The smallest number requiring nine words to express.

- What is it?
  - Nine hundred ninety nine million nine thousand ninety nine?
  - One trillion to the power of one trillion factorial?

http://algs4.cs.princeton.edu

5.5 DATA COMPRESSION

- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*
- *Kolmogorov complexity*

http://algs4.cs.princeton.edu

5.5 DATA COMPRESSION

- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*
- *Kolmogorov Complexity*

Data compression

Compression reduces the size of a file:

- To save space when storing it.
- To save time when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18–24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, …

> *" Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone. "* — *IBM report on big data (2011)*

Basic concepts ancient (1950s), best technology recently developed.

Applications

Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, HFS+, ZFS.

Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



Communication.

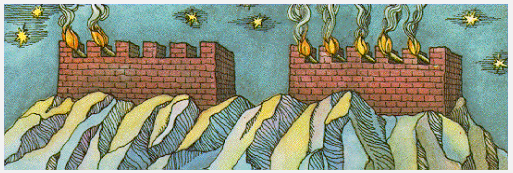
- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype.


Databases. Google, Facebook, ...

Google facebook

Way back in the day

- Polybius, 200 BC



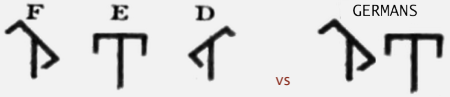
http://people.seas.harvard.edu/~jones/cscie129/images/history/encoding.html

Semaphore telegraph

The Claude Chappe system (1792)

Every sensible person will agree that a single man in a single day could, without interference, cut all the electrical wires terminating in Paris; it is obvious that a single man could sever, in ten places, in the course of a day, the electrical wires of a particular line of communication, without being stopped or even recognized.

[Clearly, no serious competition could be expected from] "a few wretched wires."
" — Dr. Jules Guoyot (July 5th, 1841)



Then: 1 symbol per minute

These days

NETFLIX

Now

Video

- Uncompressed 1080p, 100 megabytes per second
- 1080p on Netflix, ~0.5 megabytes per second

Lossless compression and expansion

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation $C(B)$.

Expand. Reconstructs original bitstream B.

uses fewer bits (you hope)

Basic model for data compression

Compression ratio. Bits in $C(B)$ / bits in B.

Ex. 50–75% or better compression ratio for natural language.

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N-character genome: A T A G A T G C A T A G . . .

Standard ASCII encoding.
- 8 bits per char.
- 8 N bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.
- 2 bits per char.
- 2 N bits.

char	binary
A	00
C	01
T	10
G	11

AG AG AG

0100000101000111 → 0011 → 0100000101000111

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N-character genome: A T A G A T G C A T A G . . .

Standard ASCII encoding.
- 8 bits per char.
- 8 N bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.
- 2 bits per char.
- 2 N bits.

char	binary
A	00
C	01
T	10
G	11

Fixed-length code. k-bit code supports alphabet of size 2^k.

Amazing but true. Initial genomic databases in 1990s used ASCII.

Reading and writing binary data

Binary standard input and standard output. Libraries to read and write bits from standard input and to standard output.

public class BinaryStdIn	
boolean readBoolean()	*read 1 bit of data and return as a* boolean *value*
char readChar()	*read 8 bits of data and return as a* char *value*
char readChar(int r)	*read r bits of data and return as a* char *value*
[similar methods for byte *(8 bits);* short *(16 bits);* int *(32 bits);* long *and* double *(64 bits)]*	
boolean isEmpty()	*is the bitstream empty?*
void close()	*close the bitstream*

public class BinaryStdOut	
void write(boolean b)	*write the specified bit*
void write(char c)	*write the specified 8-bit* char
void write(char c, int r)	*write the r least significant bits of the specified* char
[similar methods for byte *(8 bits);* short *(16 bits);* int *(32 bits);* long *and* double *(64 bits)]*	
void close()	*close the bitstream*

Writing binary data

Date representation. Three different ways to represent 12/31/1999.

