- Quantizer
    - n-level -> split into n intervals
- Uniform
    -
        - Q(x)=$\lfloor \frac{x-d_0}{\Delta} \rfloor$ // this gives i where $x \in [d_i \; d_{i+1})$ //Exercise: Prove that
        - $Q^{-1}(i) = \mathrm{d}_0 + \left(i + \frac{1}{2}\right)\Delta$ // that is because $Q^{-1}(i) \stackrel{\text{def}}{=} r_i$ and $r_i = \mathrm{d}_0 + \left(\mathrm{i} + \frac{1}{2}\right)\Delta$
- Non-uniform
    - q: binary search
    - deq: use r_i
- Semi uniform
    -
        - Quantization: as in uniform quantizers, Q(x)=$\lfloor \frac{x-d_0}{\Delta} \rfloor$, which takes O(1) time
        - Dequantization: $Q^{-1}(i) = r_i$, which takes a constant time (O(1))
- Max LLOYD
    - NON-UNIFORM
    - Initial the value as uniform q
    - loop
        - ri= average in [di,di+1)
        - di = (ri-1+ri)/2
    - until di doesn't change much
- Transform
    -
        - Specifically, the transforms must have the following properties:
            - **Decorrelation of data**
            - **Separation of data** into vision-sensitive data and vision-insensitive data
            - **Energy compaction**: concentrating the important data into a very small subset
            - **Invertability:** since data loss should occur only in quantization, transforms must be lossless
    - decorrelated data: less blurring, less loss of patterns.
- • **Example:** $\begin{bmatrix} 1 & 5 \\ 3 & 4 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 4 & 1 & 3 \\ 1 & 2 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 14 & 21 & 28 \\ 10 & 20 & 19 & 29 \\ 5 & 10 & 6 & 11 \end{bmatrix}$
-

THE MATRIX OF THE HADAMARD TRANSFORM

- To express a_{kl}, we need to express k and l in binary (using $n = \log N$ bits)
    - $k = k_{n-1}k_{n-2} \dots k_1 k_0, \;\; l = l_{n-1}l_{n-2} \dots l_1 l_0$
    - $a_{kl} = \sqrt{\frac{1}{N}}(-1)^{k_{n-1}l_{n-1}+k_{n-2}l_{n-2}+\cdots+k_1 l_1 + k_0 l_0}$
- $A_N^{-1} = A_N^T = A_N$
- Illustrations:

$$A_2 = \sqrt{\tfrac{1}{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad A_4 = \tfrac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

$$A_8 = \sqrt{\tfrac{1}{8}}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{pmatrix}$$

-

AN ALTERNATIVE DEFINITION OF THE MATRIX OF THE HADAMARD TRANSFORM

- The Hadamard matrix can be defined recursively (where N is a power of 2, i.e., $N = 2^n$ for some positive integer n)
- $A_1 = (1)$ and $\forall N > 1, A_N = \sqrt{\frac{1}{2}}\begin{pmatrix} A_{\frac{N}{2}} & A_{\frac{N}{2}} \\ A_{\frac{N}{2}} & -A_{\frac{N}{2}} \end{pmatrix}$
- **Exercise**: Derive $A_2, A_4$and A_8 using this recursive definition
- **Exercise**: Compare the values $A_2, A_4$and A_8 with their values on the previous slide to verify that the two definitions are equivalent
- **Exercise**: Using the recursive definition of A_N, prove by induction on n that A_N is a symmetric matrix, and that $A_N A_N = I_N$, i.e., $A_N^{-1} = A_N$.
- **Note**: In Matlab, if you call hadamard(N), it returns to you the Hadamard matrix A_N but without the constant multiplier $\sqrt{\frac{1}{N}}$

-

SPEED OF THE TRANSFORM
-- 2D SIGNALS --

- For 2D input signals, i.e., an $N \times M$ image X
- Recall that $Y = A_N X A_M^T$ (i.e., we transform every column then every row)
- Every column of A takes O(N^2) time to compute, and so the M columns take O(MN^2) time
- Every row takes O(M^2) time to compute, and so the N columns take O(NM^2) time
- Thus, the 2D transform takes O(MN^2 + NM^2)= O(N^3) for $N = M$
- For N=1000, O(N^3)=1 second on a 1GFLOPS device (or 17 mins on a 1MFLOPS)
- But even on a 1GFLOPS device, applying the 2D transform on all the frames of a 2-hour video (with a 30fps rate) takes 2.5 days!

