ENEE324 Class Project Huffman Encodings

May 13, 2020

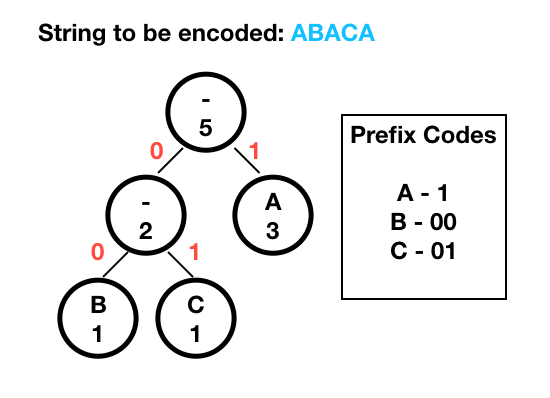
Muhammad Mohd Yusof, Joseph Westra, Samar Sarda

**Introduction**

The goal of this project is to give us hands-on experience on using statistical and probabilistic concepts in a real application. We were able to achieve this goal in this project where we designed and coded a compression algorithm using Huffman coding.

Huffman coding is an algorithm to generate optimal prefix codes for lossless data compression. It works in a way such that characters with higher frequency are encoded with less bits than characters that occur less frequently.

Here is an example of how Huffman Coding works in the diagram below.



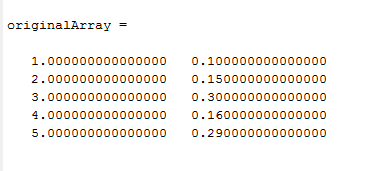
Our code first prompts the user as to whether they wish to perform compression or decompression as can be seen in the code snippet below.

|  |
| --- |
| function huffmanCode(filename, mode)  if mode == 1  huffmanCompress(filename);  elseif mode == 2  huffmanDecompress(filename);  else  error('Mode should be either \n 1 for Compression or \n 2 for Decompression');  end end |

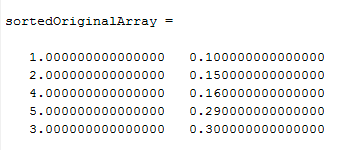
We will first discuss the compression code then the decompression code.

**Compression**

For the first part of the project, we worked on developing a program that takes a .mat file with samples of a random variable to generate a compressed bin file. It also generates a histogram of the values in the original file to have a visual representation of the frequencies.

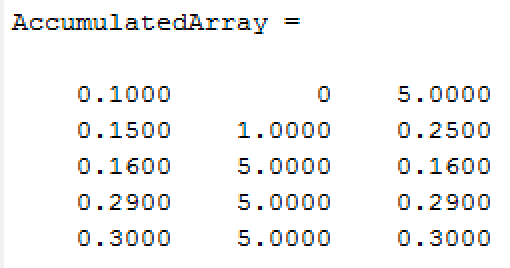
First of all, we created a 2x5 matrix. The first column represents the letters. 1 is A, 2 is B, etc. The second column is the probability that a letter will occur. 

Then, we sorted the matrix in ascending order based on the probability, which gives us the matrix below.

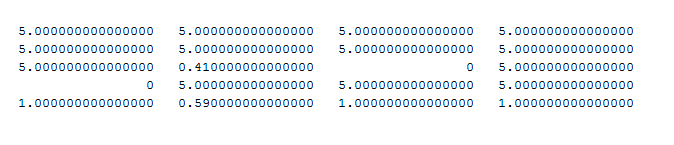
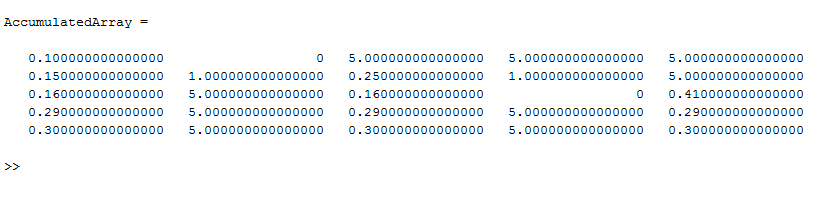


We created an array called AccumulatedArray. AccumulatedArray contains all the information needed when constructing the “tree”. Note that all the 5s in AccumulatedArray will be ignored. The very first column of AccumulatedArray is the sorted probability.