A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```

```
00110001 00110010 00101111 00110001 00110001 00101111 00110001 00111001 00111001 00111001
   1        2        /        3        1        /        1        9        9        9
```

80 bits

Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);
BinaryStdOut.write(day);
BinaryStdOut.write(year);
```

```
00000000000000000000000000001100 00000000000000000000000000011111 00000000000000000000011111001111
                12                               31                             1999
```

96 bits

A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```

```
1100 11111 011111100111110 00
 12   31        1999
```

21 bits (+ 3 bits for byte alignment at close)

Binary dumps

Q. How to examine the contents of a bitstream?

Standard character stream

```
% more abra.txt
ABRACADABRA!
```

Bitstream represented as 0 and 1 characters

```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

Bitstream represented with hex digits

```
%  java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
12 bytes
```

Bitstream represented as pixels in a Picture

```
% java PictureDump 16 6 < abra.txt
```

16-by-6 pixel window, magnified

```
96 bits
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Hexadecimal to ASCII conversion table

http://algs4.cs.princeton.edu

5.5 Data Compression

- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*
- *Kolmogorov complexity*

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

```
0000000000000000111111110000000011111111111
```

40 bits

Representation. 4-bit counts to represent alternating runs of 0s and 1s: 15 0s, then 7 1s, then 7 0s, then 11 1s.

```
1111 0111 0111 1011
 15    7    7   11
```

16 bits (instead of 40)

Q. How many bits to store the counts?

A. 4 bits (but we'll use 8 bits in the code on the next slide)

Q. What to do when run length exceeds max count?

A. If longer than 15, intersperse runs of length 0.

```
1111 0000 0001 0111 0111 1011
 15    0    1    7    7   11
```

If we had 16 1s at the front instead

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Run-length encoding: Java implementation

```
public class RunLength
{
   private final static int R   = 256;
   private final static int lgR = 8;

   public static void compress()
   {  /* see textbook */  }

   public static void expand()
   {
      boolean bit = false;
      while (!BinaryStdIn.isEmpty())
      {
         int run = BinaryStdIn.readInt(lgR);
         for (int i = 0; i < run; i++)
            BinaryStdOut.write(bit);
         bit = !bit;
      }
      BinaryStdOut.close();
   }

}
```

maximum run-length count

number of bits per count

read 8-bit count from standard input

write 1 bit to standard output

pad 0s for byte alignment

An application: compress a bitmap

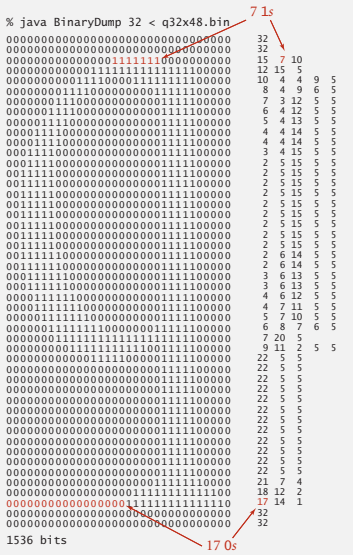
Typical black-and-white-scanned image.

- 300 pixels/inch.
- 8.5-by-11 inches.
- 300 × 8.5 × 300 × 11 = 8.415 million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.

40 lines × 75 chars per line = 3,000 chars.



A typical bitmap, with run lengths for each row

5.5 Data Compression

- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*
- *Kolmogorov complexity*

David Huffman

Variable-length codes

Use different number of bits to encode different chars.

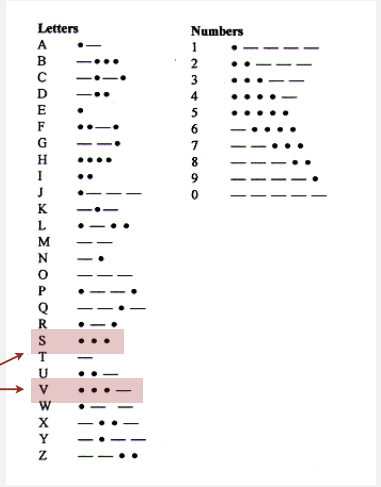
Ex. Morse code: • • • – – – • • •

Issue. Ambiguity.

