CS 6351 DATA COMPRESSION

THIS LECTURE:
LOSSLESS COMPRESSION PART III

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe and apply Differential Pulse-code Modulation (DPCM) to losslessly compress 1D and 2D input data streams
- Optimize the parameters of DPCM to fit the method to the input data
- Connect DPCM to the broader category of predictive coders
- Explain and apply Bitplane Coding of grayscale and color images
- Explicate the limitations and drawbacks of regular binary representations of decimals when it comes to spatial redundancy
- Derive and apply Gray codes, which are better than regular binary representations of decimals in their ability to preserve spatial redundancy when moving to the bit level
- Appreciate and prove the limitations of lossless compression
- Relate the lossless compression techniques covered so far to where they're used in the standards, graphic file formats, and OS utilities, for lossless data compression

OUTLINE

- Differential Pulse-code Modulation (DPCM)
- Bitplane Coding
  - Gray codes
- Limitations Of Lossless Compression
- Lossless Compression in Standards, Graphic File Formats, and Utilities
- Begin a tutorial of Matlab/Octave

DIFFERENTIAL PULSE-CODE MODULATION (DPCM)
-- MOTIVATION --

- So far, all the redundancy in the data was explicit
  - Either immediately visible like successive symbols being equal (as we saw in RLE/Golomb examples)
  - Or certain symbols/blocks/substrings occur more often than others, as we saw in Huffman and LZ examples
  - Or in the form of correlations/dependencies between symbols, as we saw in Arithmetic Coding examples
- But in some streams, the redundancy is implicit in some latent patterns
  - Take this extreme example: Suppose the data is: 1, 2, 3, 4, … , (n-1), n
  - None of the above types of redundancy/correlation is apparent, but there is a "loud" regularity in the data begging to be exploited!

DPCM
-- PRELIMINARIES (1/2) --

- But in some streams, the redundancy is implicit in some latent patterns
  - Take this extreme example: Suppose the data is: 1, 2, 3, 4, … , (n-1), n
  - None of the above types of redundancy/correlation is apparent, but there is a "loud" regularity in the data begging to be exploited!
  - If we apply any of the lossless compression techniques we have covered, none will recognize this regularity and none will be able to truly compress this stream
- DPCM works by computing differences (also called residuals) between nearby values, turning the regularity/implicit-redundancy into explicit redundancy
- Example: call the input 1,2,…,n as $x_1, x_2, \ldots, x_n$,
  - Let $e_1 = x_1, e_i = x_i - x_{i-1}\ \forall i \geq 2$. Then the data becomes: 1, 1, 1, …, 1
  - The resulting data has obvious, explicit redundancy, and can be easily compressed

DPCM
-- PRELIMINARIES (2/2) --

- Of course, real-world data is rarely so regular, but the same idea still works
- For example, if the input stream x is: 1, 1, 2, 2, 1, 2, 3, 3, 4, 4,3, 3, 4, 5, …
  - And you compute the residuals e: $1, 0, 1, 0, -1, 1, 2, 0, 1, 0, -1, 0, 1, 1$
  - Although the original data can cover a wide range with barely any explicit redundancy, the residuals show that certain values occur a lot more often, such as 0, 1, -1, 2
- Pixels in images show similar patterns, where local variations are minor but build up over long distances
- Also, audio signals show similar patterns, where local variations are minor but build up over long periods of time

DPCM
-- MAIN METHOD: FOR 1D DATA--

- 1D data:
  - linear streams of data $x: x_1, x_2, \ldots, x_n$ as opposed to images/videos)
- DPMC for 1D data takes one parameter a (typically $a \approx 1$, like 1 or 0.99)
- **Coder Method**:
  1. Compute the residuals: $e_1 = x_1, e_i = x_i - ax_{i-1}\ \forall i \geq 2$. Notice the multiplication by the parameter a
  2. Code the residuals with Huffman (or any other suitable lossless coder)
- **Decoder Method**:
  1. Decode the coded bitstream with the right decoder, getting the e_i's
  2. Get x: $x_1 = e_1, x_i = e_i + ax_{i-1}\ \forall i \geq 2$

For step 2 of the decoder to work, the residuals in the coder must be differences between current input and **previously processed** input