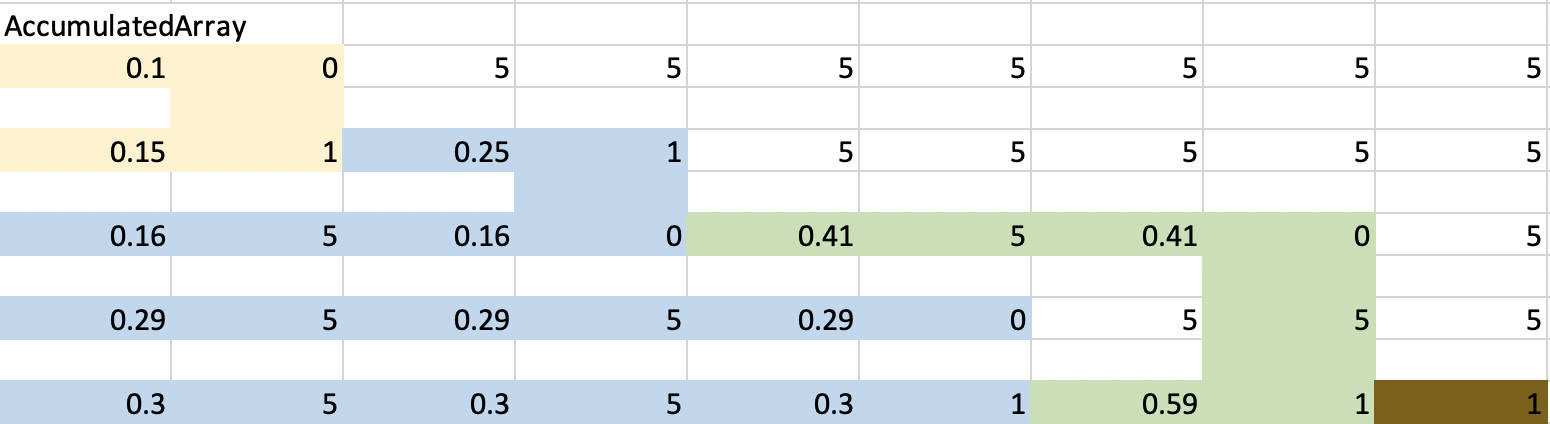
First, we pick the minimum and second minimum probability in the current column and label them as M1 and M2 respectively. For now, the current column is the first column.We then created a second column and put 0 to the right of minimum, 1 to the right of second minimum and 5 to others. Then we added a third column. This column acts as the parents of the leaf nodes(first column). As can be seen in the diagram below, the first and second element of the third column are 5 and 0.25, respectively.



Whenever M1 and M2 are added, we will replace the one with lower index in the column with 5 (useless) and the one with higher index with the sum at the current+2 column. That is the first iteration. We keep doing it until the last column contains only one 1 and the rest are 5s. After finishing the iteration, the result can be seen in the diagram below.

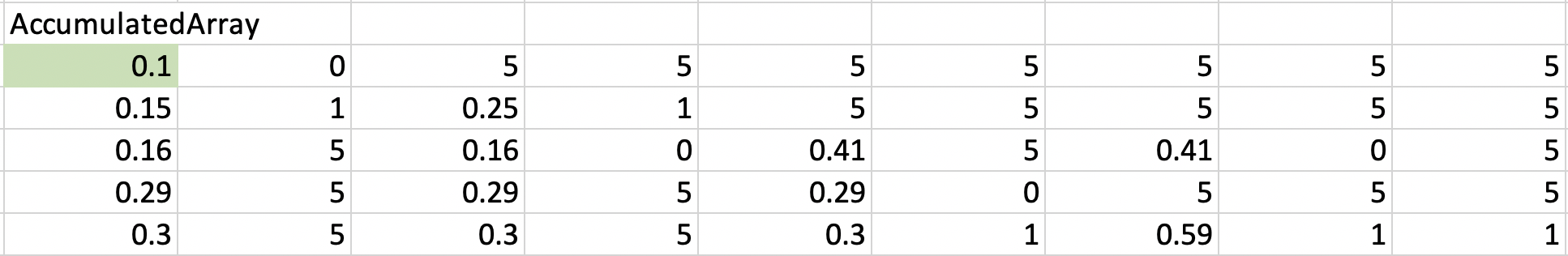


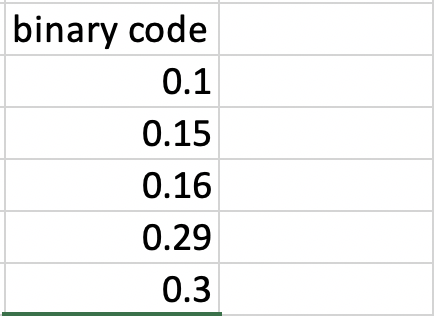
Below is what the tree would look like on Microsoft Excel.



Then this is the crucial part because we are assigning the binary code. In this part, we use a for loop and a while loop. The for loop is used to run through each of the rows and the while loop to run from first column to the last column. We also have a 2x5 matrix to store the binary code

In the first Iteration of for loop, the current location(green) would be 0.1.





While current location is not at the last column, it will iterate two steps to the right. There are 3 scenarios;

