COMP482- Programming Studio

Project 1- Image Compression Using Huffman Coding

Prof.Dr. Muhittin Gökmen

Project 1: Image Compression using Huffman coding

- In this project, you will develop a program in Python to compress an image by using Huffman method, save the compressed file, and decompress the image from the compressed file.
- Input to your program is an image file in *.png or *.bmp format
- You will implement methods by yourself without using any image processing libraries other than reading, writing and showing images.
- You will prepare a report to show your implementation and results.

Image Compression using Huffman Coding

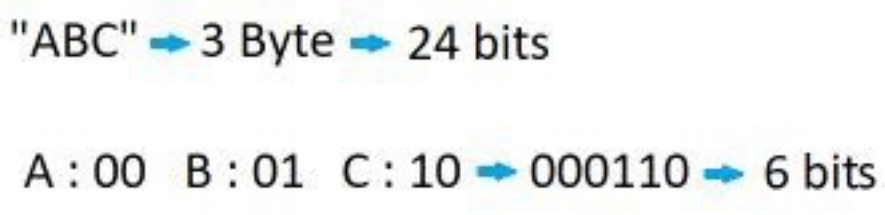
- The project will consist of 5 Levels:
  - Level 1: Huffman Encoding and Decoding
  - Level 2: Image Compression (Gray Level)
  - Level 3: Image Compression (Gray Level differences)
  - Level 4: Image Compression (Color)
  - Level 5: Image Compression (Color differences)
  - Level 6: GUI

Level 1: Huffman Encoding and Decoding

From Yağmur Çiğdem Aktaş's blog

Huffman Encoding

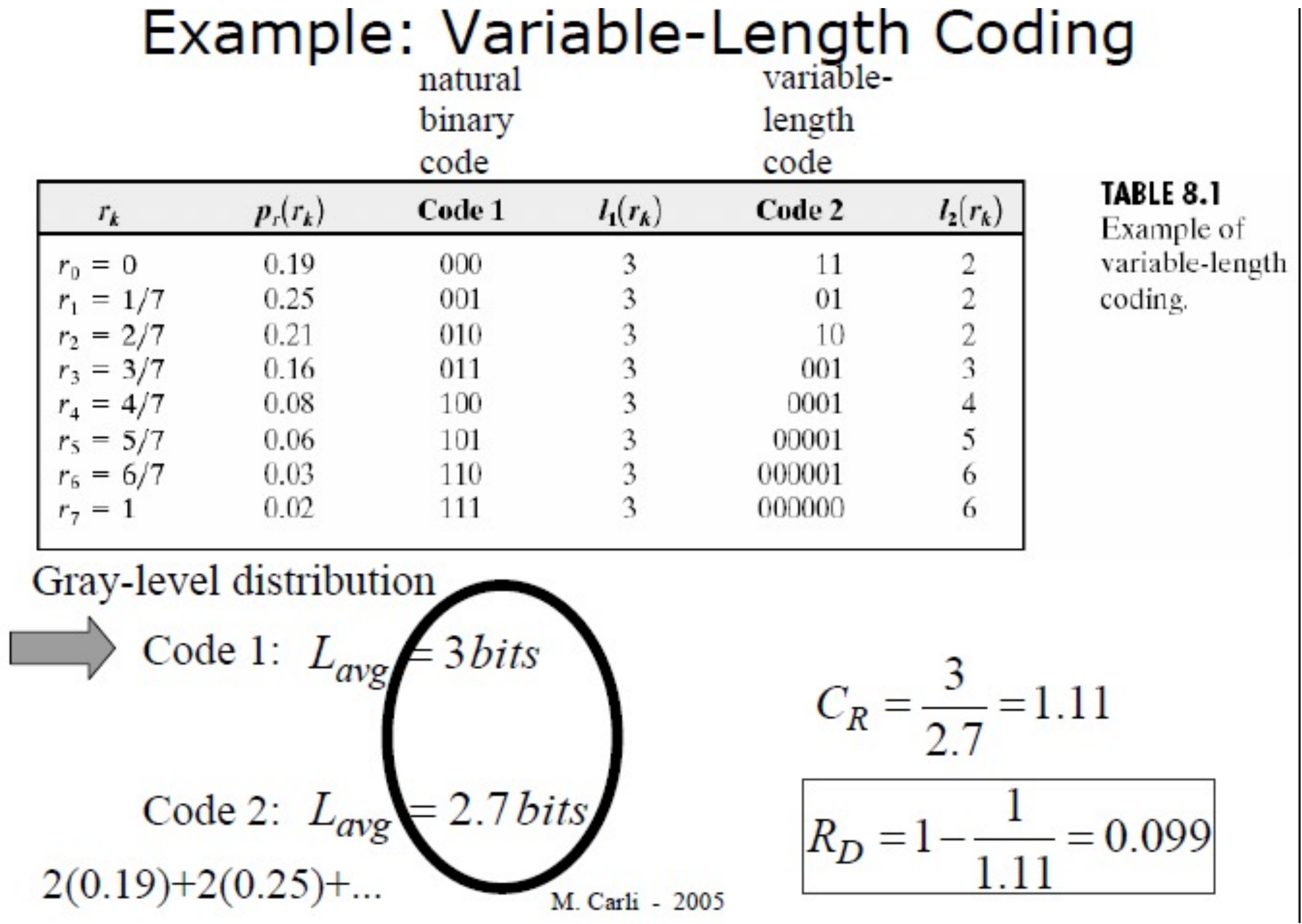
- An old but efficient compression technique.
- Huffman Encoding is a Lossless Compression Algorithm used to compress the data. It is an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper “A Method for the Construction of Minimum-Redundancy Codes”. [1]
- As it can be understood from being a “Compression Technique”, the aim is to encode the same data in a way that takes up less space. Accordingly, when a data is encoded with Huffman Coding, we get a unique code for each symbol in the data.
- For example, the string “ABC” occupies 3 bytes without any compression. Let’s assume while the character A is given the code 00, the character B is given the code 01, the character C is given the code 10 as the result of encoding.
- To store the same data, we would only need to use 6 bits instead of 3 bytes. Before examining the working principle of Huffman Encoding, I hope what I mean by compression is better understood !