```
SOS ?
V7 ?
IAMIE ?
EEWNI ?
```

In practice. Use a medium gap to separate codewords.

codeword for S is a prefix of codeword for V



Variable-length codes

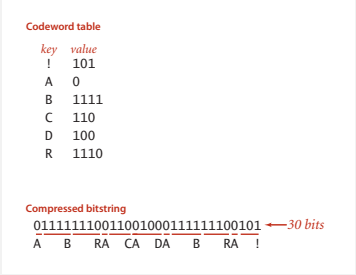
Q. How do we avoid ambiguity?

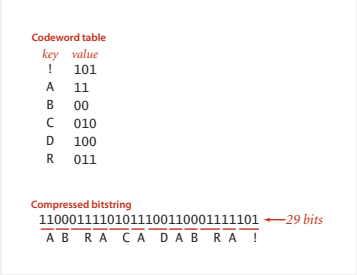
A. Ensure that no codeword is a prefix of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.



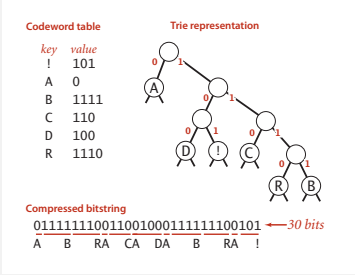


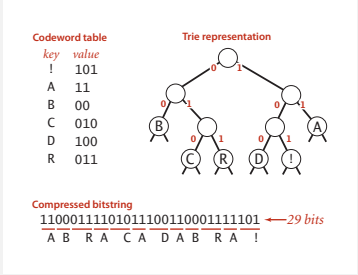
Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

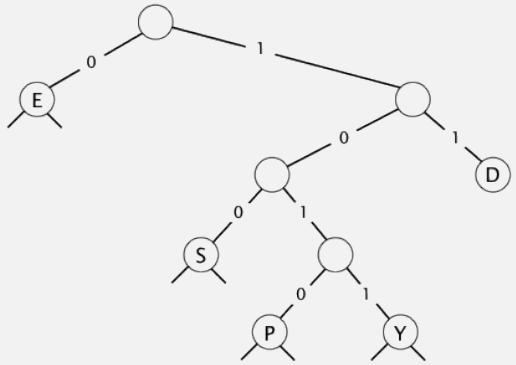
A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.





Expansion



pollEv.com/jhug text to **37607**

Q: Which string is encoded by the bitstream 011001010?

A. EDSEEP [313698]
B. YSPP [313889]
C. EDEEP [317826]
D. EDSEEPE [317827]
E. None [787438]

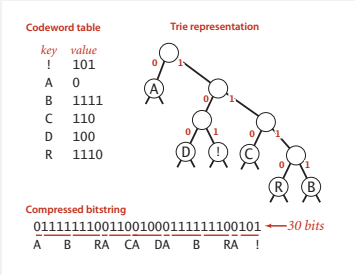
Prefix-free codes: compression and expansion

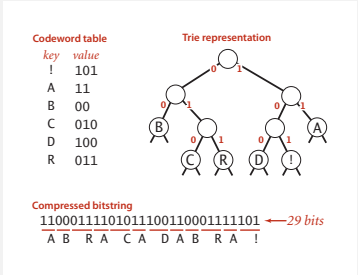
Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.





Huffman coding

Dynamic modeling

- Use a different codebook for EVERY text.

Interesting Tasks Required

- Compress
  - Build trie codebook.
  - Write trie codebook.
  - Compress input (using array codebook).
- Expand
  - Read trie codebook.
  - Expand input using trie codebook.

Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
   private final char ch;   // used only for leaf nodes
   private final int freq;  // used only for compress
   private final Node left, right;

   public Node(char ch, int freq, Node left, Node right)
   {
      this.ch    = ch;
      this.freq  = freq;
      this.left  = left;
      this.right = right;
   }

   public boolean isLeaf()
   {  return left == null && right == null; }

   public int compareTo(Node that)
   {  return this.freq - that.freq;  }
}
```

initializing constructor

is Node a leaf?