(1) if the current location is the same as the previous location

* Do nothing

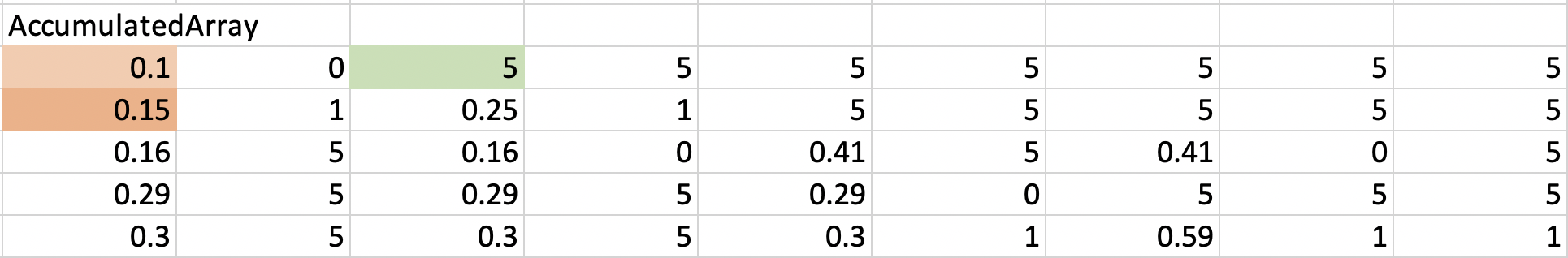
(2) if the current location is 5

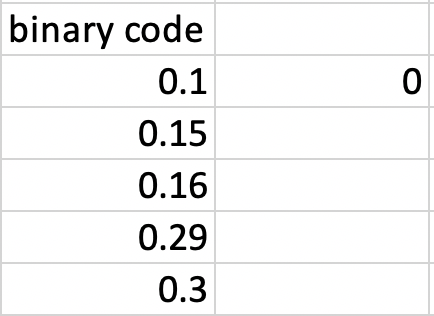
* Concat number in column-1 to binary matrix, identify the higher index between the minimum and second minimum in column-2 and move current location to that index in the same column.

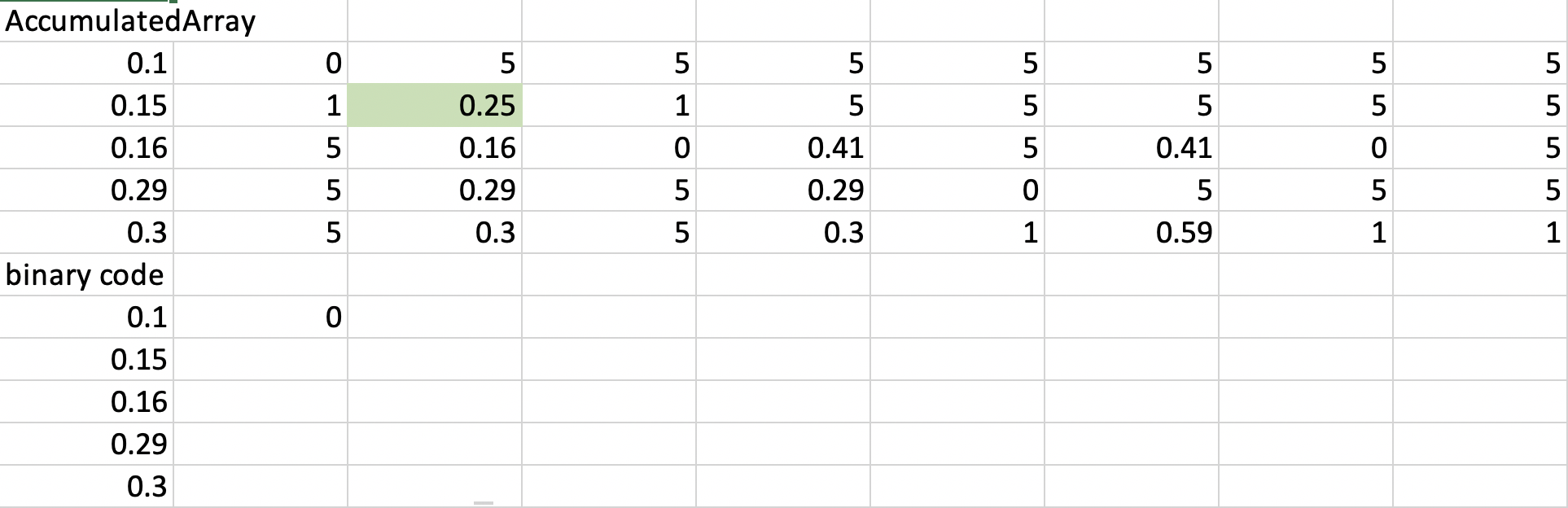
(3) if the current location is not the same as the previous location.

* Concat number in column-1 to binary matrix and stay at the same location

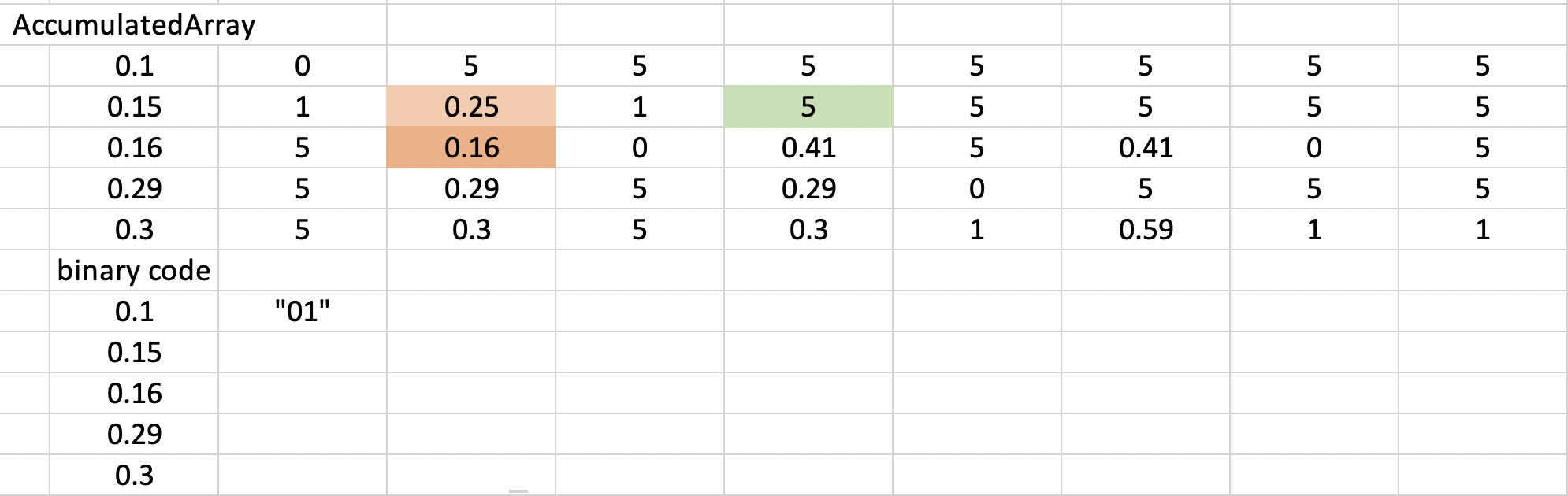
But let’s just follow one step at a time for now. From the current location it goes to the next 2 columns in the same row. As can be seen below, now the current location(green) is 5. This is the scenario (2). Concat the number in column-1 to the binary matrix. Then, look at column-2 and pick the highest index between minimum and second minimum. Move the current location to that index in the same column.