"ABC" → 3 Byte → 24 bits

A : 00 B : 01 C : 10 → 000110 → 6 bits

Huffman Coding is a variable length coding



Example: Variable-Length Coding

r_k	$p_r(r_k)$	**Code 1** (natural binary code)	$l_1(r_k)$	**Code 2** (variable-length code)	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

TABLE 8.1 Example of variable-length coding.

Gray-level distribution

Code 1: $L_{avg} = 3\,bits$

Code 2: $L_{avg} = 2.7\,bits$

$2(0.19)+2(0.25)+\ldots$

$$C_R = \frac{3}{2.7} = 1.11$$

$$\boxed{R_D = 1 - \frac{1}{1.11} = 0.099}$$

M. Carli - 2005

Coding

- The average lenght of the code words assigned to the various gray-level values: the sum of the product of the number of bits used to represent each gray level and the probability that the gray level occurs.

- Total number of bits to code an MxN image is MNL_{avg}

- Usually $l(r_k) = m$ bits (constant). $\Rightarrow L_{avg} = \sum_k \; m p_r(r_k) = m$

Coding

- It makes sense to assign fewer bits to those r_k for which $p_r(r_k)$ are large in order to reduce the sum.
- $\Rightarrow$ achieves data compression and results in a <u>variable length code</u>.
- More probable gray levels will have fewer number of bits.

$$\boxed{L_{avg} = \sum_{k=0}^{L-1} l(r_k)\, p_r(r_k)}$$

Example: Variable-Length Coding

r_k	$p_r(r_k)$	**Code 1** (natural binary code)	$l_1(r_k)$	**Code 2** (variable-length code)	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

TABLE 8.1 Example of variable-length coding.

Gray-level distribution

Code 1: $L_{avg} = 3\,bits$

Code 2: $L_{avg} = 2.7\,bits$

2(0.19)+2(0.25)+...

$$C_R = \frac{3}{2.7} = 1.11$$

$$\boxed{R_D = 1 - \frac{1}{1.11} = 0.099}$$

M. Carli - 2005

Huffman Encoding

The Algorithm

- Huffman Encoding is an algorithm which uses ***frequency*** (or ***probability***) feature of symbols and a binary tree structure. Basic idea is to allocate short codes to symbols with high probabilities and long codes to symbols with low probabilities.
- It consists of the following 3 steps:
  - Probability Calculation & Ordering the Symbols
  - Binary Tree Transformation
  - Assigning Codes to the Symbols

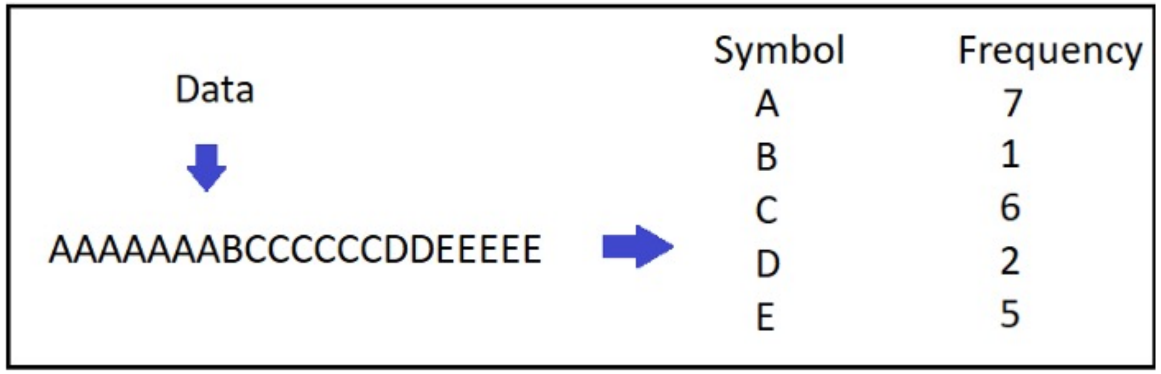
Huffman Encoding

Probability Calculation & Ordering the Symbols

- We count the number of each symbol in the whole data, then we calculate the “probability” of each symbol by dividing that count by the total number of characters in the data.
- Since it’s an algorithm using probability, more common symbols — the symbols having higher probability — are generally represented using fewer bits than less common symbols. **This is one of the advantageous sides** of Huffman Encoding.

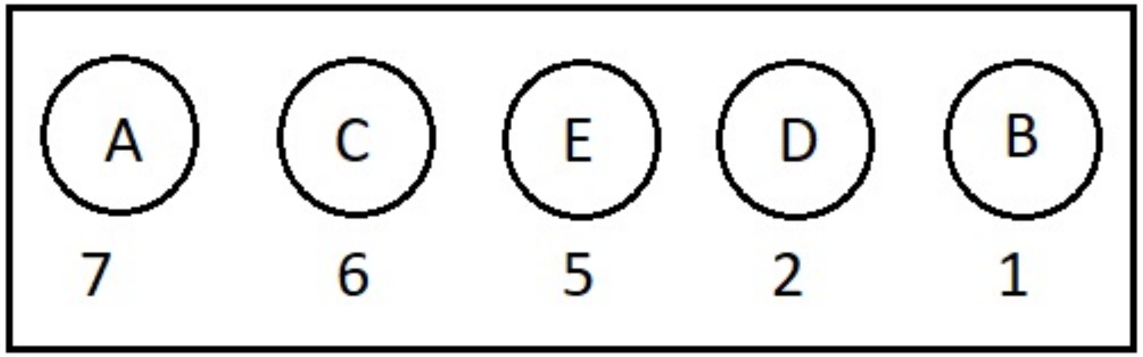
Huffman Encoding

As an example, for the following data having 5 different symbols as A B C D E, we have the probabilities as shown right:



Symbol	Frequency
A	7
B	1
C	6
D	2
E	5

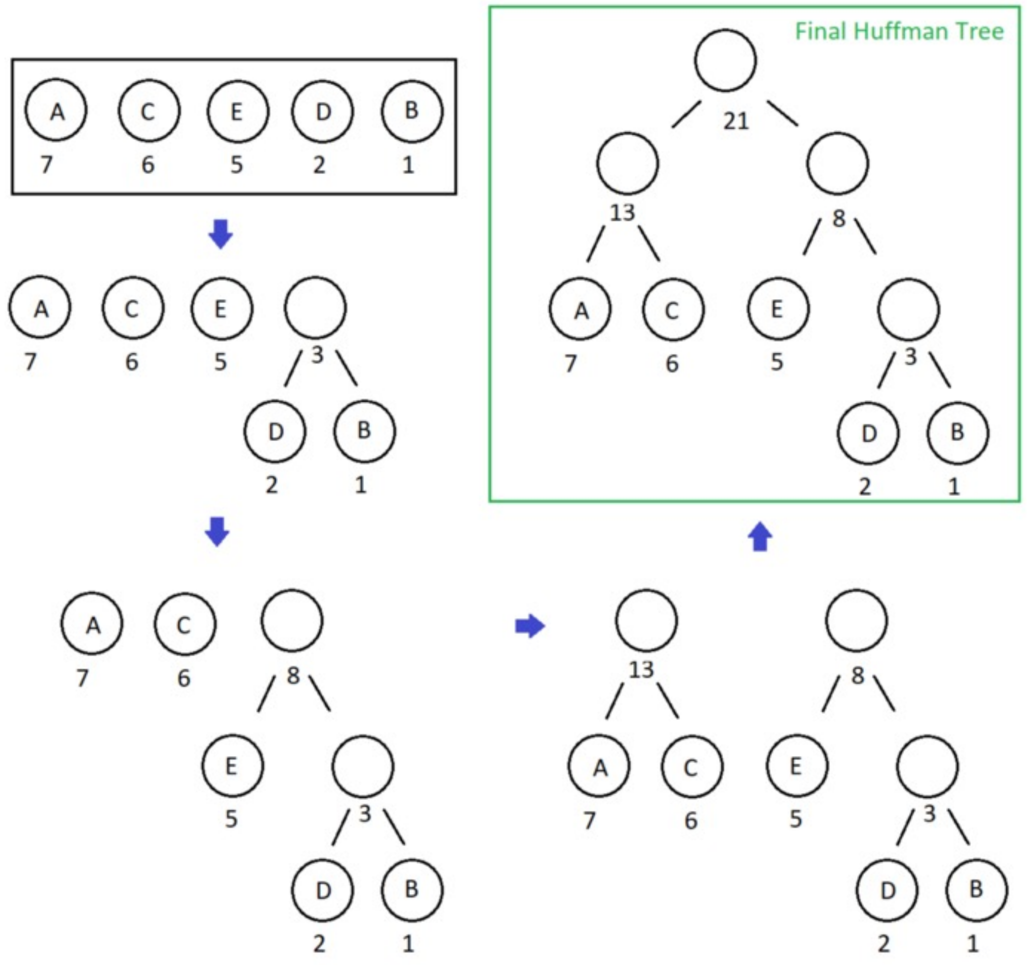
Then we easily order the symbols according to their probabilities representing each symbol as a node and call that our “collection”. Now, we are ready to pass the next step.



Huffman Encoding

Binary Tree Transformation

- From the collection, we **pick out the two nodes with the smallest sum** of probabilities and combine them into a new tree whose root has the probability equal to that sum.
- We add the new tree back into the collection.
- We **repeat** this process until one tree encompassing all the input probabilities has been constructed.

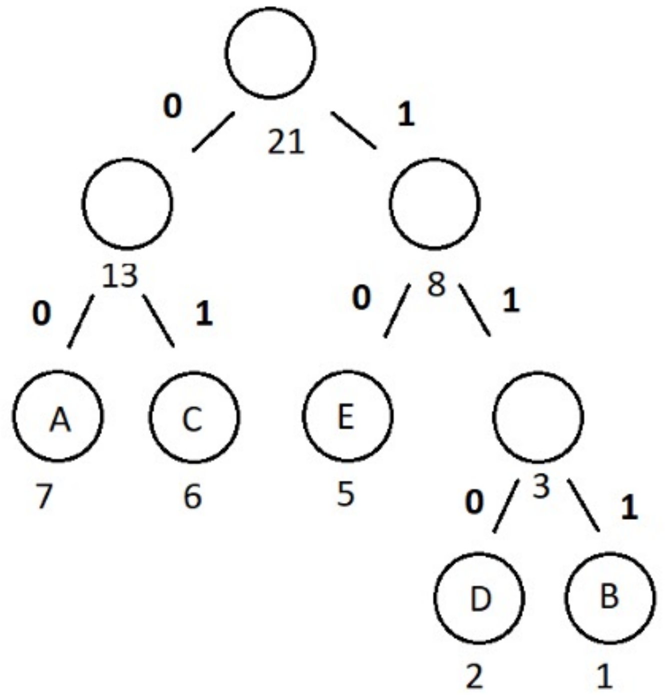


Huffman Encoding

Assigning Codes to Symbols

- After obtaining this binary tree — so called ***Huffman Tree***, the only thing we should do is to assign **1** for each time we go **right** child and **0** for each time we go **left** child.
- Finally, we have the symbols and their codes obtaining by Huffman Encoding!

