Andy W - sfu cs365 Midterm 2 CheatSheet 11/20/2025

Compression: the process of coding that will effectively reduce the total number of bits needed to represent certain information. After Compression, add a new over header on top of the data.
== Compression Ratio = (size) before / after
Entropy - Self information
New info and no new info are inverse relationship
New information = high Pr() = rare events have high info content
No new info = low Pr() = common events have low info content
= info of an event is a function of its pr(): i(A) = f(Pr(A))
2 independent events = P(AB)=P(A)P(B) → i(AB) = i(A) + i(B)
By Shannon's Definition = Self-information of an event is

$$i(x) = log_b \frac{1}{Pr(A)} = -log_b Pr(A)$$

Pr(A) in [0, 1]; b=2 = unit info in bits, if b=8, i(x)=1, then pr(A) = 1/8
info and prob have inverse relationship.
First Order Entropy (Simply Entropy):
= avg of self-info of the data set
The first-order entropy represents the average minimal number of bits needed to losslessly represent one output of the source.
Pr(Ai) = Pr(Event), -log_2 Pr(A) = self info

$$H = \sum -Pr(A) \cdot log_2 Pr(A)$$

The entropy η = the average amount of information contained per symbol in the source S: η specifies the lower bound for the average number of bits to code each symbol in \mu \leq l_bar

Variable Length Coding = minimizing the avg codeword length (num of bits) loselessly - approach the entropy of the source
Runlength Coding
= Memoryless Source - an info source that is independent distributed
= Source with memory - data generated from the source depends on previous generated data
= Run-Length Coding (RLC) exploits the memory in the source = Encode group pattern. Ex: 111 00 111 = (1, 3), (0, 2), (1, 3)
Entropy Coding: Prefix-free Code - prefix code
= No codeword is a prefix of another one. = Can uniquely decoded
= The code only used by 1 character
= use binary tree (left subtree starts 0, right subtree starts 1)
Morese Code don't work = different length and same prefix

============Shannon-Fano Compression==================
a top-down approach of a binary tree.
1= sort all symbols according to the frequency of occurrences
2= recursively divide the symbols into two parts. Each with approximately the same number of counts. Until all parts contain only one symbol.
Ex: Hello: L = 2. H, E, O = 1 Start with root = 5, L(2) on the left and HEO(3) on the right. Separate again, put H(1) on the left and EO(2) on the right. Then E(1) and O(1). Result in SF coding table. Each letter has a code: L=0, H=10, E=110, O=111. Total num of bits used = 2+2+3+3=10. The binary tree is not unique.

============**Huffman Coding**============================
Goal=Construct optimal prefix-free code: start from the leaf nodes
Frequent symbols have short codes. and the two codewords that occur least frequently will have the same length.
Limitations of Huffman Code:
Need a probability distribution
= Usually estimated from a training set ("steady file")
= not good for constantly change(video)
Minimum codeword length is 1 bit
= Serious penalty for very-high-probability symbols
= The table will be too large
Ex: Binary source, P(0)=0.9
= Entropy: 0.469; huffmanCode = 0,1
= Avg code length = 1bit/symbol. Very redundancy
Init: sorted all symbols on a list base on frequency counts.
Repeat until the list has only one symbol left:
= From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node.
= Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
= Delete the children from the list.
Assign a codeword for each leaf based on the path from the root
Properties of Huffman Coding
Binary tree = left leaf/subtree <= right leaf/subtree
Unique Prefix Property:
=No code is a prefix of other codes.
=No ambiguity in decoding
Optimality:
= Optimal prefix code given accurate probability distribution
= Most frequent symbol – shortest code
= Least frequent symbol – longest code
Average Huffman code length is strictly less than entropy + 1
= L bar < n + 1 OR H(s) <=L <= H(s) + 1
Ex: "HELLO" -> H(1) E(1) L(2) O(1) = [(L, 2), (H, 1), (E, 1), (O, 1)]
= create with (E, 1), (O, 1), root=2. Get next symbol H(1) to the elft and create a new subtree with root = 2+1=3. Get next symbol L(2) to the left and create a new tree with root=3+2=5.
= Coding for "HELLO" L(1), H(10), E(110) and O(111). Avg codeward length = 1(2/5)+2(1/5)+3(1/5)+3(1/5) = 2bit / symbol
Ex: A = {a, b, c, d, e}, Pr = {0.2, 0.4, 0.2, 0.1, 0.1}, code = {01, 1, 000, 0010, 0011}
Entropy = -(0.2*log2(0.2)*2 + 0.4......)
Avg Huffman codeword length = L = 0.2*2 + 0.4*1... = 2.2bit/symbol
= H(s) <= L < H(s) + 1

