Experiment No.1

Name: Huzaib Mulla
Roll No: 18ET15
Date: 08/08/2020

Aim: To study and implement
Huffman coding

Equipment/Software:

Python and Huffman packages

Theory:

Huffman coding is a popular method for compressing data with variable-length codes. Given a set of data symbols (an alphabet) and their frequencies of occurrence (or, equivalently, their probabilities), the method constructs a set of variable-length codewords with the shortest average length and assigns them to the symbols. Huffman coding serves as the basis for several applications implemented on popular platforms. The Huffman method is somewhat similar to the Shannon–Fano method, proposed independently by Claude Shannon and Robert Fano in the late 1940s .It generally produces better codes, and like the Shannon–Fano method, it produces the best variable-length codes when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon–Fano constructs its codes from top to bottom (and the bits of each codeword are constructed from left to right), while Huffman constructs a code tree from the bottom up (and the bits of each codeword are constructed from right to left).

The algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols.

The steps of Huffman coding algorithm are given below:

1. Arrange the source symbols in increasing order of heir probabilities.
2. Take the bottom two symbols & tie them together as shown in Figure 1. Add the probabilities of the two symbols & write it on the combined node. Label the two branches with a ‘1’ & a ‘0’ as depicted in Figure 1.
3. Treat this sum of probabilities as a new probability associated with a new symbol. Again pick the two smallest probabilities, tie them together to form a new probability. Each time we perform the

combination of two symbols we reduce the total number of symbols by one. Whenever we tie together two probabilities (nodes) we label the two branches with a ‘0’ & a ‘1’.

4. Continue the procedure until only one procedure is left (& it should be one if your addition is correct). This completes the construction of the Huffman Tree.

5. To find out the prefix codeword for any symbol, follow the branches from the final node back to the symbol. While tracing back the route read out the labels on the branches. This is the codeword for the symbol.

The Huffman is an instantaneous uniquely decodable block code. It is a block code because each source symbol is mapped into a fixed sequence of code symbols. It is instantaneous because each codeword in a string of code symbols can be decoded without referencing succeeding symbols. That is, in any given Huffman code, no codeword is a prefix of any other codeword. And it is uniquely decodable because a string of code symbols can be decoded only in one way. Thus any string of Huffman encoded symbols can be decoded by examining the individual symbols of the string in left to right manner. Because we are using an instantaneous uniquely decodable block code, there is no need to insert delimiters between the encoded pixels.

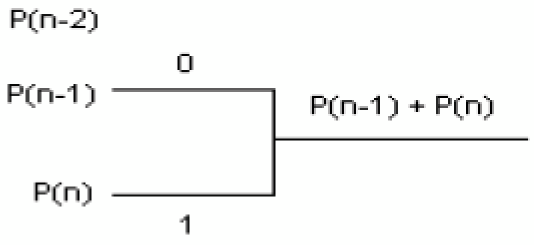
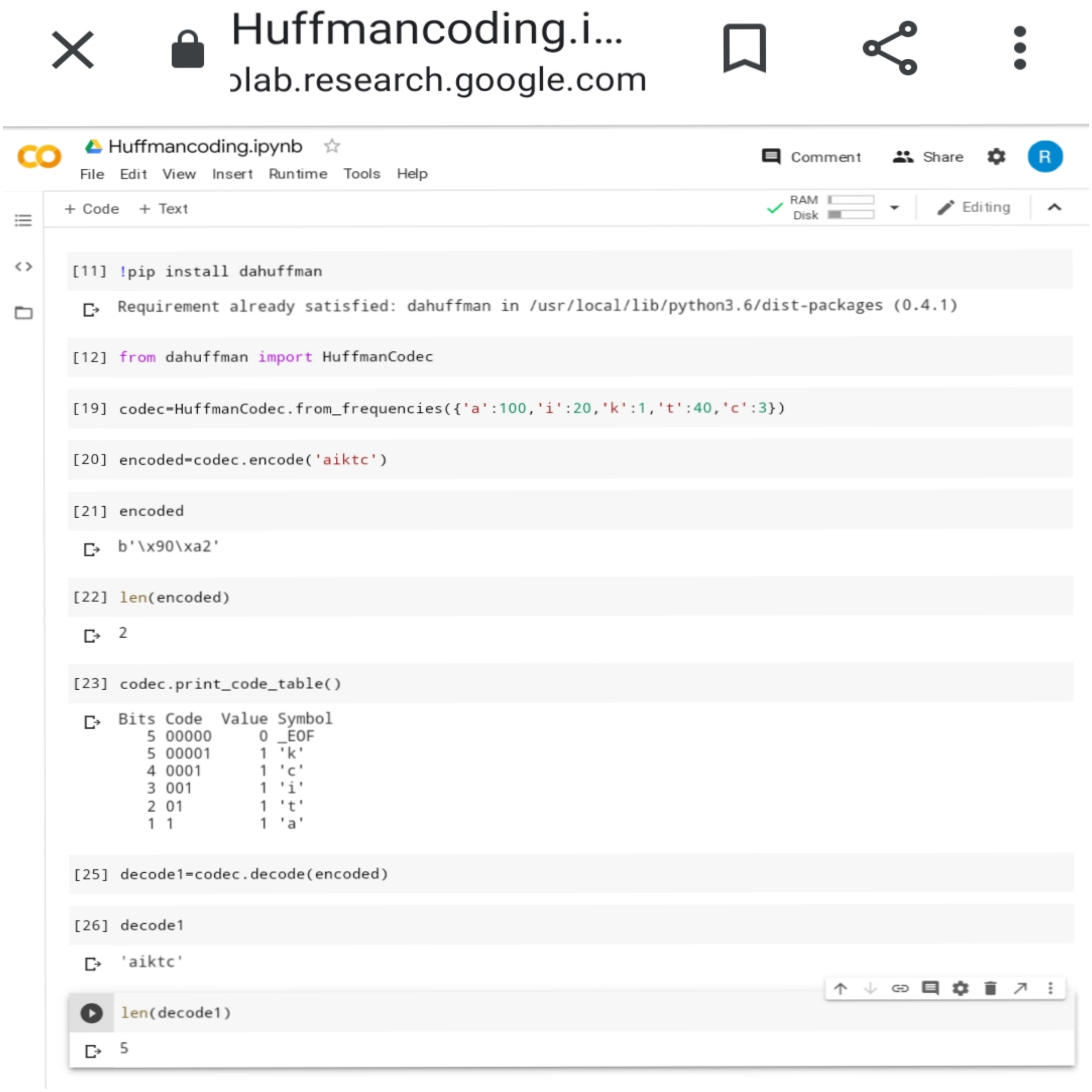


Figure 1

OUTPUT:



```
[11] !pip install dahuffman
Requirement already satisfied: dahuffman in /usr/local/lib/python3.6/dist-packages (0.4.1)

[12] from dahuffman import HuffmanCodec

[19] codec=HuffmanCodec.from_frequencies({'a':100,'i':20,'k':1,'t':40,'c':3})

[20] encoded=codec.encode('aiktc')

[21] encoded
b'\x90\xa2'

[22] len(encoded)
2

[23] codec.print_code_table()
Bits Code   Value Symbol
   5 00000      0 _EOF
   5 00001      1 'k'
   4 0001       1 'c'
   3 001        1 'i'
   2 01         1 't'
   1 1          1 'a'

[25] decode1=codec.decode(encoded)

[26] decode1
'aiktc'

len(decode1)
5
```