Symbol	Bit Code
A	00
B	111
C	01
D	110
E	10



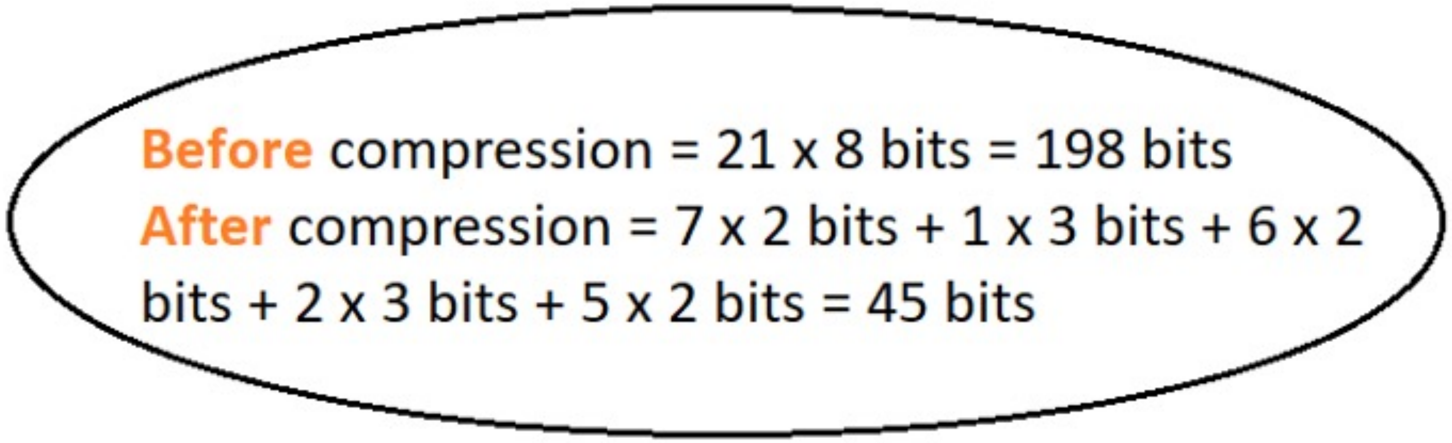
Huffman Encoding

- We can take a quick look and see even only for 21 characters, the difference between compressed and non-compressed data is nontrivial.

AAAAAAABCCCCCCDDEEEEE : 21 x 8 bits = 198

00000000000000111010101010111011010101010 : 45 bits

Compression Ratio (Cr) : 198 / 45 = 4.4



Before compression = 21 x 8 bits = 198 bits

After compression = 7 x 2 bits + 1 x 3 bits + 6 x 2 bits + 2 x 3 bits + 5 x 2 bits = 45 bits

Huffman Encoding

Implementation with Python

- To implement Huffman Encoding, we start with a Node class, which refers to the nodes of Binary Huffman Tree.
- In that essence, each node has a symbol and related probability variable, a left and right child and code variable.
- Code variable will be 0 or 1 when we travel through the Huffman Tree according to the side we pick (left 0, right 1)

AAAAAAABCCCCCCDDEEEEE

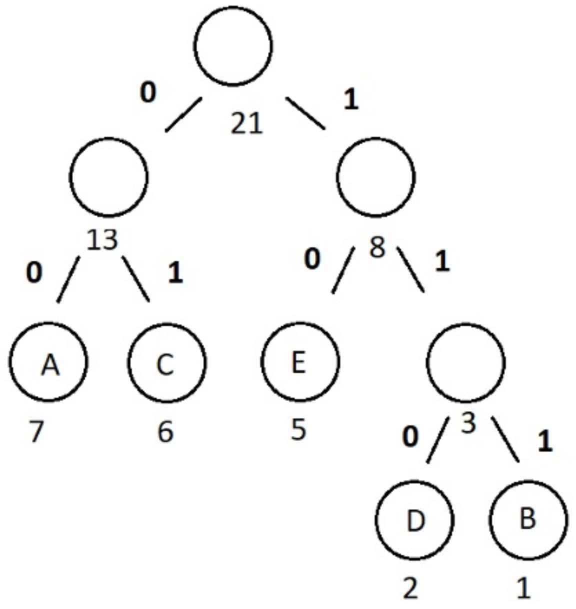
Huffman Decoding

- **Huffman Decoding**
- Decompress your Huffman-encoded data to obtain your initial data
- We already saw how to encode a given data using Huffman Encoding. Now we will examine how to decode a Huffman Encoded data to obtain the initial, uncompressed data again.
- Having our Binary Huffman Tree obtained during encode phase, decoding is a very simple process to perform.

Huffman Decoding

- Let’s consider we have the same example with Huffman Encoding post.
- In decoding, we will restore the original symbols from the compressed data

Symbol	Bit Code
A	00
B	111
C	01
D	110
E	10



- 1110011100 ?
- Start from the root node
    - If code is 1, move to right child
    - If code is 0, move to left child
    - If the node is a leaf (no children) , peak the symbol, go to the root of the tree.