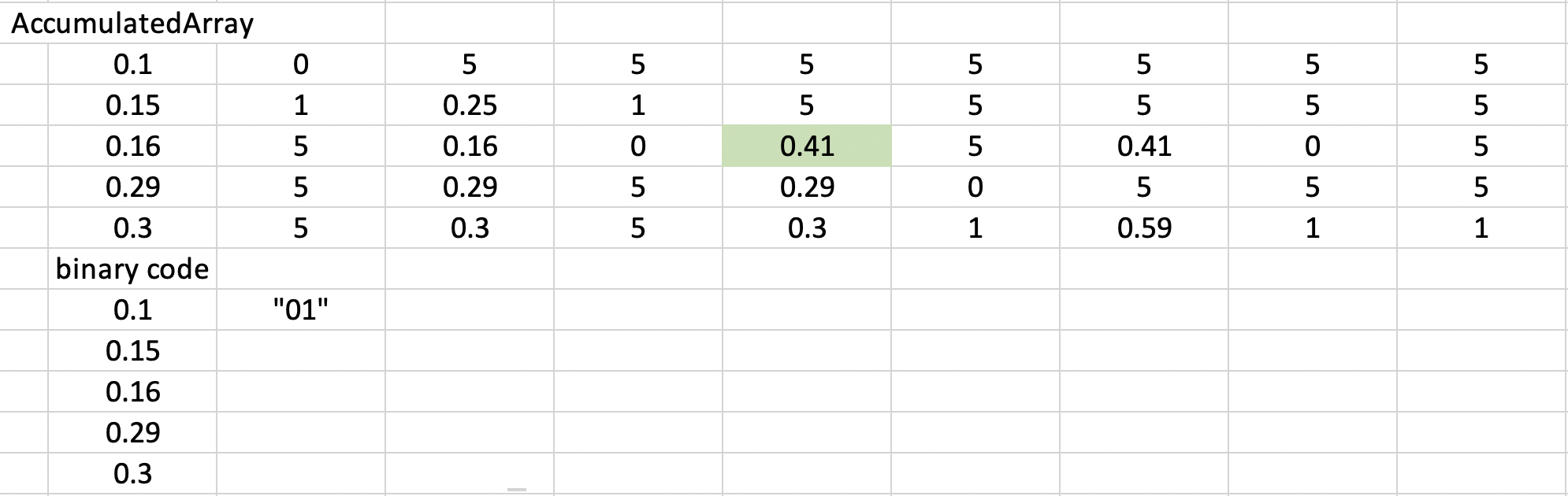




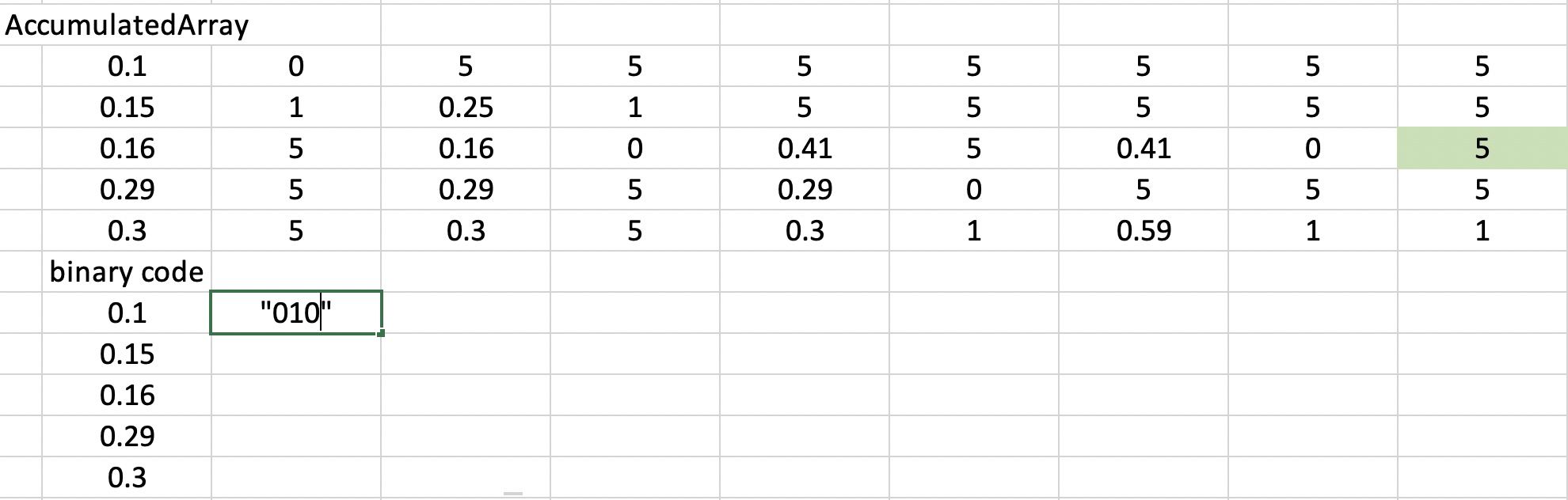
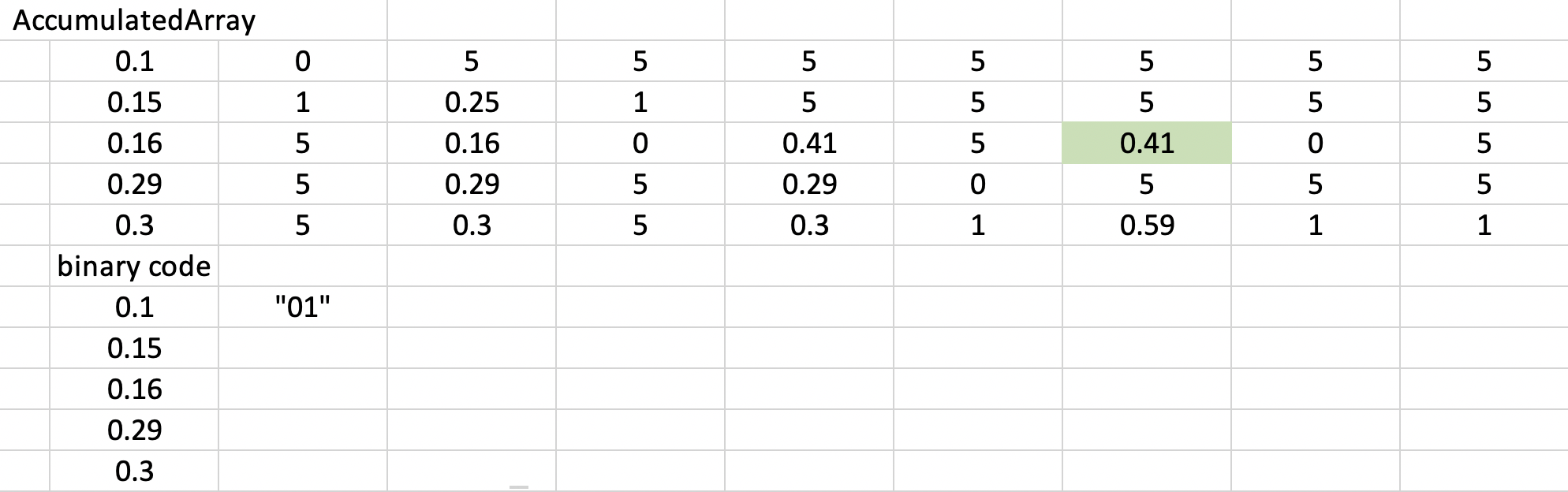


Repeat the while loop process and move to column+2. It is a 5, so it is the same process as before. Concat the number in column-1 to the binary matrix. So, look at column-2 and pick the highest index between minimum and second minimum. Then, move the current location to that index in the same column.

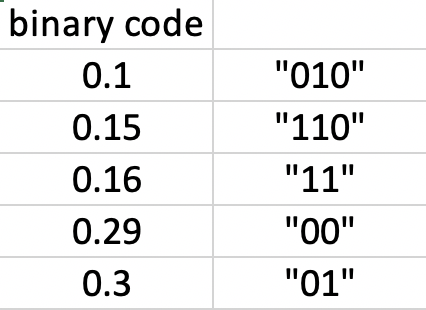




Now, we move the current location to column+2. It is 0.41. It is the same number as column-2. This is scenario (1). So ignore the 5 in column-1 and move to column+2 for the next step.

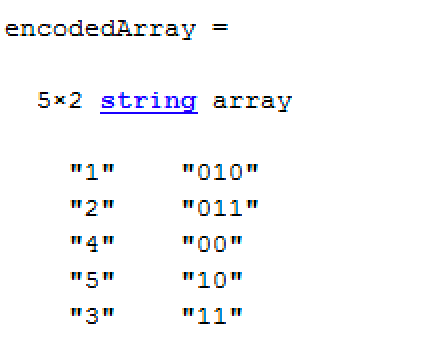


Now that it reaches the last column, the while loop breaks and the code runs the second iteration of the for loop. Eventually, we get the result as shown below.





Note that this is not the final result. In regular Huffman code, we read the tree from top to bottom. In our case, we read it from bottom to top. So, we just need to flip the result and we have the Huffman code.



1 = A, 2 = B, 3 = C, 4 = C, and 5 = C according to the original question.