- Lossless
  - DPCM

DPCM
-- MAIN METHOD (FOR 1D DATA): PARAMETER --

- The choice of the parameter a is up to the user, and can be adapted to your data/apps
- It can also be optimized to fit your input data:
  - Let $E(a) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n}(x_i - ax_{i-1})^2$, assuming that $x_0 = 0$
  - The smaller the magnitudes of the residuals e_i, the better, since that would increase the redundancy (and make certain residual values occur with high probabilities)
  - Therefore, we are interested in minimizing $E(a)$
  - To get the optimal value of a that minimizes $E(a)$, compute the derivative $E'(a)$, set it to 0, and solve for a
  - $E'(a) = -2\sum_{i=1}^{n} x_{i-1}(x_i - ax_{i-1}) = -2[\sum_{i=1}^{n} x_{i-1}x_i - a\sum_{i=1}^{n} x_{i-1}^2]$,
  - Setting $E'(a) = 0$, and solving, we get $a = \frac{\sum_{i=1}^{n} x_{i-1}x_i}{\sum_{i=1}^{n} x_{i-1}^2}$
  - If x is integers and you want to keep the residues integer, take $a = \text{NINT}(\frac{\sum_{i=1}^{n} x_{i-1}x_i}{\sum_{i=1}^{n} x_{i-1}^2})$

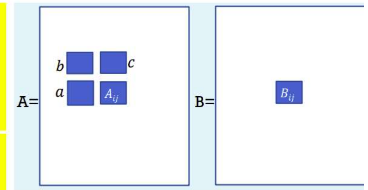
> - NINT(z) = nearest integer to z
> - Ex: NINT[1.7]=2,
> - NINT[1.3]=1
> - NINT[1.5]=2

DPCM
-- MAIN METHOD: FOR 2D DATA--

- DPCM for 2D data (i.e., an image A)
- DPMC for 2D data takes three parameters (a, b, c)

Coder Method (to code image A):
1. Compute a residual image B:
   $B[i,j] = A[i,j] - (aA[i,j-1] + bA[i-1,j-1] + cA[i-1,j])$;
   // any pixel out of boundary is assumed = 0
2. Code the residual B with Huffman (or any other suitable lossless coder like Bitplane Coding to be covered later)

Decoder Method:
1. Decode the coded bitstream with the right decoder, to get the residual image B
2. For $i = 1{:}n$ do // image is an $n \times m$ matrix.
   For $j = 1{:}m$ do // pixels out of boundary are = 0
   $A[i,j] = B[i,j] + (aA[i,j-1] + bA[i-1,j-1] + cA[i-1,j])$;



DPCM
-- MAIN METHOD (FOR 2D DATA): PARAMETERS --

- The choice of the parameters (a, b, c) is up to the user, and can be adapted to your data/applications
- It can also be optimized to fit your input data:
  - Let $E(a,b,c) = \sum_{i=1}^{n}\sum_{j=1}^{m} B_{ij}^2 = \sum_{i=1}^{n}\sum_{j=1}^{m}[A[i,j] - (aA[i,j-1] + bA[i-1,j-1] + cA[i-1,j])]^2$
  - Compute the partial derivatives of E with respect to $a, b,$ and c, set them to 0, and solve the system of 3 linear equations with three unknowns (the unknowns are $a, b,$ and c)
- **Exercise**: Carry out the above step, and derive formulas for optimal $a, b,$ and c.
- Note: If you would like to keep every thing integer (like the pixels in image B), take the NINT values of the optimal values of $a, b,$ and c.

- Gray codes
  - make adjacent binary only change 1 number

GRAY CODES
-- CONSTRUCTION EXAMPLES --

- Let $G_n = 0G_{n-1}, 1G_{n-1}^R$, that is concatenate the two sequence $0G_{n-1}$ and $1G_{n-1}^R$
- Construct G_2 from G_1:
  1. We saw that G_1 is: 0 , 1 and so $0G_1$ is 00, 01
  2. G_1^R is: 1, 0 and so $1G_1^R$ is: 11, 10
  3. Hence, $G_2 = 0G_1, 1G_1^R$ is : 00, 01, 11, 10
- Construct G_3 from G_2:
  1. G_2: 00, 01, 11, 10 $\Rightarrow$ $0G_2$: 000, 001, 011, 010
  2. G_2^R: 10, 11, 01, 00 $\Rightarrow$ $1G_2^R$: 110, 111, 101, 100
  3. $G_3 = 0G_2, 1G_2^R$ is : 000, 001, 011, 010, 110, 111, 101, 100
- Construct $G_4 = 0G_3, 1G_3^R$: 0000, 0001, 0011, 0010, 0101, 0111, 0101, 0100,
  1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

GRAY CODES
-- CODING OF INTEGERS--

- Once we have a Gray code G_n, we can code any integer k (i.e, a pixel value between 0 and $2^n - 1$) into the k^{th} binary string in G_n
- Example:
  - k: 0, 1, 2, 3, 4, 5, 6, 7
  - G_3: 000, 001, 011, 010, 110, 111, 101, 100