Level 1 actions:

Compression

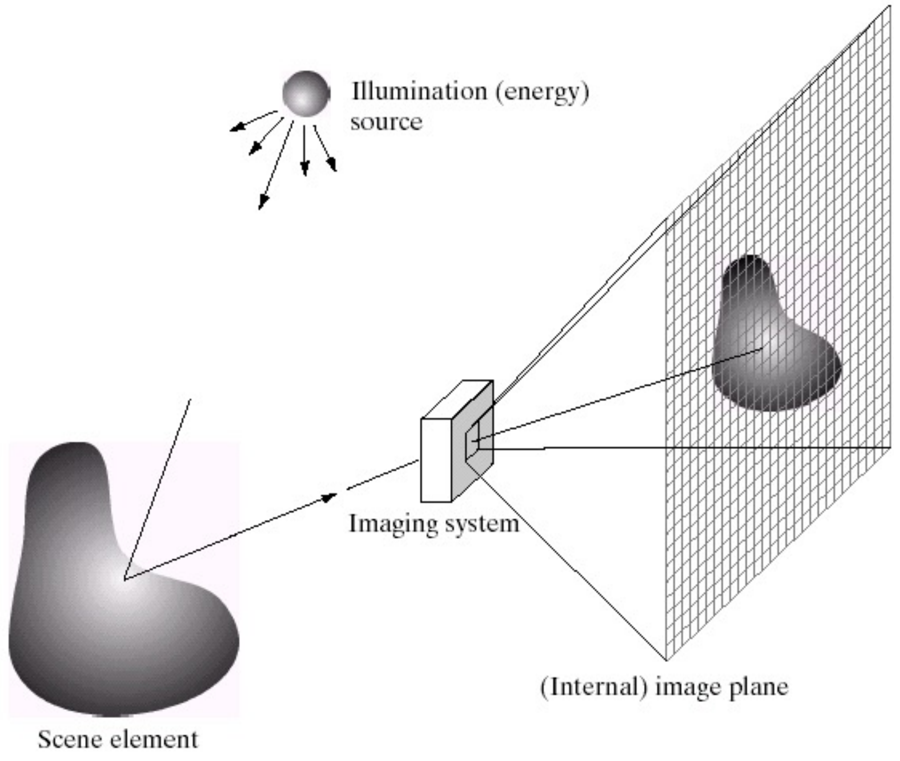
- Read the characters from text file
- Calculate the probality of each symbol
- Construct the Huffman tree and code
- Encode the symbols in the input file
- Save the compressed file
- Calculate entropy, average code length and compression ratio

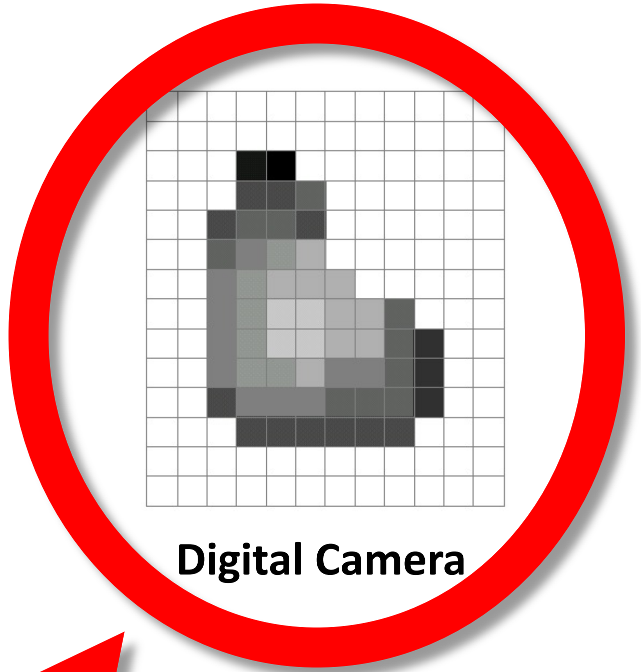
Decompression

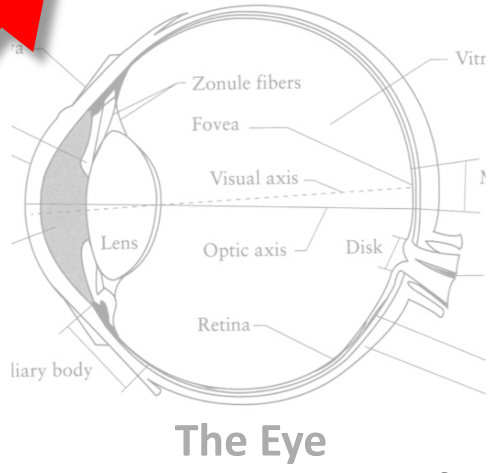
- Read the stored data
- Restore the Huffman tree
- Restore the symbols from the compressed data
- Save the restored text
- Compare the original and restored text

Level 2:Image Compression

What is an image?







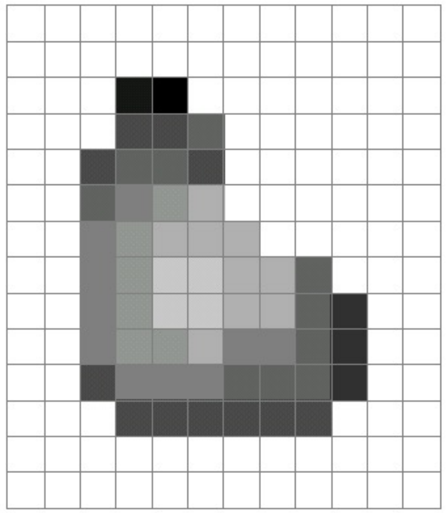
We' ll focus on these in this class

(More on this process later)

Source: A. Efros

What is an image?