compare Nodes by frequency (stay tuned)

Prefix-free codes: expansion

```
public void expand()
{
   Node root = readTrie();
   int N = BinaryStdIn.readInt();

   for (int i = 0; i < N; i++)
   {
      Node x = root;
      while (!x.isLeaf())
      {
         if (!BinaryStdIn.readBoolean())
            x = x.left;
         else
            x = x.right;
      }
      BinaryStdOut.write(x.ch, 8);
   }
   BinaryStdOut.close();
}
```

read in encoding trie

read in number of chars

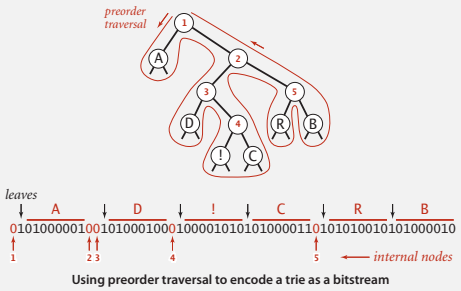
expand codeword for i^{th} char

Running time. Linear in input size N.

Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Using preorder traversal to encode a trie as a bitstream

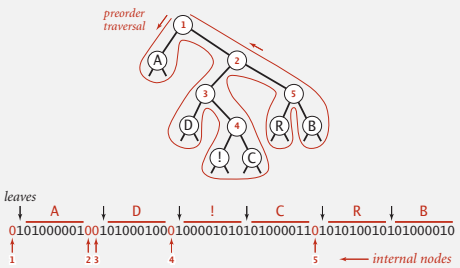
```
private static void writeTrie(Node x)
{
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch, 8);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```

Note. If message is long, overhead of transmitting trie is small.

Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



Using preorder traversal to encode a trie as a bitstream

