# Esame MDP del 19/02/2021

## Esercizio 1

Una software house sta realizzando una app per dispositivi mobili che deve trasmettere dati in formato JSON. Uno dei campi richiesti è però una piccola immagine a colori RGB e JSON non supporta la trasmissione di dati binari. Per questo motivo la software house decide di utilizzare un semplice encoding Base64. L'immagine viene compressa a piani separati con l'algoritmo PackBits e poi ogni piano così compresso viene codificato in Base64. Le informazioni vengono passate come campi di un dizionario JSON. Ad esempio:

```
{
   "width": 6,
   "height": 6,
   "red": "3f+A",
   "green": "/gD+//4A/v/+AP7//gD+//4A/v/+AP7/gA==",
   "blue": "3QCA"
}
```

In questo esempio è codificata un'immagine con tre colonne rosse (255,0,0) e tre gialle (255,255,0), come in figura:

| r | r | r | g | g | g |
|---|---|---|---|---|---|
| r | r | r | g | g | g |
| r | r | r | g | g | g |
| r | r | r | g | g | g |
| r | r | r | g | g | g |
| r | r | r | g | g | g |

quindi i piani colore contengono (i valori sono mostrati qui in esadecimale e c'è uno spazio più grande tra una riga e l'altra, solo per chiarezza):

```
R:  FF,FF,FF,FF,FF,FF,  FF,FF,FF,FF,FF,FF,  FF,FF,FF,FF,FF,FF,
    FF,FF,FF,FF,FF,FF,  FF,FF,FF,FF,FF,FF,  FF,FF,FF,FF,FF,FF

G:  00,00,00,FF,FF,FF,  00,00,00,FF,FF,FF,  00,00,00,FF,FF,FF,
    00,00,00,FF,FF,FF,  00,00,00,FF,FF,FF,  00,00,00,FF,FF,FF

B:  00,00,00,00,00,00,  00,00,00,00,00,00,  00,00,00,00,00,00,
    00,00,00,00,00,00,  00,00,00,00,00,00,  00,00,00,00,00,00
```

Dopo la codifica PackBits:

```
R:  DD,FF,80
G:  FE,00,FE,FF,FE,00,FE,FF,FE,00,FE,FF,FE,00,FE,FF,FE,00,FE,FF,FE,00,FE,FF,80
```

```
B:  DD,00,80
```

Dopo l'encoding in Base64:

```
R:  3f+A
G:  /gD+//4A/v/+AP7//gD+//4A/v/+AP7/gA==
B:  3QCA
```

Nel file `process_ppm.cpp` implementare la funzione corrispondente alla seguente dichiarazione:

```
bool LoadPPM(const std::string& filename, mat<vec3b>& img);
```

La funzione deve caricare un'immagine a colori nel formato PPM, seguendo le specifiche allegate. Se la lettura va a buon fine la funzione ritorna `true`, altrimenti ritorna `false`. L'implementazione della classe `mat` e della classe `vec3b` sono disponibili nei file `mat.h`, `ppm.h` e `ppm.cpp`.

Sono dati a titolo di esempio i file `test.ppm` e `facolta.ppm`.

## Esercizio 2

Sempre nel file `process_ppm.cpp` implementare la procedura corrispondente alla seguente dichiarazione:

```
void SplitRGB(const mat<vec3b>& img, mat<uint8_t>& img_r, mat<uint8_t>& img_g,
```

La procedura deve separare la matrice di valori RGB `img` in tre matrici (`img_r`, `img_g`, e `img_b`) ognuna contenente un solo piano colore, R, G, e B.

## Esercizio 3

Nel file `compress.cpp` implementare la procedura corrispondente alla seguente dichiarazione:

```
void PackBitsEncode(const mat<uint8_t>& img, std::vector<uint8_t>& encoded);
```

che applica l'algoritmo PackBits ad un piano colore, `img`, salvando il risultato in un vettore di `uint8_t`.

Come visto a lezione, PackBits è uno schema di compressione veloce e semplice senza perdita di dati per la codifica dei dati in run-length.

Il flusso di dati PackBits è costituito da pacchetti con un'intestazione da un byte seguita da uno o più byte di dati compressi.

Nella tabella seguente è riportata l'associazione tra il valore del byte di intestazione, `L`, e i byte dei dati.

| header (L) | byte di dati che seguono l'header |
|---|---|
| da 0 a 127 | (L+1) byte letterali di dati (copia) |
| da 129 a 255 | un byte di dati, ripetuto (257-L) volte nell'output decompresso (run) |
| 128 | End of data marker (EOD) |

L'algoritmo deve scorrere l'input e per ogni sequenza di byte uguali emettere una run, altrimenti una copia per le sequenze prive di ripetizioni e al termine dei dati un EOD.

## Esercizio 4

Sempre nel file `compress.cpp` implementare la funzione corrispondente alla seguente dichiarazione:

```
std::string Base64Encode(const std::vector<uint8_t>& v);
```

che applica ad una sequenza `v` l'encoding Base64 e ritorna la stringa risultante.

Si ricorda che il Base64 è un sistema di codifica che consente la traduzione di dati binari in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi. L'algoritmo che effettua la conversione suddivide l'input in gruppi di tre byte da cui si ottengono quattro sottogruppi da 6 bit, i quali possono quindi contenere valori da 0 a 63. Ogni possibile valore viene convertito in un carattere ASCII secondo la seguente tabella:

| Valore | ASCII | Valore | ASCII | Valore | ASCII | Valore | ASCII |
|---|---|---|---|---|---|---|---|
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |

| Valore | ASCII | Valore | ASCII | Valore | ASCII | Valore | ASCII |
|---|---|---|---|---|---|---|---|
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |

Invece che implementare il padding standard del Base64, è sufficiente aggiungere byte uguali a 128 in fondo al vettore, in modo che la lunghezza sia un multiplo di 3 e quindi si ottengano sempre gruppi completi di 3 byte.

## Esercizio 5

Sempre nel file `compress.cpp` implementare la funzione corrispondente alla seguente dichiarazione:

```
std::string JSON(const std::string& filename);
```

che data un'immagine RGB in formato PPM costruisce e ritorna la stringa in formato JSON come richiesto dalla software house.