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 $E(a) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (x_i - a x_{i-1})^2$, 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
  - $E'(a) = -2 \sum_{i=1}^{n} x_{i-1}(x_i - a x_{i-1}) = -2[\sum_{i=1}^{n} x_{i-1} x_i - a \sum_{i=1}^{n} x_{i-1}^2]$,
  - Setting $E'(a) = 0$, and solving, we get $\boldsymbol{a = \frac{\sum_{i=1}^{n} x_{i-1} x_i}{\sum_{i=1}^{n} x_{i-1}^2}}$
  - If x is integers and you want to keep the residues integer, take $\boldsymbol{a = \text{NINT}(\frac{\sum_{i=1}^{n} x_{i-1} x_i}{\sum_{i=1}^{n} x_{i-1}^2})}$

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

DPCM
-- MAIN METHOD (FOR 1D DATA): LONGER WINDOWS --

- The DMPC method just presented assumes that the data dependence on history is limited to the "most recent history"
- The dependence can be deeper, like every x_i depends on both x_{i-1} & x_{i-2} (or more)
- DPCM can be generalized to capture whatever dependence depth
  - Depth=2: DPCM takes two parameters a_1, a_2, and the residues are $e_i = x_i - a_1 x_{i-1} - a_2 x_{i-2}$
  - Depth=d: DPCM takes d parameters $a_1, \ldots, a_d$, and the residues are $e_i = x_i - \sum_{k=1}^{d} a_k x_{i-k}$
- Example of multi-parameter values:

Any x_i out of bound (i.e., $i < 1$) is taken to be 0

  - Depth=2: $a_1 = a_2 = \frac{1}{2}$, i.e., e_i is the difference between x_i and the average of the 2 previous values
  - For general depth d, you can take $a_k = \frac{1}{d}$, or have the parameters like weights that add up to ≈ 1
  - Alternatively, you can optimize the parameters to fit your specific input, by computing the $E(a_1, a_2, \ldots, a_d) = \sum_{i=1}^{n} e_i^2$, computing its partial derivatives with respect to each a_k, setting those to 0, and solving a system of d linear equations in d unknowns (where the unknowns are $a_1, \ldots, a_d$)

DPCM
-- EXERCISE --

- **Exercise**: Carry out the optimization of the parameters for depth=2, and derive formulas for optimal a_1 and a_2.

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

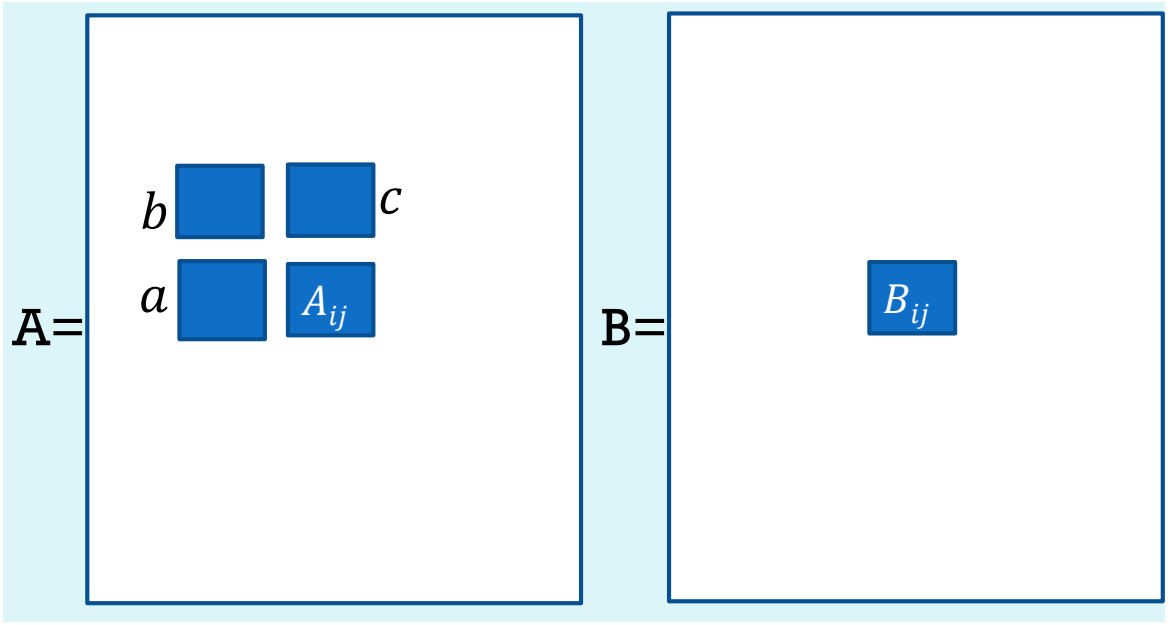
- DPCM for 2D data (i.e., an image A)
- DPMC for 2D data takes <u>three</u> parameters (a, b, c)