===**Huffman Decoding**===
Direct Approach: Slow = Read one bit, compare with all codewords
Binary tree approach: slow = Embed the Huffman table into a binary tree data structure; For loop: { if 0, go left; if 1, go right. Decode a symbol when a leaf is reached}
Example below
Table Look-up Method:
= N: # of codewords; L: max codeword length
Expand to a full tree:
= Each Level-L node belongs to the subtree of a codeword.
= Equivalent to dividing the range [0, 2^L] into N intervals,
= each interval corresponding to one codeword.
If a symbol is not a leaf node, the interval will be bigger, more code used to represnet. Higher the symbol = more frequent.
Ex: 000:{'a', 2}, 001:{'a', 2}, 010:{'b', 3}, 011:{'c', 3}
'a' is on the second = two leaf node under a. 2 = number of bits to store. 'b' and 'c' are leaf nodes.
Example below
+++CODE

Extended Huffman Code (joint Huff..Code)
= Treat k symbols as one symbol (X1, X2...Xk)
= k=group size, 1 codeword = k symbols
= Alphabet increased exponentially: N^k
K=2, read 2 symbol at a time. (no overlap)
Second Order Entropy = H(x, x+1) = sum - Pr(x, x+1) log_2 Pr(x, x+1)
H per symbol = H / 2

============**LZW Coding**================================
Used in Zip, Dictionary based
Uses fixed-length codewords to represent variable-length strings, symbols or character that commonly occur together
= Also slow for dynamic file change
= minimum codeword length is 1 bit
= serious penalty for high-probability
Ex: Example: Binary source, P(0)=0.9
= entropy = -0.9*log2(0.9)-0.1*log2(0.1) = 0.469 bit
Huff..Code = 0, 1: avg code length = 1 bit
Joint coding is not practical for large alphabet.
= Encoder and decoder build up the same dictionary dynamically while receiving the data.
= not everything is needed to store to dic such as a-z, A-Z

+++CODE+++

Ex: With Input: AABABAB. Set initially Dictionary to be A: 0, B: 1.
AA not in dic, store AA:2 into dic. Put 0 to output
AB not in dic, store AB:3 into dic. Put 0 to output
BA not in dic, store BA:4 into dic. Put 1 to output
AB in dic, but ABA not, store ABA:5 into dic, put 3 to output
AB in dic, put 3 to output
EOF = return 00133.
Without compression with ASCII, we need 8*7=56 bits
With compression, we got 3*5 = 15 bits.

============**Arithmetic Coding**==========================
Overview: Arithmetic coding applies Huff recursively
= normalizes the range [0, 2^L] to [0, 1], and map a sequence to a unique tag in [0, 1)
= Disjoint and complete partition of the range
= all interval is a CDF.
= Each interval corresponds to one symbol
= Interval size is proportional to symbol probability
= least frequent = small size
= more grequent = larger interval size
= encode a sequence of symbols at a time
= The first symbol restricts the tag position to the interval
= The reduced interval is partitioned recursively

CDF = Cumulative Density Function
= for discrete probability distribution
= start with CDF(0) = 0, add up to 1.
Fx(i) = P(X<=i) = SUM^i Pr(X = k)

============**Arithmetic Encode**
= Map probability to [0, 1)
= order does not amtter, but follow abc, 123 order
= Disjoint but complete partition into 3 intervals
Ex: Pr(1) = 0.8, Pr(2) = 0.02, Pr(3) = 0.18
1 = Encode symbol (1) Map the 3 interval from [0,1) to [0, 0.8)
2 = ratio don't change with the new 0.8 range.
3 = encode 2nd symbol (3) Map the 3 intervals from [0, 0.8) to [0.656, 8), 0.656 = original 3 interval: [0.82-1.0] 0.82* 0.8 = 0.656
4 = ratio don't change with new 0.144 range
5 = encode next symbol (2) and map between [0.7712, 0.77408)
New lower bound = lower bound + new range * pr()
0.7712 = 0.656 + 0.144 * 0.8 = lower bound
0.77408 = 0.656 + 0.144 * (0.8 + 0.02) = upper bound
6 = return 0.7712, ratio don't change but there are floating point precision error.
The return encoded num can be any number from the final interval. But usually use the lower bound.