- A grid (matrix) of intensity values



=

255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	20	0	255	255	255	255	255	255	255
255	255	255	75	75	75	255	255	255	255	255	255
255	255	75	95	95	75	255	255	255	255	255	255
255	255	96	127	145	175	255	255	255	255	255	255
255	255	127	145	175	175	175	255	255	255	255	255
255	255	127	145	200	200	175	175	95	255	255	255
255	255	127	145	200	200	175	175	95	47	255	255
255	255	127	145	145	175	127	127	95	47	255	255
255	255	74	127	127	127	95	95	95	47	255	255
255	255	255	74	74	74	74	74	74	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255

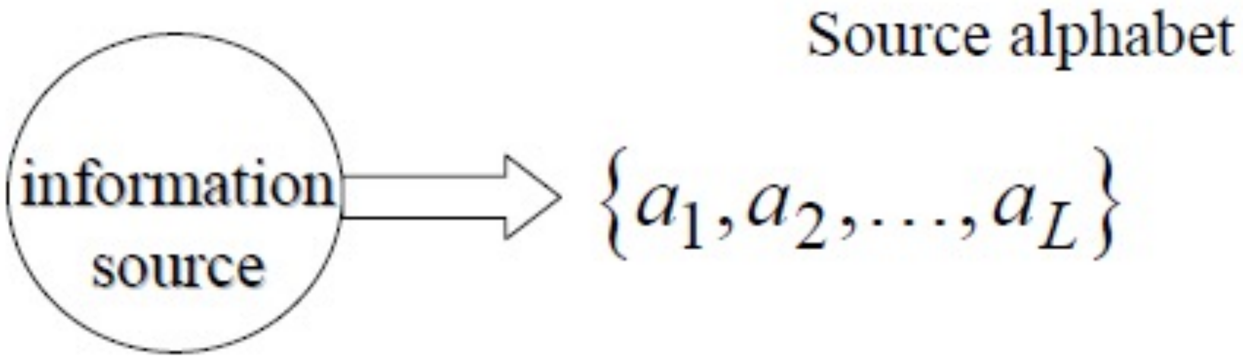
(common to use one byte per value: 0 = black, 255 = white)

Lossless vs. Lossy Compression

Compression techniques:
- lossless
- Lossy

- Lossless (or information preserving) compression: Images can be compressed and restored without any loss of information. The original image and restored image are bit by bit identical.(.e.g. medical imaging, satellite imaging). Huffman coding is lossless compression technique.
- Lossy (or information reducing) compression: perfect recovery not possible. Larger data compression (e.g.,TV signals, teleconferencing)

Entropy



• Let $p(a_l), l = 1, 2, \ldots, L$ be the probability of each symbol

• Then the entropy (or uncertainty) of the source is given by

$$H = -\sum_{l=1}^{L} p(a_l) \log(p(a_l)) \quad \text{bits/symbol}$$

Computing the Entropy of an Image

8-bit gray level source- statistically independent pixels emission

- Consider the 8-bit image:

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

Gray Level	Counts	Probability
21	12	3/8
95	4	1/8
169	4	1/8
243	12	3/8

$$H = 1.81 \text{ bits/pixel}$$

(first-order estimate)

Huffman (Variable-Length) Coding

- Devised by Huffman in 1952 for removing coding redundancy

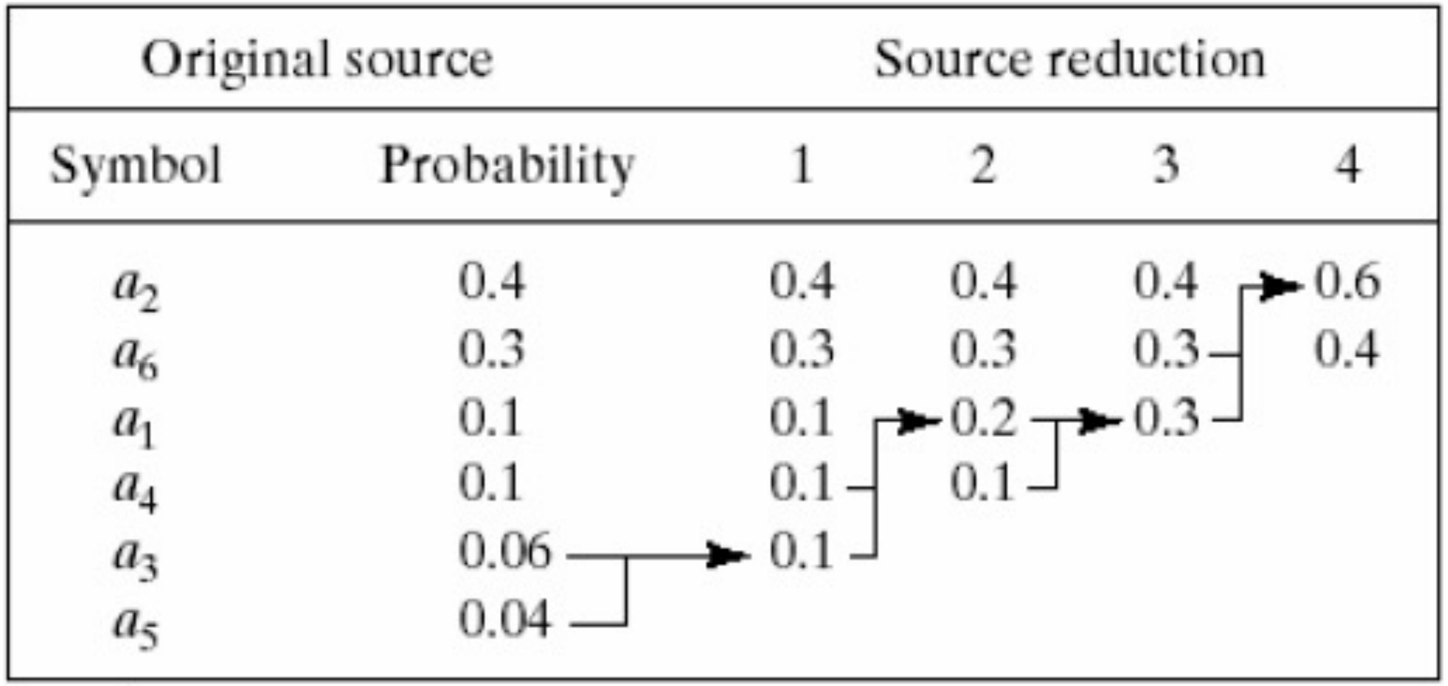
- Property: If the symbols of an information source are coded individually, the Huffman coding yields the smallest possible number of code symbols per source symbol