```
private static Node readTrie()
{
   if (BinaryStdIn.readBoolean())
   {
      char c = BinaryStdIn.readChar(8);
      return new Node(c, 0, null, null);
   }
   Node x = readTrie();
   Node y = readTrie();
   return new Node('\0', 0, x, y);
}
```

used only for leaf nodes

Huffman coding

Interesting Tasks Required

- Compress
  - Build trie codebook.
  - Write trie.
  - Compress input using codebook (codebook as symbol table).
- Expand
  - Read trie.
  - Expand input using trie.

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols S into two subsets S_0 and S_1 of (roughly) equal freq.
- Codewords for symbols in S_0 start with 0; for symbols in S_1 start with 1.
- Recur in S_0 and S_1.

char	freq	encoding
A	5	0...
C	1	0...

S_0 = codewords starting with 0

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

S_1 = codewords starting with 1

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

Huffman algorithm demo

- Count frequency for each character in input.

char	freq	encoding
A		
B		
C		
D		
R		
!		

input

A B R A C A D A B R A !

Huffman algorithm demo

- Count frequency for each character in input.

char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

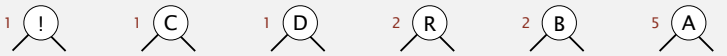
input

A B R A C A D A B R A !

Huffman algorithm demo

- Start with one node corresponding to each character with weight equal to frequency.

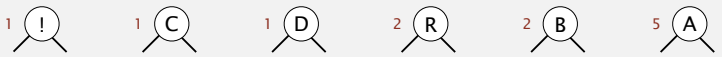
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

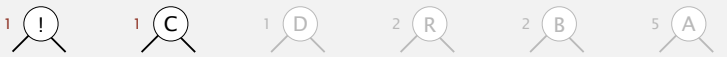
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

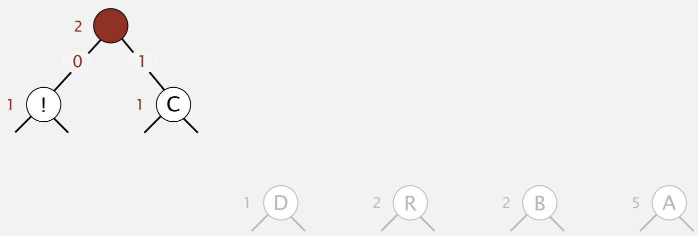
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

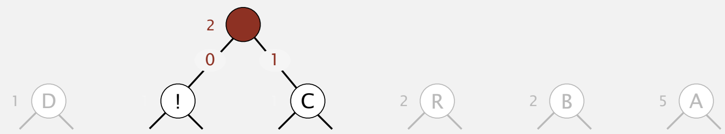
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