Gray_code(0)=000, Gray_code(2)=011
Gray_code(4)=110, Gray_code(7)=100

- For speed reason, it is convenient to know the Gray code $g_{n-1} \dots g_1 g_0$ of an integer k, directly from k, without having to have all G_n (of 2^n strings)
- **Method**: **input** is k, **output** is the Gray code of k, denoting it $g_{n-1} \dots g_1 g_0$
  1. Convert k to regular binary $b_{n-1} \dots b_1 b_0$, using decimal-to-binary conversion
  2. Set $g_{n-1} = b_{n-1}$
  3. For i=0 to n-2 do: $g_i = b_i$ **XOR** b_{i+1};

Recall that (0 **XOR** 0) = (1 **XOR** 1) = 0, and (1 **XOR** 0) = (0 **XOR** 1) = 1

- Gray-to-binary conversion (for decoding): to get $b_{n-1} \dots b_1 b_0$ from $g_{n-1} \dots g_1 g_0$
  - $b_{n-1} = g_{n-1}$;
  - For i=n-2 down to 0 do: $b_i = g_i$ **XOR** b_{i+1};

GRAY CODES
-- PROOF OF CORRECTNESS--

Theorem: G_n is a Gray code for all $n \geq 1$, and the last string and 1st string (of G_n) differ by only one bit.

Proof: We will prove by induction on n that every two successive strings in G_n differ by exactly one bit, and so do the 1st and last string.

- **Basis step**: $n = 1$. Prove that the theorem is true for n=1. Well, G_1: 0, 1, and so its only two successive strings 0 and 1 different by exactly one bit.
- **Induction step**:
  - Assume the theorem is true for $n-1$, i.e., every two successive strings in G_{n-1} differ by exactly one bit, and so do the 1st and last string in G_{n-1}. This is called *the induction hypothesis* (IH)
  - Prove that every two successive strings in G_n differ by exactly one bit, and so do the 1st and last strings.
  - Take two successive strings W_1 and W_2 in $G_n = 0G_{n-1}, 1G_{n-1}^R$. We have three cases:
    a. W_1 and W_2 are in $0G_{n-1}$: so $W_1 = 0V_1$ and $W_2 = 0V_2$ where V_1 *and* V_2 are two successive strings in G_{n-1}, and so, by the IH, they differ by only one bit. Therefore, since W_1 and W_2 agree in the left most bit (0), they differ in only one bit.
    b. W_1 and W_2 are in $1G_{n-1}^R$: the proof is very similar to part (a)

c. W_1 is the last string in $0G_{n-1}$, and W_2 is the 1st string in $1G_{n-1}^R$. So, $W_1 = 0V_1$ and $W_2 = 1V_2$ but also observe that $V_1 = V_2$ because the first string in G_{n-1}^R is the last string of G_{n-1}. Hence, $W_1 = 0V_1$ *and* $W_2 = 1V_1$, and so they differ by only the leftmost bit.

- One thing remains: W_1 is the first string of $0G_{n-1}$, and W_2 is the last string in $1G_{n-1}^R$. That is similar to case (c) above. Q.E.D.

ARITHMETIC CODING (AC)
-- METHOD --

Input: a binary stream $x = x_1 x_2 \dots x_n$, and a probabilistic model of x

Output: a coded bitstream

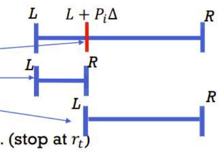
Method:

There is a patent on AC by IBM called the Q-Coder

1. Let $I = [L, R)$ where initially $L = 0, R = 1$;

Observation 1: Every subinterval is nested inside the earlier intervals

2. For i=1 to n do
   a. Let $P_i = \Pr[0/x_1 x_2 \dots x_{(i-1)}]$; // regardless of what x_i is
   b. Let $\Delta = R - L$; // length of the interval now
   c. **Split** interval I into 2 subintervals: $[L, L + P_i\Delta)$ and $[L + P_i\Delta, R)$;
   d. **Choose**: If (x_i==0), reduce I to $[L, L + P_i\Delta)$, i.e., $R := L + P_i\Delta$;
      If (x_i==1), reduce I to $[L + P_i\Delta, R)$, i.e., $L := L + P_i\Delta$;



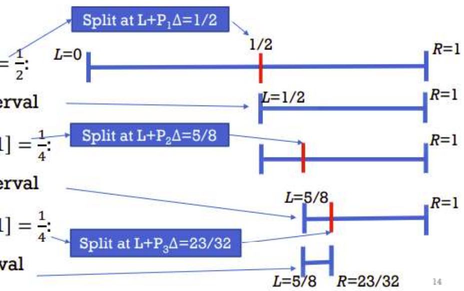
3. Let $t = \lceil -\log \Delta \rceil$, and r = $\frac{L+R}{2}$ expressed in binary as $0.r_1 r_2 \dots r_t \dots$ (stop at r_t)
4. Output:= $r_1 r_2 \dots r_t$, i.e., code the input $x_1 x_2 \dots x_n$ as $r_1 r_2 \dots r_t$

Observation 2: r is in every subinterval that was chosen. Why?