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 $a, b, \text{and } c$, set them to 0, and solve the system of 3 linear equations with three unknowns (the unknowns are $a, b, \text{and } c$)
- **Exercise**: Carry out the above step, and derive formulas for optimal $a, b, \text{and } c$.
- 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 $a, b, \text{and } c$.

DPCM
-- GENERALIZATION AND BROADER PERSPECTIVE --

- DPCM fits in a category of compression techniques that are based on ***predictive*** models
- In those techniques, the designer builds and uses a predictive model that can predict (i.e., estimate/approximate) the current value of the input stream based on previous data values already seen (and processed) in the input stream
- When coding,
  - the predictive model is used to predict the current value of the input stream,
  - then the residue (or error) is computed between the actual value and the predicted value of the current element
  - and finally those residuals are losslessly coded using some other coder such as Huffman, RLE, AC, etc.

DPCM

-- BROADER PERSPECTIVE: EXERCISE & PROJECTS --

- **Exercise**: what are the predictive model(s) in DPCM, both in 1D (of any depth) and 2D input?
- **Potential term project**: Explore other predictive models for lossless compression
  - Review the literature on this subject
  - Summarize the techniques, and evaluate their performance , comparing and contrasting them
  - Possibly develop new predictive model(s) and evaluate their performance (on some test data)

BITPLANE CODING
-- PRELIMINARIES --

- We have developed some very powerful lossless coders for binary streams, such as RLE, (diff) Golomb coding, arithmetic coding, LZ, and block Huffman coding
- Can we use those on grayscale/color images?
- Yes, if we "binarize" the images, i.e., split each image into a bunch of binary images
- How: using bitplanes?
  - Let I be an image where every pixel value is n-bit long
  - Express every pixel in binary using n bits
  - Form out of image I n binary matrices (called *bitplanes*), where the i^{th} matrix consists of the i^{th} bits of the pixels of image I

BITPLANE CODING

-- ILLUSTRATION OF THE BITPLANES --

Image I:

5	5	6	5	5	4
5	4	6	5	4	4
4	4	5	4	5	5
3	3	4	3	4	4
2	3	4	3	2	3
1	2	3	2	1	2

Decimal to binary

assume 3-bit pixels

IB:

101	101	110	101	101	100
101	100	110	101	100	100
100	100	101	100	101	101
011	011	100	011	100	100
010	011	100	011	010	011
001	010	011	010	001	010

- Express every pixel in binary
- Denote the binary representation of each pixel as $a_{n-1}a_{n-2}\dots a_2a_1a_0$
- The i^{th} plane is a binary image derived from image I by replacing each pixel by the i^{th} bit a_i
- Denote the i^{th} plane of I by IB_i

IB_2:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

IB_1:

0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
1	1	0	1	0	0
1	1	0	1	1	1
0	1	1	1	0	1

IB_0:

1	1	0	1	1	0
1	0	0	1	0	0
0	0	1	0	1	1
1	1	0	1	0	0
0	1	0	1	0	1
1	0	1	0	1	0

DRAWBACK OF CONVENTIONAL BINARY REPRESENTATION OF DECIMAL NUMBERS

- In natural images, often nearby pixels are identical or near-identical.
- Pixel values that differ by 1 or 2, in decimal, can differ in many bit positions in binary
- For example, take two neighboring pixels of values 7 and 8, which are rather similar
  - Express those values in 4-bit binary: 7 is 0111, and 8 is 1000
  - The binary strings differ in every bit position!
  - In other terms, in each bit plane, the pixels in the same relative positions differ, thus reducing the potential for compression
- So we have a wish/need/goal: **A different binary representation where similar decimal values differ in only a small number of bits**