+++CODE+++

============**Arithmetic Decode**
= Need: the double, # of symbol.
= Find which interval 0.7712 is part off
1 = Decode 1 because 0.7712 is between the first interval [0, 0.8)
2 = Map the 3 interval from [0,1) to [0, 0.8)
3 = in new range, 0.7712 is part of the interval 3.
4 = Map the 3 intervals from [0, 0.8) to [0.656, 8)
5 = in new range, 0.772 is between the second interval
6 = map the 3 intervals between [0.7712, 0.77408)
7 = 0.7712 = lower bound for the first interval. Decode '1' and stop
WE NEED TO recalculate all thresholds each time

+++CODE+++

============**Arithmetic Simplified Decode**
= Normalize RANGE to [0, 1) each time
= No need to recalculate the thresholds
New x = (X_old - low) / range
1 = Receive 0.7712 Decode(1) = fall under first interval
2 = x = (0.7712-0) / 0.8 = 0.964 Decode (3)
3 = x = (0.964-0.82) / 0.18 = 0.8 Decode (2)
4 = x = x =(0.8-0.8) / 0.02 = 0 Decode (1)

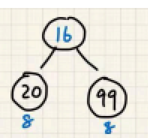
============**Scaling and Incremental**
= need high precision
= No output is generated until the entire sequence is encoded
= As the RANGE reduces, many most significant bit's (MSB's) of LOW and HIGH become identical:
Such 0.123 and 0.103 => 0.0... and 0.0...
= re-scale and remove the common MSB
= infinite precision and finite precision integers
E1 and E2 Scaling, make sure lower < 0.5 and upper >= 0.5
E1 = when low and high all less than 0.5
= both starts with 0.0... = E1(range) = 2 * range
E2 = when low and high all greater or equal to 0.5
= both starts with 0.1... = shift by 1 bit = E2(range) = 2(x – 0.5)

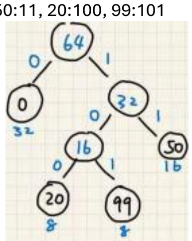
E1 and E2 scaling Encoding
Same with Arithmetic Encoding, but if there exits a common MSB, rescale using E1 or E2.
= Each time scalled, push the common MSB in to the output queue. Starts left, end right.
= After encoded, add 1 to the output. Because that is 0.5 (middle of the interval). The last number must be 1 to make sure the value will be part of the final interval.
= Complete all possible scaling before encoding the next symbol
The output 1100011 will pass down as 0.1100011
= Even without any scalling, we still get the same output
= In the decoder side, we don't need to know when it was scaled
= If without scaling, we won't know if 0.5 exists in the interval

7.8 Q1: Calculate the entropy of a "checkerboard" image in which half of the pixels are BLACK and half of them are WHITE 8x8 = 64, Pr(black) = 0.5, Pr(white) = 0.5 Formula = $H = -\sum_i P_i * \log(P_i) = -[0.5 * \log_2(0.5) + 0.5 * \log_2(0.5)] = -2 * 0.5 * \log_2(0.5) = -\log_2(0.5) = 1$ = 1 bit per pixels

7.8 Q3: (a) What is the entropy η of the image below, where numbers (0, 20, 50, 99) denote the gray-level intensities?
Frequencies x number = total = 64
32 x 0 -> pr(0) = 1/2
8 x 20 -> pr(20) = 1/8
16 x 50 -> pr(50) = 1/4
8 x 99 -> pr(99) = 1/8
Entropy = $-[\frac{1}{2}\log(\frac{1}{2}) + \frac{1}{8}\log(\frac{1}{8}) + \frac{1}{4}\log(\frac{1}{4}) + \frac{1}{8}\log(\frac{1}{8})] = -[-\frac{1}{2} + -\frac{3}{8} + -\frac{1}{2} + -\frac{3}{8}] = \frac{1}{2} + \frac{3}{8} + \frac{1}{2} + \frac{3}{8} = (4 + 3 + 4 + 3)/8 = 14/8 = 1.75$ bits per pixel, and round up to 2 bits per pixel