$t = \lceil -\log(\text{length of last subinterval}) \rceil$

ARITHMETIC CODING (AC)
-- EXAMPLE (1/2) --

- Binary Markov Source $\Pr[0/0] = \Pr[1/1] = \frac{3}{4}$, $\Pr[0/1] = \Pr[1/0] = \frac{1}{4}$, and $\Pr[0] = \Pr[1] = \frac{1}{2}$
- Code input $x = 110$
- $i = 1, x_1 = 1, \Delta = 1, P_1 = \Pr[0] = \frac{1}{2}$:
- Since $x_1 = 1$, choose right interval
- $i = 2, x_2 = 1, \Delta = \frac{1}{2}, P_2 = \Pr[0/1] = \frac{1}{4}$:
- Since $x_2 = 1$, choose right interval
- $i = 3, x_3 = 0, \Delta = \frac{3}{8}, P_3 = \Pr[0/1] = \frac{1}{4}$:
- Since $x_3 = 0$, choose left interval



ARITHMETIC CODING (AC)
-- EXAMPLE (2/2) --

- The final interval was: L=5/8 R=23/32
- Therefore, $L = \frac{5}{8}, R = \frac{23}{32}, \Delta = R - L = \frac{3}{32}$
- $t = \lceil -\log\Delta \rceil = \left\lceil -\log\frac{3}{32} \right\rceil = \lceil 3.4150 \rceil = 4$
- So the coded bitstream will have 4 bits
- $\frac{L+R}{2} = \frac{43}{64}$, convert to binary, we get $\frac{L+R}{2} = 0.101011$
- Take the first 4 bits after the point: 1010
- Therefore, the coded bitstream r is: **1010**
- Done with the example

> - This example led to expansion, not compression.
> - Not surprising because the input is small
> - For realistic input, there will be compression, usually CR ≈ 2

- decode-> stop at ceil(-log delta) = t
- LZ compression

LEMPEL-ZIV (LZ) COMPRESSION
--CODER METHOD--

Input: a binary stream $x = x_1x_2 \dots x_n$ // no need for a probabilistic model

Output: a coded bitstream

Method:

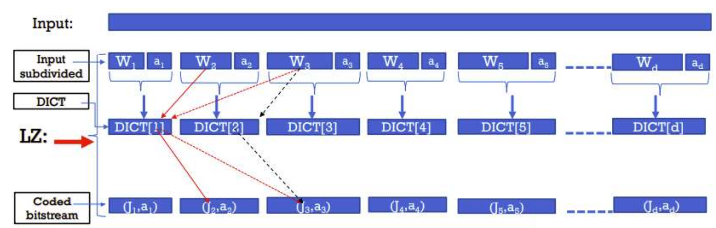
> - Assume DICT[0]= empty = – = ϵ
> - So, if W_i=empty, then j=0

1. Maintain a dictionary (DICT) of patterns seen so far
2. Set i=1, J_i=empty, W=empty, a=x_1, DICT[1]=Wa (i.e., x_1), output = x_1;
3. Set i=2;
4. While (there are still input symbols) do
   a. Read from the remaining input until the substring scanned (call it S_i) is no longer in DICT
   b. Write S_i as $S_i = W_ia_i$, where W_i is in DICT and a_i is the last symbol in S_i (the symbol after W_i)
   c. Let j be the index where W_i=DICT[j]; // j < i
   d. Let J_i=d2b(j) using $\lceil \log i \rceil$ bits // d2b is decimal to binary converter
   e. Code W_ia_i as (J_i, a_i) and append that code to the output
   f. Store W_ia_i in DICT[i], i.e., DICT[i]= W_ia_i;
   g. i++;

> Observe that:
> - x=DICT[1] DICT[2] DICT[3]…
> - Output= sequence of (J_i, a_i)'s generated

> a_i could be non-existent in the very last W_ia_i: CodeW_ia_i as (J_i,-), i.e., as J_i alone

LEMPEL-ZIV (LZ) COMPRESSION
--OVERVIEW --



> Notes:
> - Every W_i is in DICT[1 : (i-1)]
> - J_i is the index where W_i is stored in DICT[1 : (i-1)], i.e., W_i==DICT[J_i]
> - W_ia_i is stored in location i of DICT, i.e., DICT[i]:= W_ia_i
> - W_1 is non-existent, i.e., empty (we denote it as – or ϵ)
> - So J_1 is non-existent, i.e., empty (we denote it as –)

LEMPEL-ZIV (LZ) COMPRESSION

-- REMARKS --

- The dictionary is not stored/transmitted in the coded bitstream
- Rather, it is used by the coder to carry out the coding, and then discarded
- As will be seen, the decoder will be able to reconstruct the dictionary from the coded bitstream alone
- No probabilities were needed or used
- The LZ bitrate is asymptotically optimal (i.e., approaches the entropy of the source with memory) **without** the need to know or compute the underlying probability model of the input data.
- The proof will not be provided. See paper if interested