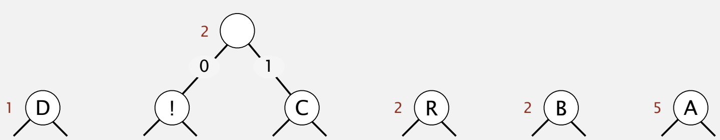
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

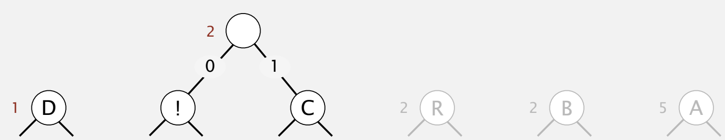
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

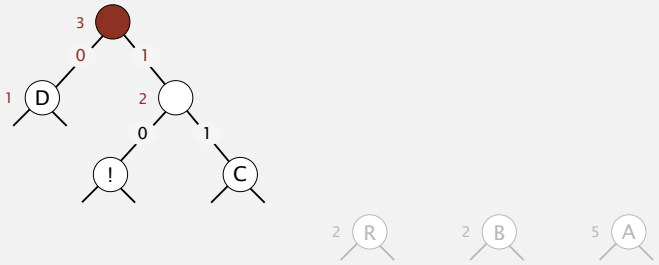
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

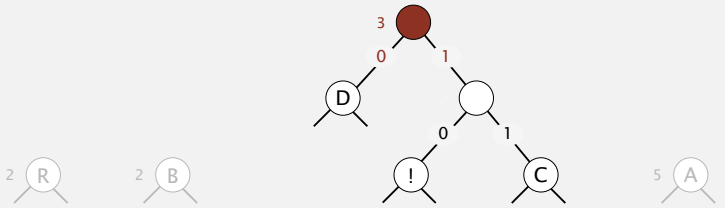
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman algorithm demo

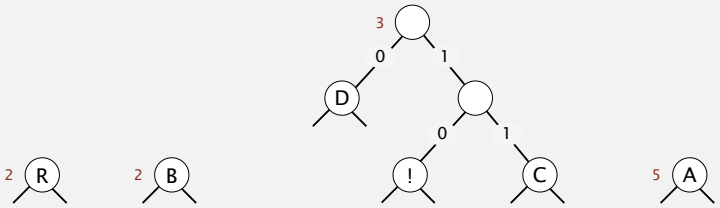
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0

pollEv.com/jhug text to **37607**

Q: What will be the weight of the new trie?

A. 2 [787387] D. 5 [787452]
B. 3 [787388] E. 6 [787453]
C. 4 [787389]



Huffman algorithm demo

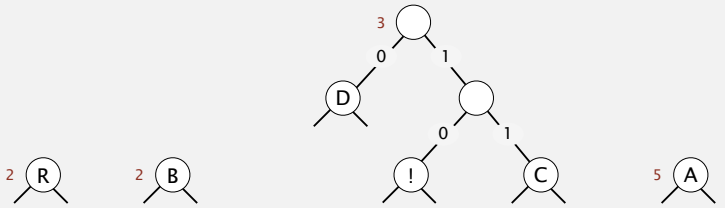
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0

pollEv.com/jhug text to **37607**

Q: What will be the weight of the new trie?

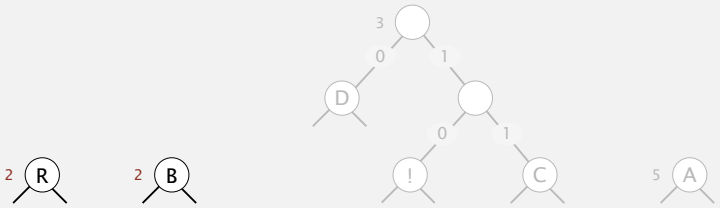
C. 4



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

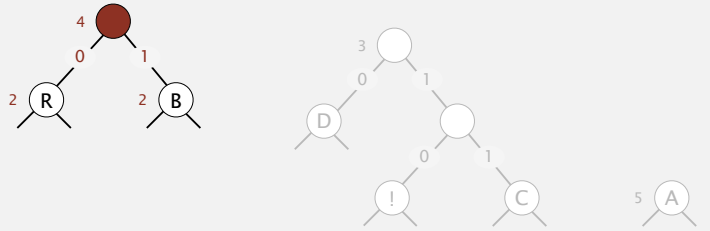
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

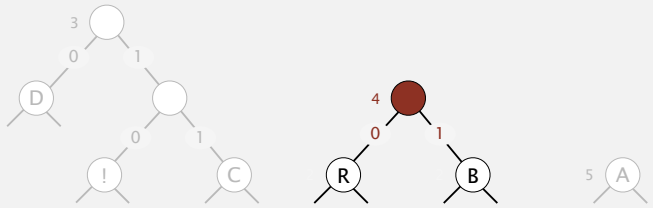
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

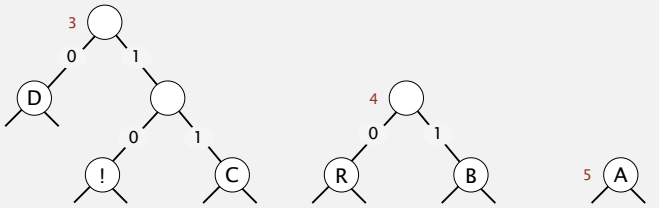
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

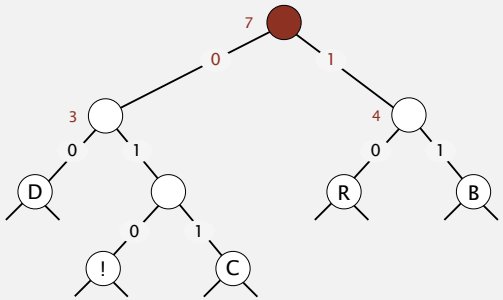
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

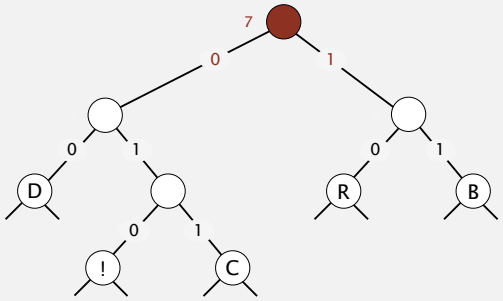
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

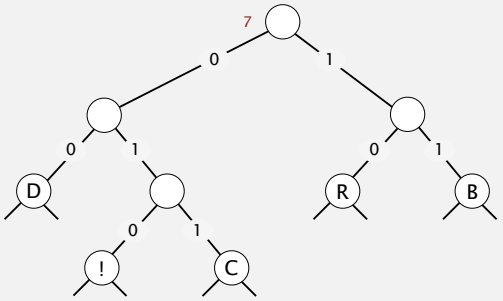
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman algorithm demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

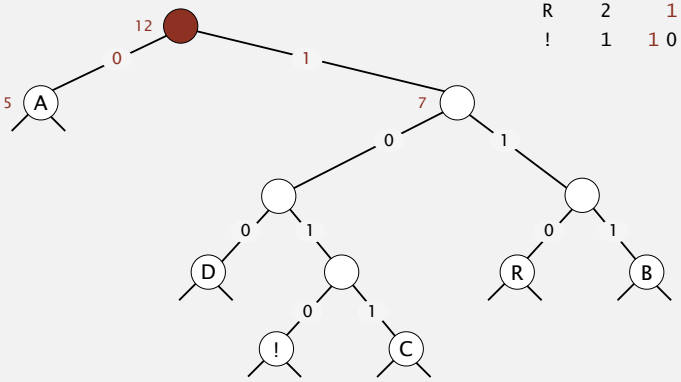
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman algorithm demo

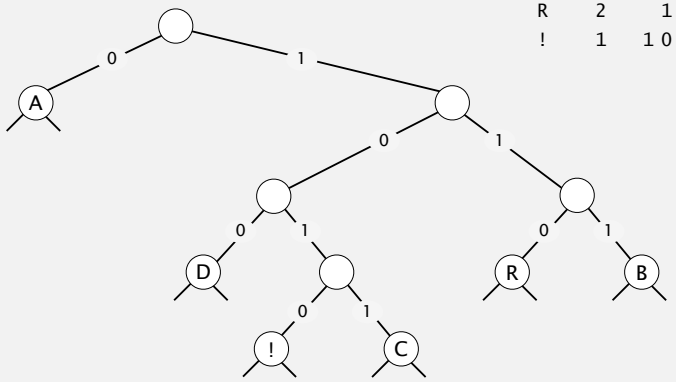
