non modifica la variabile puntata da `cb`.

Sul sito vengono forniti i file `prova.txt` e `aip.txt` per testare il codice.

# Esercizio 4 (punti 8)

Creare i file `matrix.h` e `matrix.c` che consentano di utilizzare la seguente struttura:

```
struct matrix {
	size_t N,M;
	double *data;
};
```

e la funzione:

```
extern void matrix_write(const struct matrix *matr, FILE *f);
```

La struct consente di rappresentare matrici di dimensioni arbitraria, dove N è il numero di righe, M è il numero di colonne e data è un puntatore a N×M valori di tipo double memorizzati per righe. Consideriamo ad esempio la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

questo corrisponderebbe ad una variabile struct matrix A, con A.N = 2, A.M = 3 e data che punta ad un area di memoria contenente i valori {1.0, 2.0, 3.0, 4.0, 5.0, 6.0 }.

La funzione deve scrivere la matrice matr sul file f (già aperto) in formato testuale decimale inviando il valore di N seguito da <a capo>, il valore di M seguito da <a capo>, i valori della matrice separati da <tabulazione> (\t in linguaggio C) all'interno della riga e con un <a capo> alla fine di ogni riga (compresa l'ultima).

Facendo riferimento all'esempio precedente, A verrebbe scritta come

```
2↵
3↵
1.000000	→	2.000000	→	3.000000↵
4.000000	→	5.000000	→	6.000000↵
```

Visualizzando in una tabella i singoli caratteri si vedrebbe

| | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | \n | | | | | | | | | | | | | | | | | | | | | | | |
| 3 | \n | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | . | 0 | 0 | 0 | 0 | 0 | 0 | \t | 2 | . | 0 | 0 | 0 | 0 | 0 | 0 | \t | 3 | . | 0 | 0 | 0 | 0 | 0 | 0 | \n |
| 4 | . | 0 | 0 | 0 | 0 | 0 | 0 | \t | 5 | . | 0 | 0 | 0 | 0 | 0 | 0 | \t | 6 | . | 0 | 0 | 0 | 0 | 0 | 0 | \n |

# Esercizio 5 (punti 5)

Nel file `histo.c` implementare la definizione della funzione:

```
extern void histogram(const char *values, size_t n);
```

La funzione accetta come parametro `values`, un puntatore ad una zona di memoria contenente un vettore di valori numerici interi da 0 a 127 e un dato di tipo `size_t` che ne indica la dimensione `n`. La funzione, per ogni valore, invia su standard output una sequenza di tanti caratteri `*` quanto è il valore in esame, seguita da un a capo. Ad esempio, chiamando la funzione con values={1,2,3} e n=3, sul file troveremmo:

```
*↲
**↲
***↲
```

una sorta di istogramma orizzontale corrispondente ai valori contenuti in `values`. Se `values` è `NULL` o `n=0`, la funzione non invia nulla in output.