LEMPEL-ZIV (LZ) COMPRESSION

-- EXAMPLE--

- Input: x=00101001000100101001110101

Notation: When we find $S_i = W_i a_i$ in x, we color W in green and a in red. DICT[i]=$W_i a_i$

i	⌈log i⌉	J_i=$(j)_2$	W_i	a_i	DICT[i]	
1	0	empty	empty	0	0	00101001000100101001110101
2	1	$(1)_2$=1	0	1	01	00101001000100101001110101
3	2	$(10)_2$=2	01	0	010	00101001000100101001110101
4	2	$(11)_2$=3	010	0	0100	00101001000100101001110101
5	3	$(100)_2$=4	0100	1	01001	00101001000100101001110101
6	3	$(101)_2$=5	01001	1	010011	00101001000100101001110101
7	3	$(011)_2$=3	010	1	0101	00101001000100101001110101

Output: Sequence of (J_i, a_i)'s for i=1,2,… . Therefore:
Output: (-,0) (1,1) (10,0) (11,0) (100,1) (101,1) (011, 1). Finally:
Output: 0111001101001101101111

Recall:
DICT[0]= empty = – = ϵ

LZ DECODER

-- METHOD --

Input: a coded bitstream $y = y_1 y_2 \ldots$; **Output**: the original data x

Method:

1. i=1, W_1=empty, $a_1 = y_1$, DICT[1]=$W_1 a_1$ (i.e., y_1), $x = W_1 a_1$ (i.e., y_1);
2. i=2;
3. While (the coded bitstream is not fully scanned) do
   a. j=decimal(the next ⌈log i⌉ bits from y);
   b. W_i=DICT[j];
   c. a_i=the next symbol from y;
   d. append $W_i a_i$ to the right of x;
   e. DICT[i]=$W_i a_i$
   f. i++;

   // if a_i=empty, at end of decoding, simply append W_i to the right of x

Observe that:
- x=DICT[1] DICT[2] DICT[3]…
- Which is what we want based on what we observed in the LZ coder

LZ DECODER

-- EXAMPLE--

Decoding y=01110 0110100110110111 from the previous example

i	⌈log i⌉	j=next ⌈log i⌉ bits of y	W_i=DICT[j]	a_i	DICT[i] =$W_i a_i$	x= (previous(x))($W_i a_i$)
1	0	empty	empty	0	0	0
2	1	$(1)_2$=1	0	1	01	001
3	2	$(10)_2$=2	01	0	010	001010
4	2	$(11)_2$=3	010	0	0100	0010100100
5	3	$(100)_2$=4	0100	1	01001	001010010001001
6	3	$(101)_2$=5	01001	1	010011	001010010001001010011
7	3	$(011)_2$=3	010	1	0101	00101001000100101001110101

Decoded data x = 00101001000100101001110101

COMPRESSION PERFORMANCE METRICS
-- SIZE REDUCTION MEASURES --

- **Bit Rate** (or **bitrate or BR**): Average number of bits per original data element, after compression
  - In text compression: BR= avg. number of bits per character $= \frac{\#bits\ in\ the\ coded\ bitstream}{\#characters\ in\ the\ original\ text}$
  - In images/videos: BR= average number of bits per pixel $= \frac{\#bits\ in\ the\ coded\ bitstream}{\#pixels\ in\ the\ image\ or\ video}$
  - In audio: BR= avg. num. of bits per sound sample$= \frac{\#bits\ in\ the\ coded\ bitstream}{\#samples\ in\ the\ audio\ file}$
- **Compression Ratio** (CR) $= \frac{size\ of\ the\ uncompressed\ data\ (in\ bits)}{size\ of\ the\ coded\ bitstream\ (in\ bits)} = \frac{|I|}{|b|}$
- **Exercise**: A file has 1000 characters, and takes 8K bits before compression. After compression, its size became 4K bits. What is the bitrate? What the CR?

BASIC DEFINITIONS IN COMPRESSION/CODING
-- QUALITY MEASURES: SNR --

- Let I be an original signal (e.g., an image), and $\hat{I}$ be its lossily reconstructed counterpart
- **Signal-to-Noise Ratio** (**SNR**) in the case of lossy compression:
  - It is a measure of the quality of the reconstructed (decompressed/decoded) data
  - More accurately, it is the **fidelity** of the decoded data w.r.t. the original
  - Mathematically: **SNR** $= 10\log_{10}\left(\frac{\|I\|^2}{\|I-\hat{I}\|^2}\right) = 20\log_{10}\left(\frac{\|I\|}{\|I-\hat{I}\|}\right)$

    where for any vector/matrix/set of number $E = \{x_1, x_2, \ldots, x_N\}$,

    $$\|E\|^2 = x_1^2 + x_2^2 + \cdots + x_N^2$$
- The unit of SNR is "***decibel***" (or **dB** for short)
- So, if SNR = 23, we say the SNR is 23 dB

Question: Does it make sense to compute SNR for lossless compression? Why or why not?