Huffman Coding

- Method: create a series of source reductions by ordering the probabilities of the symbols under consideration and combining the lowest probability symbols into a single symbol that replaces them in the next source reduction

- Rationale: To assign the shortest possible codewords to the most probable symbols

Example: Huffman Source Reductions

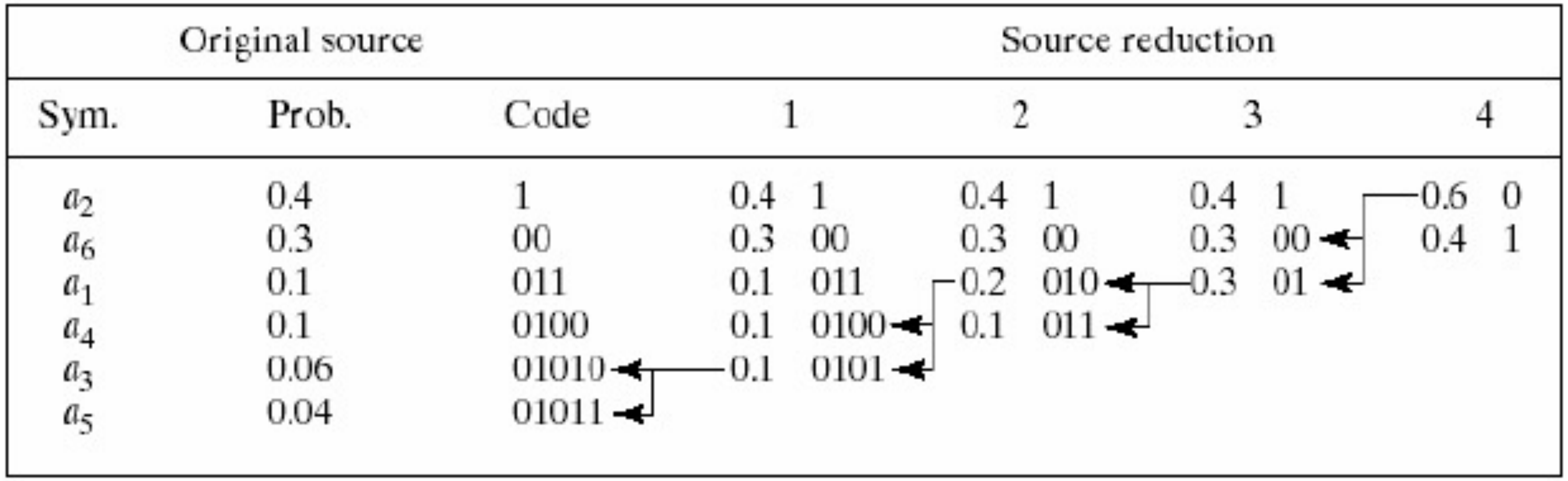


Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

symbols are ordered according to
decreasing probability

M. Carli - 2005

Example: Huffman Code Assignment Procedure



Original source			Source reduction							
Sym.	Prob.	Code	1		2		3		4	
a_2	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
a_6	0.3	00	0.3	00	0.3	00	0.3	00	0.4	1
a_1	0.1	011	0.1	011	0.2	010	0.3	01		
a_4	0.1	0100	0.1	0100	0.1	011				
a_3	0.06	01010	0.1	0101						
a_5	0.04	01011								

$$L_{avg} = 0.4\cdot 1 + 0.3\cdot 2 + 0.1\cdot 3 + 0.1\cdot 4 + 0.06\cdot 5 + 0.04\cdot 5 = 2.2 \text{ bits/symbol}$$

$$H = -\sum p\log_2(p) = 2.14 \text{ bits/symbol}$$

- Coding/decoding is accomplished with a lookup table
- It is a block code: each source symbol is mapped into a fixed sequence of code symbols
- It is instantaneous: each code word in a string of code symbols can be decoded without looking at succeeding symbols
- It is uniquely decodable: any string of code symbols can be decoded only in one way;

example:

$$\underbrace{01010}_{a_3}\underbrace{011}_{a_1}\underbrace{1}_{a_2}\underbrace{1}_{a_2}\underbrace{00}_{a_6}$$

Level 2 actions:

Compression

- Read the image file
- Calculate the probality of each gray level
- Construct the Huffman tree and code
- Scan the image and store the binary code instead of gray levels
- Save the compressed file
- Calculate entropy, average code length and compression ratio

Decompression

- Read the stored data
- Restore the Huffman tree
- Restore the image from the compressed data
- Save the restored image
- Compare the original image and restored image

Level 3:Image Compression (Gray level differences)

Difference image

- Since the entropy of the source is the lower limit of the average code length, we would like to reduce the entropy of the image by taking the differences between successive pixels.
- Keep the first column and replace following with the aritmetic difference between adjacent columns. And then keep the first row, and replace following pixels in the first column with the arithmetic difference between adjacent pixel in the first column.