After generating this encoding, we then have to read through the original data, match each value with its corresponding Huffman code, and then write that code into the binary file as a binary number. We ensure that all the bits are stored in order by specifying a big-endian format in the fwrite function.

If the original data file is called XXX.mat, then the PMF file is called XXX-PMF.mat, and the binary file is called XXX.bin. The PMF is stored in an array called originalArray.

Below is the compression code.

|  |
| --- |
| function encodedArray = huffmanCompress(fileName)  data = load(fileName);   myData = data.testfile;  myMax = max(myData); %max random variable in the list of RVs  myMin = min(myData); %min random variable in the list of RVs    HistMat = zeros(myMax, 2); %create [2 X max] matrix of zeros  HistMat(:,1) = 1:myMax; %number 1 to max for the first column of HistMat    for x = 1:length(myData)  temp = myData(1, x);  HistMat(temp,2) = HistMat(temp,2) + 1; %add 1 at location temp for every occurence of a RV  end    size(myData);    figure(1); %plot the PMF  title('Histogram of PMF Px');   hold on  bar(HistMat(:,2));  plot(HistMat(:,2));  grid on  hold off    sortedHist = sortrows(HistMat, 2); %sort the RVs based on frequency  format long  originalArray = [sortedHist(:,1) sortedHist(:,2)./length(myData)];  %-----------------------compression stage---------------    [path, name, ext] = fileparts(fileName);  y = strcat(name,'-PMF.mat');  N = length(myData);  save(y, 'originalArray', 'N'); %store the Px in (\*user input)-PMF.mat file    sortedOriginalArray = sortrows(originalArray, 2);  AccumulatedArray = [];    freqOri = sortedOriginalArray(:,2);  AccumulatedArray = [AccumulatedArray freqOri];    counter = 0;  freqOriLength = length(freqOri);      while counter ~= freqOriLength  counter = counter + 1;  [M1,I1] = min(freqOri(freqOri > 0));  freqOri(I1) = 5 ;   [M2,I2] = min(freqOri(freqOri > 0));      if M2 == 5  freqOri(I2) = 5; %if second smallest is 5, means M1 is the only element left  else  if I1 > I2 %always add to store the new value at lower side  freqOri(I2) = 5;  freqOri(I1) = M1 + M2;  else  freqOri(I2) = M1 + M2;  end      codeArray = 5\*ones(size(freqOri)) ; % second column to add   codeArray(I1) = 0;  codeArray(I2) = 1;    AccumulatedArray = [AccumulatedArray codeArray freqOri];  end  end  %AccumulatedArray;  encodedArray = [];%mat2str(sortedOriginalArray)  [Row, Column] = size(AccumulatedArray);    for i = 1:length(originalArray)  x\_ind = i;  y\_ind = 1;  location = AccumulatedArray(x\_ind,y\_ind);  binary = "";    while y\_ind < Column  temp = AccumulatedArray(:,y\_ind);  [M1,I1] = min(temp(temp > 0));  temp(I1) = 5 ;  [M2,I2] = min(temp(temp > M1));    y\_ind = y\_ind + 2;  location = AccumulatedArray(x\_ind,y\_ind);    if location == AccumulatedArray(x\_ind,y\_ind-2)    elseif location == 5  bin = AccumulatedArray(x\_ind,y\_ind-1);  binary = strcat(binary, num2str(bin) );    if I1 > I2  x\_ind = I1;  elseif I2 > I1  x\_ind = I2;  end    elseif location ~= AccumulatedArray(x\_ind,y\_ind-2)  bin = AccumulatedArray(x\_ind,y\_ind-1);  binary = strcat(binary, num2str(bin) );    end  end  encodedArray = [encodedArray ; sortedOriginalArray(i,1) reverse(binary)];  end  keySet = str2double(encodedArray(:,1));  valueSet = encodedArray(:,2);    binFileName = [fileName(1:length(fileName)-4) '.bin'];  binFile = fopen(binFileName,'w');  fclose(binFile);  binFile = fopen(binFileName,'a');    % for each value in myData, find its corresponding code and save that  % code to the binFile in binary  for x=1:N  textCode = valueSet(keySet == myData(x));  encoding = ['ubit' num2str(strlength(textCode))];  B = bin2dec(textCode);  fwrite(binFile,B,encoding,'b');  end    fclose(binFile);   end |

**Decompression**

The second part of the project involved decompression which reversed the compression process using a -PMF file that was passed in.

First, we import the PMF file, which contains PX. It also contains N, the amount of values in the file. This considerably simplifies the loop to recover the original data.

Again, we generate the Huffman coding using the same algorithm specified previously. Using this coding to decompress the binary file is trickier, however. Before, we specified the length of each string of bits before writing it to the file. Now we do not know the amount of bits required to represent the code we are scanning in.

To tackle this issue, we used a while loop to append 1 bit from the file at a time to a binary string until it matches one of our Huffman codes. We then take this code and find its corresponding value, thereby recovering one value from our original data array.

If the binary file is called XXX.bin, then the generated data file is called XXX-data.mat. The data array is stored in a variable called myData.

