# Esercizio 1 (5 Punti)

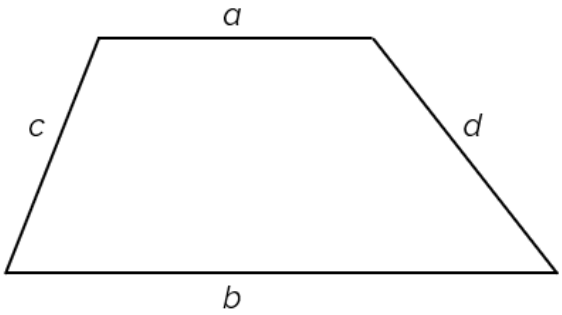
Nel file `area_trapezio.c` inserire la definizione della funzione:

```
extern double area_trapezio(double a, double b, double c, double d);
```

La funzione calcola l'area del trapezio di lati a, b, c e d, utilizzando la formula di Bhāskara I:

$$Area = \frac{1}{2}(a+b)\sqrt{c^2 - \frac{1}{4}\left((b-a) + \frac{c^2-d^2}{b-a}\right)^2}$$

La formula prevede che a e b siano i lati paralleli, con b > a. Questa condizione sarà sempre rispettata dai parametri della funzione.




# Esercizio 2 (6 Punti)

Nel file `partition.c` inserire la definizione la funzione:

```
extern size_t find_first_partition(const int *seq, size_t n);
```

Data una sequenza di numeri, la funzione ritorna l'indice del primo elemento maggiore di tutti i precedenti e minore di tutti i successivi. Il primo elemento della sequenza viene considerato "maggiore di tutti i precedenti" e l'ultimo viene considerato "minore di tutti i successivi", pur non avendo rispettivamente precedenti o successivi. Ritornare n se non esiste un elemento con tale proprietà. Se `seq` è `NULL`, o se n è `0`, ritornare `0`;

Alcuni esempi:

```
Input: seq = {5, 1, 4, 3, 6, 8, 10, 7, 9}, n = 9
Output: 4
```

Perché l'elemento di indice 4 (il valore 6) è maggiore di 5,1,4,3 e minore di 8,10,7,9.

```
Input: seq = {11, 1, 4, 3, 6, 8, 10, 7, 9}, n = 9
Output: 9
```

Nessun elemento soddisfa la proprietà, perché 11 non è minore di tutti i successivi e tutti i successivi non sono maggiori di 11. Quindi ritorniamo 9, la lunghezza della sequenza.

```
Input: seq = {1, 2, 4, 3, 6, 8, 10, 7, 9}, n = 9
Output: 0
```

Perché l'elemento di indice 0 (il valore 1) non ha elementi precedenti ed è minore di tutti i successivi.

```
Input: seq = {1, 1, 4, 3, 6, 8, 10, 7, 9}, n = 9
Output: 4
```

Perché l'elemento di indice 4 (il valore 6) è maggiore di 1,1,4,3 e minore di 8,10,7,9. Non possiamo ritornare 0 o 1 perché l'elemento di indice 0 non è minore di quello di posizione 1 (è uguale) e lo stesso vale per l'altro.

# Esercizio 3 (7 Punti)

Nel file `rimuovi_singoli_spazi.c` inserire la definizione della funzione:

```c
extern char *rimuovi_singoli_spazi(const char *s);
```

La funzione accetta come parametro una stringa C `s`, e ritorna una nuova stringa C allocata dinamicamente su heap. La nuova stringa deve essere formata eliminando da `s` i singoli spazi, ma non le sequenze di più spazi consecutivi.

Ad esempio:

```
Input: " a b c "
Output: "abc"
```

```
Input: "  a  b  c  "
Output: "  a  b  c  "
```

```
Input: "  abc   def ghi   jkl    mno pqr  s "
Output: "  abc   defghi   jkl    mnopqr  s"
```

# Esercizio 4 (8 Punti)

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

```c
struct matrix {
    size_t rows, cols;
    double *data;
};
```

e la funzione:

```
extern struct matrix *mat_pad(const struct matrix *mat);
```

La struct consente di rappresentare matrici di dimensioni arbitraria, dove `rows` è il numero di righe, `cols` è il numero di colonne e `data` è un puntatore a `rows×cols` 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.rows = 2`, `A.cols = 3` e `A.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 `mat_pad()` accetta come parametro un puntatore a matrice `mat`, e restituisce il puntatore ad una nuova matrice allocata dinamicamente su heap. La nuova matrice è ottenuta effettuando un *padding* di `mat`:

- Tra ogni coppia di colonne consecutive, inserire una nuova colonna di soli zeri;
- Tra ogni coppia di righe consecutive, inserire una nuova riga di soli zeri.

Ad esempio:

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

$$Output = \begin{pmatrix} 1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 \end{pmatrix}$$

Oppure:

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

$$Output = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 0 & 0 \\ 3 & 0 & 4 \\ 0 & 0 & 0 \\ 5 & 0 & 6 \end{pmatrix}$$

Il puntatore alla matrice `mat` non sarà mai `NULL`.

# Esercizio 5 (7 Punti)

Creare il file `product.h` e `product.c` che consentano di utilizzare la seguente struttura:

```
struct product {
    char *productId;
    char *fullName;
    int price;
}
```

E la funzione:

```
extern void write_products(FILE *f, const struct product* list, size_t n);
```

La funzione accetta come parametri un puntatore a file già aperto in scrittura `f` ed un puntatore a vettore di `struct product` contenente `n` elementi.

La funzione stampa il vettore `list` su `f` nel formato JSON seguente:

```
[
<tab>{
<tab><tab>"productId":<spazio>"@productId",
<tab><tab>"fullName":<spazio>"@fullName",
<tab><tab>"price":<spazio>@price
<tab>},
<tab>{
<tab><tab>"productId":<spazio>"@productId",
<tab><tab>"fullName":<spazio>"@fullName",
<tab><tab>"price":<spazio>@price
<tab>},
...
<tab>{
<tab><tab>"productId":<spazio>"@productId",
<tab><tab>"fullName":<spazio>"@fullName",
<tab><tab>"price":<spazio>@price
<tab>}
]
```

Gli elementi compresi tra parentesi angolari `<>` sono caratteri speciali (tab e spazio). Gli elementi espressi nella forma **@var** indicano il valore della variabile `var`.

Un esempio di output valido è il seguente:

```
[
    {
        "productId": "289347",
        "fullName": "Pizza Margherita",
        "price": 6
    },
    {
        "productId": "67832",
        "fullName": "Chicken Nuggets",
        "price": 5
    },
    {
        "productId": "563454",
        "fullName": "Pizza Quattro Stagioni",
        "price": 7
    }
]
```

Attenzione: non c'è la virgola dopo il price e dopo l'ultimo prodotto.

Se `n` è 0, la funzione scrive:

```
[
]
```