Difference image

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

Original Image

```
arr[i][j]
N=5
M=7
```

21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0

Intermediate Difference Image:

```
for i in range(N):
    for j in range(1,M):
            darr[i][j] = arr[i][j] - arr[i][j-1]
```

21							
0	0	0	74	74	74	0	0
0	0	0	74	74	74	0	0
0	0	0	74	74	74	0	0
0	0	0	74	74	74	0	0

Difference Image:

```
Diff_arr = darr
pivot = darr[0[0]
diff_arr[0][0] = darr[0][0] – pivot
for i in range(1, N):
    diff_arr[i][0] = darr[i][0] - darr[i-1][0]
```

Gray Level	Counts	Probability
21	12	3/8
95	4	1/8
169	4	1/8
243	12	3/8
	H = 1.65 bits/pixel	

Gray Level	Counts	Probability
0	16	1/2
21	4	1/8
74	12	3/8
	H = 1.40 bits/pixel	

Gray Level	Counts	Probability
0	20	1/2
74	12	3/8
	H = 1.03 bits/pixel	

Level 3 actions:

Compression

- Read the image file
- Obtain the difference image by taking the row-wise differences between successive gray levels in each row, starting from the second pixel in a row.
- Take the column-wise differences between successive pixels in the first column, starting from second pixel.
- Calculate the probality of each gray level
- Construct the Huffman tree and code from the difference image
- Scan the difference image and store the binary code instead of gray levels
- Save the compressed file
- Calculate entropy, average code length and compression ratio

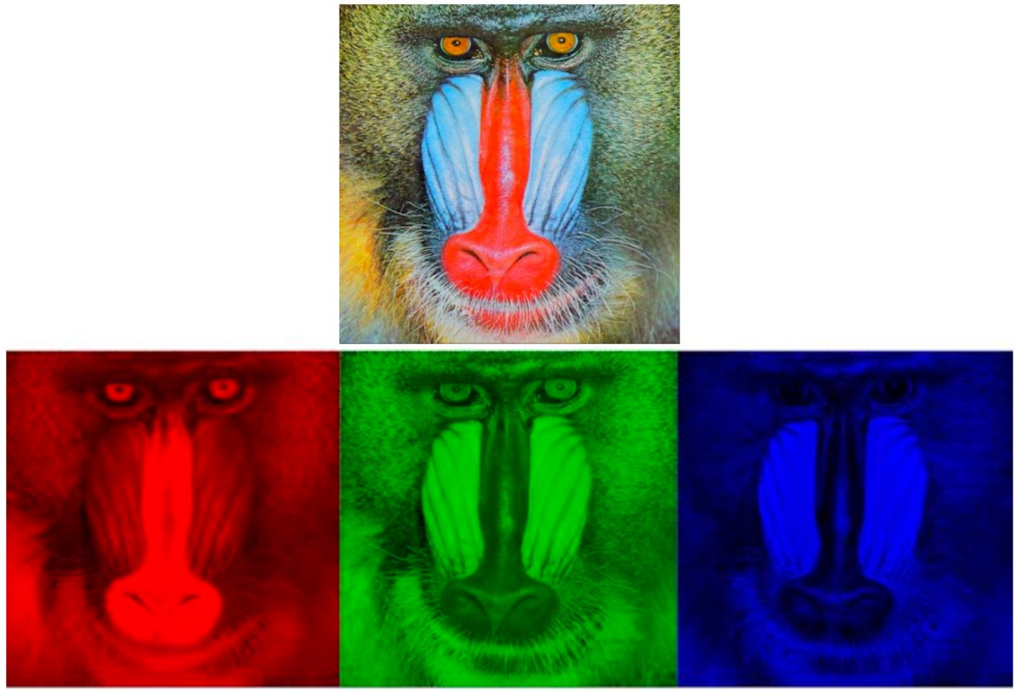
Decompression

- Read the stored data
- Restore the Huffman tree
- Restore the difference image from the compressed data
- Restore the original image from differences
- Save the restored image
- Compare the original image and restored image

Level 4: Color Image Compression

Color Images

- Image has three channels (bands)
- Each channel spans a-bit values.



- Level 4: Apply Huffman coding to each color components. (repeat level 2 for each color component)
- Level 5: Apply Huffman coding to the gray level differences at each components. (repeat level 3 for each color component)

Level 4 actions:

Compression

- Read the image file, decompose into color components (you will generate three gray level image from a color image by separating RGB components)
- Calculate the probality of each gray level for each component
- Construct the Huffman tree and code for each component
- Scan the image and store the binary code instead of gray levels
- Save the compressed file
- Calculate entropy, average code length and compression ratio

Decompression

- Read the stored data
- Restore the Huffman tree
- Restore the image from the compressed data
- Save the restored image
- Compare the original image and restored image

Level 5 actions:

Compression

- Read the color image file, decompose into color compoents (you will generate three gray level image from a color image by separating R,GiB components)
- Obtain the difference image by taking the row-wise differences between successive gray levels in each row, starting from the second pixel in a row.
- Take the column-wise differences between successive pixels in the first column, starting from second pixel.
- Calculate the probality of each gray level
- Construct the Huffman tree and code from the difference image
- Scan the difference image and store the binary code instead of gray levels
- Save the compressed file
- Calculate entropy, average code length and compression ratio

Decompression

- Read the stored data
- Restore the Huffman tree
- Restore the difference image from the compressed data
- Restore the original image from differences for each color component
- Save the restored color image by combining RGB images.
- Compare the original image and restored color image

GUI

Design and Develop a GUI to achieve the following:

- The user will be able to
  - Select a file from a directory and load it
  - Select image of text file
  - Select a method (gray levels or differences)
  - Save the compressed image into a file
  - Reads a compressed file, show the decompressed text/image file
  - Shows average code length and compression ratio, size of compresses file.

Deliverables

- At the end of the project
- You should prepare a project report with
    - Title page
    - Abstract
    - Description of the project
    - Description of your solution
        - Use UML diagrams to demonstrate your program
    - Examples of test files, input, outputs showing that your program works correctly
    - List of achievements (what did you learn during the project)
    - References
- Prepare a presentation with power point and present it in the class (Do not directly show your code, but start with Usecases, class diagrams etc.)
- Submit your report, presentation and code to the blackboard

Recommended Calendar for the project 1

- February 15, assignment of the project, design and implement Huffman Encoding & Decoding
- February 22, design and implement Gray Level Image compression (Level 2 and 3)
- March 1, design and implement Color image compression and GUI (Level 4,5 and 6)
- March 8, project demonstration and submission of the report, code and presentation