Below is the code for the decompression algorithm:

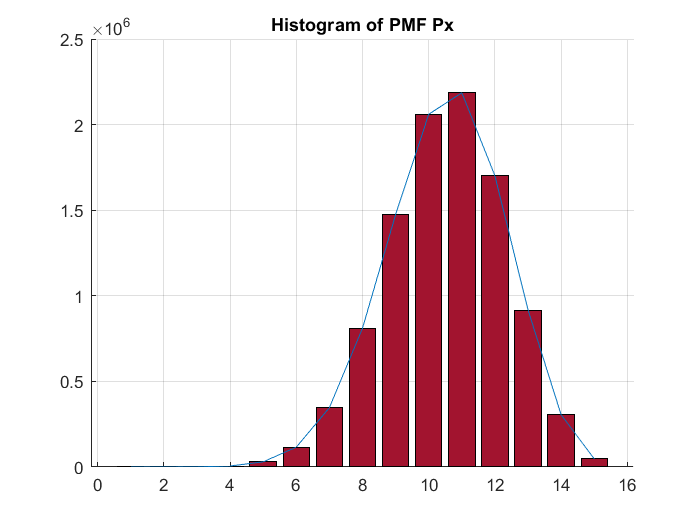
|  |
| --- |
| function encodedArray = huffmanDecompress(fileName)  pmfFileName = [fileName(1:strlength(fileName)-4) '-PMF.mat'];  pmf = load(pmfFileName);  myPMF = pmf.originalArray;  N = pmf.N;    sortedOriginalArray = sortrows(myPMF, 2);  AccumulatedArray = [];    freqOri = sortedOriginalArray(:,2);  AccumulatedArray = [AccumulatedArray freqOri];    counter = 0;  freqOriLength = length(freqOri);      while counter ~= freqOriLength  counter = counter + 1;  [M1,I1] = min(freqOri(freqOri > 0));  freqOri(I1) = 5 ;   [M2,I2] = min(freqOri(freqOri > 0));      if M2 == 5  freqOri(I2) = 5; %if second smallest is 5, means M1 is the only element left  else  if I1 > I2 %always add to store the new value at lower side  freqOri(I2) = 5;  freqOri(I1) = M1 + M2;  else  freqOri(I2) = M1 + M2;  end      codeArray = 5\*ones(size(freqOri)) ; % second column to add   codeArray(I1) = 0;  codeArray(I2) = 1;    AccumulatedArray = [AccumulatedArray codeArray freqOri];  end  end  encodedArray = [];  [Row, Column] = size(AccumulatedArray);    for i = 1:length(myPMF)  x\_ind = i;  y\_ind = 1;  location = AccumulatedArray(x\_ind,y\_ind);  binary = "";    while y\_ind < Column  temp = AccumulatedArray(:,y\_ind);  [M1,I1] = min(temp(temp > 0));  temp(I1) = 5 ;  [M2,I2] = min(temp(temp > M1));    y\_ind = y\_ind + 2;  location = AccumulatedArray(x\_ind,y\_ind);    if location == AccumulatedArray(x\_ind,y\_ind-2)    elseif location == 5  bin = AccumulatedArray(x\_ind,y\_ind-1);  binary = strcat(binary, num2str(bin) );    if I1 > I2  x\_ind = I1;  elseif I2 > I1  x\_ind = I2;  end    elseif location ~= AccumulatedArray(x\_ind,y\_ind-2)  bin = AccumulatedArray(x\_ind,y\_ind-1);  binary = strcat(binary, num2str(bin) );  end  end  encodedArray = [encodedArray ; sortedOriginalArray(i,1) reverse(binary)];  end  keySet = str2double(encodedArray(:,1));  valueSet = encodedArray(:,2);    binFile = fopen(fileName,'r');  myData = zeros(1,N);  for x=1:N  matchesCode = 0;  B = '';  % scan the binary file 1 bit at a time until B matches a code in the Huffman encoding scheme  while matchesCode==0  b = fread(binFile,1,'ubit1','b');  if isempty(b)  b;  break;  end  B = [B num2str(b)];  k=keySet(B==valueSet);  if ~isempty(k)  matchesCode=1;  end  % if no match is found, exit with an error message  if length(B) > max(strlength(valueSet))  error('%s does not match the expected file format for %s. Exiting.',fileName,pmfFileName);  end  end  if matchesCode  myData(x)=k;  else  myData(x) = -1;  end  end  fclose(binFile);    % save the recovered data to XXX-data.mat  dataFileName = [fileName(1:strlength(fileName)-4) '-data.mat'];  save(dataFileName,'myData'); end |

**Test Case 1: test.mat**

To run our program, we use huffmanCode.m, which runs huffmanCompress if mode==1 or huffmanDecompress if mode == 2. It returns the Huffman encoding array generated.

Below is its implementation:

|  |
| --- |
| function encoding = huffmanCode(filename, mode)  if mode == 1  encoding = huffmanCompress(filename);  elseif mode == 2  encoding = huffmanDecompress(filename);  else  error('Mode should be either \n 1 for Compression or \n 2 for Decompression');  end  end |



*test.mat Histogram*

Below is the Huffman encoding generated by running huffmanCode(‘test.mat’,1) or huffmanCode(‘test.mat’,2):

|  |
| --- |
| "1" "110110000000"  "2" "110110000001"  "3" "11011000001"   "4" "1101100001"   "5" "110110001"   "15" "11011001"   "6" "1101101"   "14" "110111"   "7" "11010"   "8" "1100"   "13" "100"   "9" "101"   "12" "111"   "10" "00"   "11" "01" |

Note that when we open test.bin with a hex editor after running this program, we indeed get the correct bits in order. Below is a comparison of the first 18 bits:

|  |
| --- |
| 00110100 10100110 10 00 11010 01 01 00 11010  10,7,11,11,10,7 |

This result is the same as the original array from indices 1:6, so we know that this algorithm is working properly.