BASIC DEFINITIONS IN COMPRESSION/CODING
-- QUALITY MEASURES: MEAN-SQUARE ERROR --

- Mean-Square Error (MSE): $\text{MSE} = \frac{1}{N}\left\|I - \hat{I}\right\|^2$
- Relative Mean-Square Error (RMSE): $\text{RMSE} = \frac{\|I-\hat{I}\|^2}{\|I\|^2}$
- **Therefore, SNR** $= -10\log_{10}\text{RMSE}$
- **Observations:**
  - The smaller the RMSE (or the MSE), the higher the SNR
  - Therefore, the higher the SNR, the better the quality of the reconstructed data
- **Exercise**: Prove that if RMSE is decreased by a **factor** of 10, then SNR increases by 10 decibels.
- That justifies the multiplicative factor in the definition of the SNR.

- Therefore, Entropy of S, denoted $H(S)$, is:

$$H(S) = -(p_1\log p_1 + p_2\log p_2 + \cdots + p_n\log p_n) = -\sum_{i=1}^{n} p_i \log p_i$$

INFORMATION THEORY (ADJOINT SOURCES)

- Adjoint Source of Order N of a source with memory:

Source data S: 100100110100110110100110100000100010000110111011

Split into blocks of N bits (N=3)

Adjoint source: 100 100 110 100 110 110 100 110 100 000 100 010 000 110 111 011

$P(100) = \frac{6}{16}, P(110) = \frac{5}{16}, P(000) = \frac{2}{16}$

$P(010) = P(011) = P(111) = \frac{1}{16}$

  - Split the source blocks of N symbols
  - Treat each block A as a macrosymbol, and compute its probability P_A
  - Treat the sequence of macrosymbols as a memoryless source. That is, treat the macrosymbol) as independent of one another
  - The entropy of this adjoint source is: $H_N(S) = -\sum_A P_A \log P_A$

$H_3(S) = 2.18$

- **Theorem** (Shannon): $\frac{H_N(S)}{N} \to H(S)\ as\ N \to \infty$

As N gets larger, $\frac{H_N(S)}{N}$ gets smaller until $\approx H(S)$

- This implies that for any source S with memory, if we divide it into blocks of large enough size N and then block-code it without taking advantage of inter-block correlation, then we still approximate the performance of the best coder S.
- **Therefore, block coding is a good way to go when we have correlation**

HUFFMAN CODING
-- THE CODING ALGORITHM--

Input: alphabet $\{a_1, a_2, \ldots, a_n\}$ and symbol probabilities $\{p_1, p_2, \ldots, p_n\}$

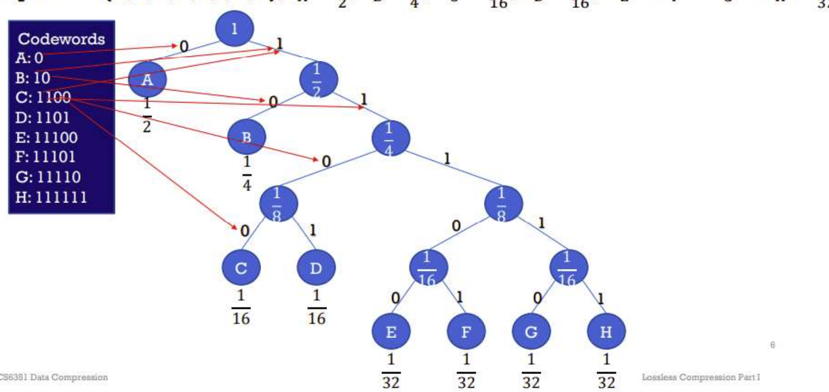
Output: the codewords of the alphabet symbols

Method: (a Greedy method for creating a Huffman tree as follows)

1. Create a node for each symbol a_i // these nodes will be the leaves
2. **While** (there are two or more uncombined nodes) **do**
   - Select 2 uncombined nodes a and b of <u>minimum</u> probabilities
   - Create a new node c of prob $P_a + P_b$, and make a and b children of c
3. Label the tree edges: left edges with 0, right edges with 1
4. The codeword of each alphabet symbol a_i (a leaf) is the binary string that labels the path from the root down to leaf a_i

HUFFMAN CODING
-- ILLUSTRATION OF THE CODING ALGORITHM --

Alphabet={A,B,C,D,E,F,G,H}, $P_A = \frac{1}{2}$, $P_B = \frac{1}{4}$, $P_C = \frac{1}{16}$, $P_D = \frac{1}{16}$, $P_E = P_F = P_G = P_H = \frac{1}{32}$