GRAY CODES
-- DEFINITION --

- Goal: **A binary representation where similar decimal values differ in only a small number of bits**
- Solution: Gray codes
- **Definition**: a ***Gray code*** of n bits is a sequence of up to 2^n binary strings of n bits each, where every two successive strings **differ by only one bit**
- **Notation**: Let's denote by G_n the Gray code of n-bit strings
- Examples:
  - G_1 is: 0 , 1 //the two successive one-bit strings differ by only 1 bit
  - G_2 is: 00, 01, 11, 10 // every two successive strings differ by only 1 bit
- We will give a constructive (i.e., algorithmic) method for constructing Gray codes, next

GRAY CODES
-- CONSTRUCTION --

Method (recursive/inductive):

The method constructs G_n given G_{n-1}, as follows:

1. Takes the strings of G_{n-1}, and append a 0 to the left of each string; denote this new sequence by $0G_{n-1}$
2. Take a 2nd copy of G_{n-1}, but list the strings in backward (reverse) order (don't reverse the bits in each string, but rather place the first string last, the last string first, and so on); denote this new sequence by G_{n-1}^R
3. Append a 1 to the left of every string of G_{n-1}^R, and denote the new sequence by $1G_{n-1}^R$
4. Let $G_n = 0G_{n-1}, 1G_{n-1}^R$, that is, concatenate the two sequence $0G_{n-1}$ and $1G_{n-1}^R$

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
-- 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)

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 (continued):

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.

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 CONVERSION-CORRECTNESS--

- **Exercise**: Prove that the binary-to-Graycode conversion method of the last slide is correct, i.e., it computes the correct gray codes. (**Hint**: by induction on n)
- **Exercise**: Prove that the Graycode-to-binary conversion method of the last slide is correct, i.e., it computes the correct binary equivalent of each Gray code. (**Hint**: by induction on n)

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (1/9)

- Let I be the following 3-bit image:

5	5	6	5	5	4
5	4	6	5	4	4
4	4	5	4	5	5
3	3	4	3	4	4
2	3	4	3	2	3
1	2	3	2	1	2

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (2/9)

1. Represent the pixels of I in binary (middle as IB) and in Gray code (right as IG):

I:

5	5	6	5	5	4
5	4	6	5	4	4
4	4	5	4	5	5
3	3	4	3	4	4
2	3	4	3	2	3
1	2	3	2	1	2

IB:

101	101	110	101	101	100
101	100	110	101	100	100
100	100	101	100	101	101
011	011	100	011	100	100
010	011	100	011	010	011
001	010	011	010	001	010

IG:

111	111	101	111	111	110
111	110	101	111	110	110
110	110	111	110	111	111
010	010	110	010	110	110
011	010	110	010	011	010
001	011	010	011	001	011

Integer k: 0, 1, 2, 3, 4, 5, 6, 7
Graycode G_3: 000, 001, 011, 010, 110, 111, 101, 100

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (3/9)

2. Split IB and IG into bitplanes

IB:

101	101	110	101	101	100
101	100	110	101	100	100
100	100	101	100	101	101
011	011	100	011	100	100
010	011	100	011	010	011
001	010	011	010	001	010

IG:

111	111	101	111	111	110
111	110	101	111	110	110
110	110	111	110	111	111
010	010	110	010	110	110
011	010	110	010	011	010
001	011	010	011	001	011

IB_2:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

IB_1:

0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
1	1	0	1	0	0
1	1	0	1	1	1
0	1	1	1	0	1

IB_0:

1	1	0	1	1	0
1	0	0	1	0	0
0	0	1	0	1	1
1	1	0	1	0	0
0	1	0	1	0	1
1	0	1	0	1	0

IG_2:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

IG_1:

1	1	0	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	1	1	1	0	1

IG_0:

1	1	1	1	1	0
1	0	1	1	0	0
0	0	1	0	1	1
0	0	0	0	0	0
1	0	0	0	1	0
1	1	0	1	1	1

BITPLANE CODING
EXAMPLE AND BINARY-GRAY COMPARISON (4/9)

3. Flatten each binary matrix row-wise into a binary 1D stream
4. Apply simple RLE on each stream (since they're binary, the runs alternate)
5. Represent those runs by their lengths, and then Huffman-code the lengths

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (5/9)

- Flatten the IB planes row-wise and get the run-lengths:

IB_2:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

$1^{18}0^2101^20^210^9$

Get run-lengths:

18, 2, 1, 1, 2, 2, 1, 9

IB_0:

1	1	0	1	1	0
1	0	0	1	0	0
0	0	1	0	1	1
1	1	0	1	0	0
0	1	0	1	0	1
1	0	1	0	1	0

$1^201^2010^210^4101^4010^310101^201010$

Get run-lengths:

2, 1, 2, 1, 1, 2, 1, 4, 1, 1, 4, 1, 1, 3, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1

IB_1:

0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
1	1	0	1	0	0
1	1	0	1	1	1
0	1	1	1	0	1

$0^210^510^91^2010^21^201^301^301$

Get run-lengths:

2, 1, 5, 1, 9, 2, 1, 1, 2, 2, 1, 3, 1, 3, 1, 1

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (6/9)

- Flatten the IG planes row-wise and get the run-lengths:

IG_2:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	1	0	1	1
0	0	1	0	0	0
0	0	0	0	0	0

$1^{18}0^2101^20^210^9$

Get run-lengths:

18, 2, 1, 1, 2, 2, 1, 9

IG_0:

1	1	1	1	1	0
1	0	1	1	0	0
0	0	1	0	1	1
0	0	0	0	0	0
1	0	0	0	1	0
1	1	0	1	1	1

$1^501^20^41^20^61^31^20^13$

Get run-lengths:

5, 1, 1, 1, 2, 4, 1, 1, 2, 6, 1, 3, 1, 1, 2, 1, 3

IG_1:

1	1	0	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	1	1	1	0	1

$1^201^501^{21}01^301$

Get run-lengths:

2, 2, 5, 1, 21, 1, 3, 1, 1

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (7/9)

- The actual run-lengths in the IB bitplanes are:
  - IB_2: 18, 2, 1, 1, 2, 2, 1, 9;
  - IB_1: 2, 1, 5, 1, 9, 2, 1, 1, 2, 2, 1, 3, 1, 3, 1, 1
  - IB_0: 2, 1, 2, 1, 1, 2, 1, 4, 1, 1, 4, 1, 1, 3, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1
- The "alphabet" of those run lengths, and the probs:

Alphabet:	1	2	3	4	5	9	18
Frequencies:	28	11	3	2	1	2	1
Probabilities:	$\frac{28}{48}$	$\frac{11}{48}$	$\frac{3}{48}$	$\frac{2}{48}$	$\frac{1}{48}$	$\frac{2}{48}$	$\frac{1}{48}$

Apply Huffman coding on this alphabet, we get

Sym-bols	Huffman Codewords	Length
1	0	1
2	10	2
3	1110	4
4	1100	4
5	11110	5
9	1101	4
18	11111	5

- Size of coded bitstream: $28 \times 1 + 11 \times 2 + 3 \times 4 + 2 \times 4 + 1 \times 5 + 2 \times 4 + 1 \times 5 = 88$

$$\text{Bitrate} = \frac{88}{\text{number of pixels}} = \frac{88}{36} = 2.44 \text{ bits per pixel}$$

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (8/9)

- The actual run-lengths in the IG bitplanes are:
  - IG2: 18, 2, 1, 1, 2, 2, 1, 9;
  - IG1: 2, 1, 5, 1, 21, 1, 3, 1, 1
  - IG0: 5, 1, 1, 1, 2, 4, 1, 1, 2, 6, 1, 3, 1, 1, 2, 1, 3
- The "alphabet" of those run lengths, and the probs:

Alphabet:	1	2	3	4	5	6	9	18	21
Frequencies:	17	7	3	1	2	1	1	1	1
Probabilities:	$\frac{17}{34}$	$\frac{7}{34}$	$\frac{3}{34}$	$\frac{1}{34}$	$\frac{2}{34}$	$\frac{1}{34}$	$\frac{1}{34}$	$\frac{1}{34}$	$\frac{1}{34}$

Apply Huffman coding on this alphabet, we get

Sym-bols	Huffman Codewords	Length
1	0	1
2	10	2
3	1100	4
4	11010	5
5	11011	5
6	11100	5
9	11101	5
18	11110	5
21	11111	5

- Size of coded bitstream: $17 \times 1 + 7 \times 2 + 3 \times 4 + 1 \times 5 + 2 \times 5 + 1 \times 5 + 1 \times 5 + 1 \times 5 + 1 \times 5 = 78$.