When we run both programs and compare the testfile in test.mat to the myData array in test-data.mat, we find that all the values in the arrays are identical.

The entropy can be found in the following way:

|  |
| --- |
| myPMF = load(pmfFileName).originalArray; entropy = -sum(myPMF(:,2).\*log2(myPMF(:,2))); |

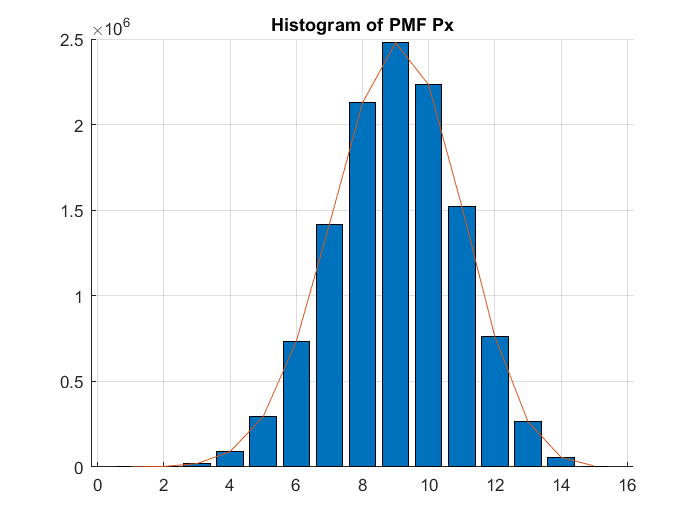
For this PMF, we get an entropy of 2.8662. If we multiply this by the length of the data array, divide by 10242, and divide by 8, we find that the minimum possible size of the binary file is 3.42MB.

|  |
| --- |
| >> entropy.\*length(myData)./1024^2/8  ans =   3.416780234489570 |

By contrast, the real length of the generated .bin file is 3.55MB.



**Test Case 2: test2nd.mat**



*test2nd.mat Histogram*

The Huffman encoding for this file is as follows:

|  |
| --- |
| "1" "10001000000"  "2" "10001000001"  "15" "1000100001"   "3" "100010001"   "14" "10001001"   "4" "1000101"   "13" "100011"   "5" "10000"   "6" "1001"   "12" "1110"   "7" "1111"   "11" "101"   "8" "110"   "10" "00"   "9" "01" |

Again, we test the first 18 bits, using a hex editor and out Huffman encoding:

|  |
| --- |
| 11111011 10101001 10 1111 101 110 101 00 110  7,11,8,11,10,8 |

This indeed corresponds to the original array from indices 1:6.

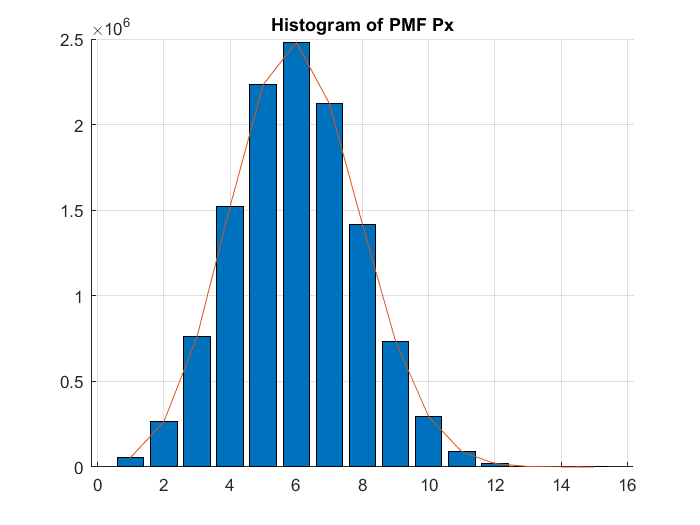
The entropy for this PMF is 2.9690, which corresponds to an expected file size of 4.25MB

|  |
| --- |
| >> entropy.\*length(myData)./1024^2/8  ans =   4.247218619891998 |

The real size of test2nd.bin is 4.45MB.



**Test Case 3: test3rd.mat**



*test3rd.mat Histogram*

The Huffman encoding for this file is as follows:

|  |
| --- |
| "15" "10001000000"  "14" "10001000001"  "13" "1000100001"   "12" "100010001"   "1" "10001001"   "11" "1000101"   "2" "100011"   "10" "10000"   "9" "1001"   "3" "1110"   "8" "1111"   "4" "101"   "7" "110"   "5" "00"   "6" "01" |

The first 19 bits translate as follows:

|  |
| --- |
| 11001111 11111001 110 110 01 1111 1111 00 1110  7,6,8,8,5,3 |

These are the same values we find in the original data at indices 1:6.

The entropy calculated for this dataset is 2.9642. Using the following calculation, this corresponds to a minimum file size of 4.24MB:

|  |
| --- |
| entropy.\*length(myData)./1024^2/8  ans =   4.238257783629065 |

The actual generated file size is 4.44MB.