HUFFMAN CODING
-- CODING PERFORMANCE --

- In lossless compression of memoryless sources
  - If the coder works by computing a codeword for each alphabet symbol
  - Then, we can compute a **coder bitrate**, independent of any actual input data
- Notation:
  - For any binary string s, denote by **|s|** the number of bits in s
  - Let **codeword(a_i)** denote the codeword for symbol a_i
- Coder bitrate: $BR = \sum_{i=1}^{n} p_i \,|\text{codeword}(a_i)|$
- Source Entropy: $H = -\sum_{i=1}^{n} p_i \log p_i$

Example: the Huffman coder just presented

- The codewords and the probabilities are

Codewords	Length	Probabilities
A: 0	1	1/2
B: 10	2	1/4
C: 1100	4	1/16
D: 1101	4	1/16
E: 11100	5	1/32
F: 11101	5	1/32
G: 11110	5	1/32
H: 11111	5	1/32

- BR=$1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 4 \times \frac{1}{16} + 4 \times \frac{1}{16} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} + 5 \times \frac{1}{32} = \frac{69}{32} = 2.125\ bits/symbol$
- Entropy: H=$-\left(\frac{1}{2}\log\frac{1}{2} + \frac{1}{4}\log\frac{1}{4} + \frac{1}{16}\log\frac{1}{16} + \frac{1}{16}\log\frac{1}{16} + \right.$

HUFFMAN CODING
-- OBSERVATIONS (1/2) --

- Observation 1 (about the example):
  - **BR=H**, i.e., the coder bitrate achieved the entropy, the best possible
  - Does that mean Huffman coding always achieves the entropy?
  - No. See the theorem next
- **Theorem**: If all the probabilities (in a memoryless source) are powers of ½, then Huffman achieves BR=H. The further away the probabilities are from powers of ½, the further away BR is from entropy H (i.e., BR>H).
- We will not prove that theorem, but it is important that you keep it in mind

RUN-LENGTH ENCODING (RLE)
-- RUN-SPLITTING EXAMPLE --

- I: aaaaaaaaaaa bbbbbbbbbb dddddddd bbbbbb cccccc aaaaa
- Intermediate code: (a,11) (b,10) (d,8) (b,6) (c,6) (a,5)
- Assume M=3 (so, max run-length representable is $7=2^3-1$)
- (a,11) is split into (a,7) (a,4), which in binary is (a,111) (a, 100)
- (b,10) is splits into (b,7) (b,3), which in binary is (b,111) (b, 011)
- (d,8) is splits into (d,7) (d,1), which in binary is (d,111) (d, 001)
- (b,6) (c,6) (a,5) need not be split: in binary (b,110) (c,110) (a,101)
- So, the next intermediate code becomes:

(a,111) (a, 100) (b,111) (b, 011) (d,111) (d, 001) (b,110) (c,110) (a,101)

RLE DECODING

- It is an alternating sequence of decodings:
  - Huffman decoding, bin-2-dec of next M bits, Huffman decoding, bin-2-dec of the next M bits, …
- This decodes to an intermediate representation of (a,L) pairs
- Finally, replace each (a,L) by aaaa…a, where the run is of length L
- Example: Codewords: a:0, b:10, c:110, d:111 M=3
  - Bitstream= 0111010010111100111111111100110110110110100101
  - Alternating decoding: (a,7) (a, 4) (b,7) (b, 3) (d,7) (d, 1) (b,6) (c,6) (a,5)
  - Final decoding: aaaaaaa aaaa bbbbbbb bbb ddddddd d bbbbbb cccccc aaaaa
  - Without spaces: aaaaaaaaaaabbbbbbbbbbddddddddbbbbbbccccccaaaaa
  - You can verify that the decoded data is identical to the original data

GOLOMB CODING
-- PRELIMINARIES (2/2) --

- Golomb coding has a parameter m
  - m is a positive integer set by the algorithm implementer or by the user
  - the optimal value of m = the nearest power of 2 to $p \times \frac{\text{Ln } 2}{1-p}$ (proof is later)
  - where p is the probability of the more probable bit in the input stream
  - p is easily computable: count the number of 0's (say N_0) and the number of 1's (say N_1) in the input binary stream, then $p = \max(\frac{N_0}{N_0+N_1}, \frac{N_1}{N_0+N_1})$
  - let's assume that the more probable bit (MPB) is 0
  - If the MPB is 1, then each run is of the form 1^i0
- Note: $\text{Ln } 2 = 0.6931$

GOLOMB CODING
-- METHOD: ASSUME 0 IS THE MPB --

1. Break the input into runs of the form 0^i1
2. Code each Golomb run 0^i1 as 1^q0y where
   - Divide i by m, integer division, we get quotient q and a remainder r, i.e., $i = qm + r$
   - y is the binary representation of r, using $\log m$ bits: $y = (r)_2$.

