Esercizio 1 (5 Punti)

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

```
extern bool is_ugly(unsigned int num);
```

La funzione deve restituire `true` se il numero `num` è "brutto", `false` altrimenti.

Un numero è detto "brutto" (*ugly number* in inglese) se i suoi **soli** fattori primi sono 2, 3 o 5. Ad esempio 14 non è brutto, perché $14 = 2 \times 7$, mentre invece 30 lo è, dato che $30 = 2 \times 3 \times 5$. $24 = 2^3 \times 3$ è brutto, $25 = 5^2$ è brutto, mentre $26 = 2 \times 13$ non è brutto.

Esercizio 2 (6 Punti)

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

```
extern bool is_pangram(const char* sentence);
```

La funzione deve restituire `true` se la frase passata come parametro è un pangramma, `false` altrimenti.

Un pangramma (o anche pantogramma, dal greco πᾶν γράμμα, pan gramma, "tutte le lettere") è una frase in cui vengono utilizzate tutte le lettere dell'alfabeto.

Per questo esercizio si considerino solo le lettere dell'alfabeto italiano, ovvero una frase è un pangramma anche se non contiene j, k, w, x, y. La variabile `sentence` punta ad una frase in cui vengono utilizzati solo valori corrispondenti a caratteri ASCII (da 0 a 127).

Se `sentence` è `NULL` o punta ad una stringa vuota, la funzione deve restituire `false`.

Esercizio 3 (7 punti)

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

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

e la funzione:

```
extern struct matrix *mat_submatrix(const struct matrix *mat,
    const int *row_idxs, const int *col_idxs);
```

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 accetta come parametro un puntatore ad una matrice `mat`, e deve restituire un puntatore a una nuova matrice allocata dinamicamente. La nuova matrice è una sottomatrice di `mat` ottenuta copiando i dati delle righe e delle colonne indicati nei vettori puntati da `row_idxs` e `col_idxs`. I due vettori sono composti da un numero arbitrario di elementi (inferiore al numero di righe o colonne di `mat`) seguiti da un valore negativo. Se `mat` è `NULL`, o se i vettori di indici sono `NULL` o contengono valori non inferiori al numero di righe e colonne rispettivamente, la funzione restituisce `NULL`.

Ad esempio, data la matrice:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

Chiamando la funzione `mat_submatrix()` con `row_idxs = {1, 2, -1}` e `col_idxs = {0, 2, 3, -1}` restituisce la nuova matrice:

$$A = \begin{pmatrix} 5 & 7 & 8 \\ 9 & 11 & 12 \end{pmatrix}$$

Esercizio 4 (7 Punti)

Creare il file `utf8.h` e `utf8.c` che consentano di utilizzare la seguente funzione:

```
extern size_t utf8_encode(uint32_t codepoint, uint8_t seq[4]);
```

La funzione riceve un `codepoint` Unicode (un valore a 32 bit compreso tra 0 e 10FFFF) e lo converte in una sequenza da 1 a 4 byte secondo lo standard UTF-8, inserendo ogni byte nello spazio puntato da `seq`. La funzione ritorna il numero di byte prodotti in output, o `0` se il codice è maggiore di 10FFFF.

La conversione avviene secondo la tabella seguente:

Numero di byte	Primo codepoint	Ultimo codepoint	Byte 1	Byte 2	Byte 3	Byte 4
1	U+0000	U+007F	0xxxxxxx			
2	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	

Numero di byte	Primo codepoint	Ultimo codepoint	Byte 1	Byte 2	Byte 3	Byte 4
4	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

I caratteri indicati con x vengono sostituiti dai bit del codice, inserendo nel primo byte i bit dal più significativo al meno significativo, continuando poi nei byte successivi, sempre dal più significativo al meno significativo.

Consideriamo la codifica del segno dell'Euro, €:

- Il codice Unicode per "€" è U+20AC.
- Siccome questo codice si trova tra U+0800 e U+FFFF, serviranno tre byte per la codifica.
- Il valore esadecimale 20AC è 0010 0000 1010 1100 in binario. I due bit a zero a sinistra vengono aggiunti perché una codifica a tre byte ha bisogno di 16 bit del codice.
- Dalla tabella vediamo che un codice a tre byte comincia con 1110...
- I quattro bit più significativi del codice vengono inseriti al posto delle quattro x del primo byte (1110 0010), lasciando 12 bits da codificare (...0000 1010 1100).
- Tutti i bit di continuazione possono contenere esattamente 6 bit. Quindi i sei bit successivi vengono messi nel secondo byte e 10 viene memorizzato nei due bit più significativi (ovvero 1000 0010).
- Infine gli ultimi 6 bit del codice vengono messi nel terzo byte e di nuovo 10 nei due bit più significativi (1010 1100).

I tre byte così ottenuti 1110 0010 1000 0010 1010 1100 possono essere scritti in esadecimale come E2 82 AC.

Chiamando quindi la funzione `utf8_encode()` con `codepoint` pari a 20AC, troveremo E2 in `seq[0]`, 82 in `seq[1]`, AC in `seq[2]` e la funzione ritorna 3. `seq[4]` non viene utilizzato dalla funzione.

Esercizio 5 (8 Punti)

Un formato binario compatto per memorizzare valori reali nell'intervallo `[-2, 2)` con precisione circa alla quarta cifra decimale è ottenuto prendendo numeri **a 16 bit** in complemento a 2 e **dividendoli per** 2^{14}.

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

```
struct dvec {
    size_t n;
    double *d;
};
```

e la funzione:

```
struct dvec *read_dvec_comp(const char *filename);
```

La funzione apre il file `filename` in modalità non tradotta e legge dal file binario numeri a 16 bit in complemento a 2 in little endian, producendo in output una `struct dvec` allocata

dinamicamente con `n` pari al numero di valori a 16 bit nel file, e `d` un puntatore ad un'area di memoria contenente gli `n` double corrispondenti ai valori codificati nel file.

I valori in `double` si ottengono prendendo i numeri a 16 bit con segno in complemento a 2 e dividendoli per 2^{14}.

Se non è possibile aprire il file, o il file non contiene alcun valore, la funzione crea una `struct dvec` con `n=0` e `d=NULL`. Ad esempio il file: `00 80 CD AB FF FF 00 00 01 00 32 54 FF 7F` contiene i valori (rappresentati in base 10 con 6 cifre decimali): `-2.000000, -1.315613, -0.000061, 0.000000, 0.000061, 1.315552, 1.999939`

Come è possibile vedere dal'esempio, non c'è alcun header, né alcuna informazione ulteriore nel file. Solo valori a 16 bit.