$$\text{Bitrate} = \frac{78}{\text{number of pixels}} = \frac{78}{36} = 2.17 \text{ bits per pixel} < 2.44 \text{ bpp}$$

BITPLANE CODING

EXAMPLE AND BINARY-GRAY COMPARISON (9/9)

- We just saw that
  - the BR with Gray code (2.17 bpp) is < the BR with regular binary (2.44bpp)
- This illustrates that Graycodes lead to better compression
- The savings don't seem impressive, but for large images, that saving could be larger and amounts to a significant reduction in the total size
- This is even more impressive when you realize that the compression ratio in lossless compression is usually around 2 or so

LIMITATIONS OF LOSSLESS COMPRESSION

- For most lossless compression applications, especially generic applications (like compressing text files and images)
  - Compression ratios are about 2:1 (i.e., CR=2)
- In special applications where there is more redundancy than average
  - special-purpose compression techniques/tweeks have been developed and tailored to the characteristics of those data streams,
  - achieving higher compression ratios of about 5, sometimes reportedly up to 8
- Still, for heavy-duty applications, involving images, videos, and/or sound, such compression ratios are inadequate
- Therefore, there is a huge need for lossy compression
- So, for the rest of the semester, we turn our attention to lossy compression
- But as will be seen next lecture, even lossy compression uses lossless compression

CAN LOSSLESS COMPRESSION COMPRESS EVERY INPUT STREAM? (1/2)

- Can lossless compression compress every input stream?
- Remember the importance of redundancy
  - Compression is the exploitation of redundancy
  - If a data stream has no redundancy, can it be compressed?
- More broadly, let's reason mathematically
- If there is a lossless coder that is guaranteed to compressed every binary stream by at least one bit, what will happen?
  - Use that coder on the input stream, then on the resulting coded bitstream, then on the coded bitstream that results from that, and so on, each time shrinking the bitstream by at least one bit
  - At the end, we get a single-bit coded bitstream, a 0 or a 1
  - Is that possible? Why or why not? (What are the implications?)

CAN LOSSLESS COMPRESSION COMPRESS EVERY INPUT STREAM? (2/2)

1. Here is another mathematical argument
2. Reason by contradiction, and limit ourselves to binary input of n bits each
3. How many different such input streams can there be? 2^n
4. A lossless coder maps the input streams to coded bitstreams in 1-2-1 fashion
   a. No two different binary inputs map to the same encoded bitstream (Why?)
   b. Therefore, the number of coded bitstreams for the 2^n inputs must be 2^n
   c. If the lossless coder compresses everything by at least one bit, then for the n-bit inputs, each coded bitstream has at most n-1 bits
   d. How many binary strings (i.e., coded bitstreams) of at most n-1 bits can there be?

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1} = 2^n - 1 < 2^n$$

   e. Contradiction: (d) contradicts (3).
5. Therefore, there can't be a lossless coder that can compress every input

LOSSLESS COMPRESSION IN STANDARDS

Standards	Compression
JBIG and JBIG 2	Arithmetic coding
Grayscale and color JBIG	bitplane + Arithmetic coding
Lossless JPEG	DPCM
Fax: - Group 3 - Extended 2D Group 3 - Group 4	RLE and Huffman coding

LOSSLESS COMPRESSION IN GRAPHIC FILE FORMATS

Graphic Format	Compression
BMP (Microsoft)	RLE
GIF (CompuServe)	LZW
TIFF	Choice of Group 3, Group 4, LZW, or RLE
PNG	a variant of LZ77 (optionally preceded with a DPCM at the byte level)
MIFF (X Window)	RLE or DPCM
PIX (SGI IRIS)	RLE
BW (SGI IRIS)	RLE

The following graphic file formats do not use compression:
PBM, PGM, PPM, PNM, RAS (SUN Raster file format), PCX

LOSSLESS COMPRESSION IN OS UTILITIES

Utility	Compression
Compress (Unix)	LZW
gzip (Unix)	A variant of LZ77 (Lempel-Ziv 1977)
Windows Zip	LZW, LZ77, and variants DEFLATE: LZSS and Huffman coding

NEXT LECTURE

- General scheme of lossy compression
- Scalar quantization
  - Uniform quantizers
  - Semi-uniform quantizers
  - Optimal Max-Lloyd quantizers