   Code of 0^i1: → $1^q\underbrace{0}_{\text{Separator bit}}y$

3. The final coded bitstream is: **MPB $code_1$ $code_2$ … $code_n$ tail?** where
   - MPB: the (1-bit) value of the more probable bit
   - $code_j$: the code of the j-th Golomb run
   - tail?: a single bit that is 1 if the last run has a tail; it is 0 if the last run has no tail

GOLOMB CODING
-- EXAMPLE--

- Input stream: $x = 0^9 1 0^{15} 1 1$
- MPB=0 since 0 occurs 24 times, while 1 occurs 3 times;
- $p = \frac{24}{27}, p \times \frac{\text{Ln } 2}{1-p} = 8 \times 0.6931 = 5.54$, the closest 2-power to 5.54 is 4, so $m = 4$, **log m=2**;
- The Golomb runs of $x = 0^9 1 0^{15} 1 1$ are $0^9 1, 0^{15} 1$, and $1 = 0^0 1$ Code 0^i1 as 1^q0y
- Code $0^9 1$: $9 = 2 \times 4 + 1$, so $q = 2$ and $r = 1 = (01)_2$, thus code($0^{10}1$)=1^2 0 01
- Code $0^{15} 1$: $15 = 3 \times 4 + 3$, so $q = 3$ and $r = 3 = (11)_2$, thus code($0^{15}1$)=1^3 0 11
- Code $0^0 1$: $0 = 0 \times 4 + 0$, so $q = 0$ and $r = 0 = (00)_2$, thus code($0^0 1$)=1^0 0 00 = 000
- tail? = 1 because the last Golomb run has a tail
- The code of x is: **0** $1^2$001 $1^3$011 000 **1**, i.e., 0110011110110001
- CR=$\frac{27}{16} = 1.69$, BR $= \frac{16}{27} = 0.59$ bits/bit

Input: a coded bistream (& parameter m); **Output**: the original data

Method:

1. Grab first bit as the MPB, and the last bit of the coded bitstream as the tail
2. Set k=2 // index of the bits in the coded bitstream
3. Scan the bitstream rightward from position k looking for successive 1's, keeping a count q, and incrementing k along the way
4. When a 0 is met, read the next $\log m$ bits as y, set k=k+ $\log m$
5. Set r=b2d(y) // binary to decimal conversion
6. Compute $i = q \times m + r$
7. Append 0^i1 to the output (if MPB==0), else append 1^i0 to the output
8. Repeat from 3 until the coded bitstream is exhausted
9. If the last bit (tail?) is 0, strip the final bit from the output

GOLOMB DECODING
-- EXAMPLE --

- Example: coded bitstream 0110011110110001, and $m = 4$ ($\log m = 2$)
- MPB=0, tail=1;
- k=2, scan the first two 1's till 0, then read 2 bits: 0110011110110001
- So q=2, y=$(01)_2$, r=b2d(01)=1, $i = q \times m + r = 2 \times 4 + 1 = 9$, k=7
- Append $0^9 1$ to output: output=$0^9 1$
- From k=7 scan for 1's till 0: three 1's, 0 and the next two bits 11: 0110011110110001
- So q=3, y=$(11)_2$, r=b2d(11)=3, $i = q \times m + r = 3 \times 4 + 3 = 15$, k=13
- Append $0^{15}1$ to output: output=$0^9 1 0^{15} 1$
- Look for 1's from position k=13; none found, so q=0; skip 0 and read the next 2 bits, y=00, so r=0; 0110011110110001; so i=0x4+0=0; append $0^0 1$: output=$0^9 1 0^{15} 1 1$
- Now we reached the last bit, tail=1, so we keep the tail. **Final output: $0^9 1 0^{15} 1 1$**

DIFFERENTIAL GOLOMB
-- CODING METHOD --

Input: $x = x_1x_2 \dots x_n$ (& the parameter m)

Output: coded bitstream

Method:

1. Transform x to $z = z_1z_2 \dots z_n$ where $z_i = x_i - x_{i-1} \forall i > 1$, and $z_1 = x_1$
2. Delete the alternating negatives of the tails, we get $z' = z'_1z'_2 \dots z'_n$
3. Code z' using Golomb coding

DIFFERENTIAL GOLOMB
-- DECODER METHOD --

Input: Coded bitstream (& the parameter m)

Output: Original data

Method:

1. Golomb-decode the coded bitstream into $z' = z'_1z'_2 \dots z'_n$
2. Keep the first Golomb run's tail at 1
3. Alternate the signs of the remaining tails in z', getting $z = z_1z_2 \dots z_n$
4. Set $x_1 = z_1$, and for $i = 2$ to n do: $x_i = x_{i-1} + z_i$
5. Set output=$x_1x_2 \dots x_n$