(b) Show step by step how to construct the Huffman tree to encode the above four intensity values in this image. Show the resulting code for each intensity Value.
Step 1: Sort the number based on its occurrences:
- 20:8, 99:8, 50:16, 0:32
Step 2: Pick two numbers with lowest occurrences, insert to the tree of leafs and use their sum as its roots (for now, later will become parents)
Step 3: pick the next number with lowest occurrences and insert as a new leaf of a tree and add the sum from the subtree and use it as roots. The huffman tree follows the rules of a heap.
Number : Code. 0:0, 50:11, 20:100, 99:101





What is the average number of bits needed for each pixel, using your Huffman code?
The average code length = $\frac{1}{2}(1) + \frac{1}{8}(3) + \frac{1}{4}(2) + \frac{1}{8}(3) = 1.75$ bits per pixel. Round up to 2 bits per pixel.

EX1: Given sequence 00100011001011011101001110111011
(a) What is the entropy of this sequence?
1 - frequency = 18 = pr(1) = 18/32
0 - frequency = 14 = pr(0) = 14/32
Entropy = $-[18/32 * \log_2(18/32) + 14/32 * \log_2(14/32)] = 0.9887$

(b) What is the second order entropy of this sequence
Pairs for these binaries: 00, 01, 10, 11 = 31 pairs
00 - frequency = 5 = pr(00) = 4/16
01 - frequency = 9 = pr(01) = 2/16
10 - frequency = 8 = pr(10) = 4/16
11 - frequency = 9 = pr(11) = 6/16
Second order Entropy = $-[4/16 * \log_2(4/16) + 2/16 * \log_2(2/16) + 4/16 * \log_2(4/16) + 6/16 * \log_2(6/16)] = 1.9056$
Entropy per symbol = 1.965 / 2 = 0.9528

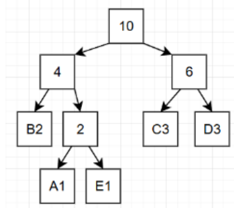
(3) What is the average codeword length if we use 2 symbol joint Huffman coding? Answers should be in per symbol
Combine 2 smallest frequency = 5/31 + 8/31 = 13/31 = 0.419
Combine the next smallest frequency = 9/31 + 13/31 = 22/31 = 0.7097
Total = 31/31 = 1.000
- To store all 4 pairs, it requires 2 bits in the huffman code, the average code length per pair is 2 bits. Therefore, it needs 1 bit per symbol.

7.8 Q7: Arithmetic coding and Huffman coding are two popular lossless compression methods. (b) Suppose the alphabet is [A, B,C], and the known probability distribution is P(A) = 0.5, P(B) = 0.4, P(C) = 0.1. For simplicity, let's also assume that both encoder and decoder know that the length of the messages is always 3, so there is no need for a terminator.
i. How many bits are needed to encode the message BBB by Huffman Coding? Put ABC into huffman tree and its coding:
A = 0 = 1 bit | B = 10 = 2 bits | C = 11 = 2 bits
When encoded [BBB] in Huffman Coding, it equals < 10 10 10 >, which is 6 bits.
ii. If Arithmetic coding is used, what are the lower and higher bounds of the interval after encoding BBB and ABC, respectively.
P(A) = 0.5, P(B) = 0.4, P(C) = 0.1;
CDF is A=[0.0,0.5), B=[0.5,0.9), C=[0.9,1.0).

For [BBB]:
Encode first B, new range = [0 + 0.5, 0 + 0.9) = [0.5, 0.9)
Encode second B = [0.5 + 0.4×0.5, 0.5 + 0.4×0.9) = [0.7, 0.86)
Encode Third B, new range = [0.7 + 0.16×0.5, 0.7 + 0.16×0.9) = [0.78, 0.844)
Then the Arithmetic coding interval is [0.78, 0.844)

For [ABC]
Encode First A, new range = [0 + (1−0)*0, 0 + (1−0)*0.5) = [0.0,0.5)
New width = 0.5
Encode Second B, new range = [0 + 0.5*0.5, 0 + 0.5*0.9) = [0.25. 0.45) = New width = 0.2
Encode Third C = [0.25 + 0.2*0.9, 0.25 + 0.2*1) = [0.43, 0.45)
Then the Arithmetic coding interval is [0.43, 0.45)

Huffman Decode using binary tree approach with 001011



1 = read 0, go left
2 = read 0, go left
3 = reach leaf node, decode B
4 = go back
====
1 = read 1. Go right
2 = read 0
3 = reach leaf node, decode C
4 = go back
====
1 = read 1
2 = read 1
3 reach leaf node, decode D
4 = go back
OUTPUT BCD

Huffman Decode using look up table with 001011
1 = expand the tree, fill all the levels, with L = level
2 = construct a table
Level, length (character's position from root), code = code = binary with length L

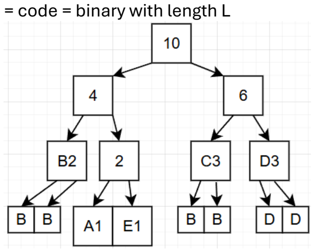


Table:

Symbol	Length	Code
B	2	000
B	2	001
A	3	010
E	3	011
C	2	100
C	2	101
D	2	110
D	2	111

1 = read L bits and decode - 001 = B
2 = shift left with B length – 001011 = 10111
3 = read next L bits and decode – 101 = C
4 = shift left with B length, and fill 0 for padding
5 = read next L bits and decode – 110 = D
6 = return decoded BCD

Construct a Huffman Coding tree
Given ABBCCCDDDE
A=1, B=2, C=3, D=3, E=1
1 = Sort from most to least frequent = [CDBAE]
2 = build Huff Tree for A E with P1(2)
3 = Sort from most to least frequent = [CDB P1(2)]
4 = build Huff Tree for B P1(2) with P2(4)
5 = Sort from most to least frequent = [P2(4), C, D]
6 = build Huff Tree for C D with P3(6)
7 = Sort again
8 = build Huff Tree for P3, P2 = P4(10)
9 = P4 = huffman tree

Encode ABAAB with LZW Dic
Init dic = A:1 and B:2 AND Output = Null
1 = read "A", exist, read "A" + "B"
2 = "AB" not in Dic, store AB: 3 into dic, Output = 1
3 = read "B", exit, read "B"+ "A"
4 = "BA" not in dic, store BA: 4 into dic, Output = 12
5 = "read A, exist, read A + A
6 = AA not in dic, store AA: 5 into dic, Output 121
7 = read A exist, read A+B
8 = AB exist in dic, read next, end string
9 = Output 1213
Table = A0, B0, AB3, BA4, AA5

Decode 1213 with LZW = NA
Init dic = A0, B1 AND Output = Null

```
//Huffman Decode Table Look-up Method
// reads first 3 bits.
x = ReadBits(L);
k = 0; //# of symbols decoded
While (not EOF) {
    symbol[k++] = HuffDec[x][0];
    length = HuffDec[x][1];
    // left shift -> more room on the right side.
    // left most bit is dropped
    x = x << length;
    //read new bit and add to x
    newbits = ReadBits(length);
    x = x | newbits;
    // read only first 3 digits.
    x = x & 0b111;
}
```

```
// LZW Coding ==================
s = next input character;
while not EOF {
    // while not empty, read next character
    c = next input character;
    // if it exists in the dictionary
    if s + c exists in the dictionary
        // add another character (next loop)
        s = s + c;
    else { //not exist
        // store the current to output
        output the code for s;
        // put the s + C to the dic
        add string s + c to the dictionary with a new code;
        s = c;
    }
}
return output
```

```
//Arthmetic Encode ==================
low=0.0; high=1.0; //inti bound
while (not EOF) {
    // read next symbol
    n = ReadSymbol();
    // update range first
    RANGE = HIGH - LOW; // new range from last loop
    // CDF(n) = current symbol's upper bound.
    HIGH = LOW + RANGE * CDF(n);
    // CDF(n-1) = previous's symbol
    LOW = LOW + RANGE * CDF(n-1);
}
output LOW;
```

```
//Arthmetic Decode ==================
Low = 0; high = 1;
x = Encoded_number
While (x ≠ low) {
    n = DecodeOneSymbol(x);
    output symbol n;
    // expand
    x = (x - CDF(n-1)) / (CDF(n) - CDF(n-1));
}
```