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



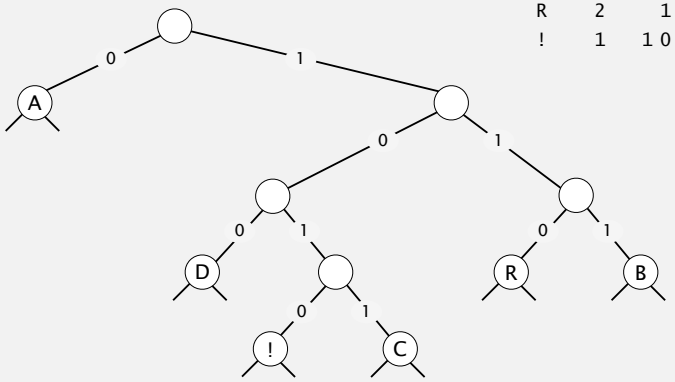
Huffman algorithm demo

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman algorithm demo

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman codes - the most singular moment of one man's life

Q. How to find best prefix-free code?

Huffman algorithm:

- Count frequency `freq[i]` for each char `i` in input.
- Start with one node corresponding to each char `i` (with weight `freq[i]`).
- Repeat until single trie formed:
  - select two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with weight `freq[i] + freq[j]`

Applications:

Constructing a Huffman encoding trie: Java implementation

```
private static Node buildTrie(int[] freq)
{
   MinPQ<Node> pq = new MinPQ<Node>();
   for (char i = 0; i < R; i++)
      if (freq[i] > 0)
         pq.insert(new Node(i, freq[i], null, null));

   while (pq.size() > 1)
   {
      Node x = pq.delMin();
      Node y = pq.delMin();
      Node parent = new Node('\0', x.freq + y.freq, x, y);
      pq.insert(parent);
   }

   return pq.delMin();
}
```

initialize PQ with singleton tries

merge two smallest tries

not used for internal nodes

total frequency

two subtries

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

no prefix-free code uses fewer bits

Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow$ $N + R \log R$.

input size, alphabet size

Q. Can we do better? [stay tuned]

5.5 Data Compression


Abraham Lempel


Jacob Ziv

Statistical methods

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.

LZW compression example

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
matches	A	B	R	A	C	A	D	A B		R A		B R		A B R			A
value	41	42	52	41	43	41	44	81		83		82		88			41 80

LZW compression for A B R A C A D A B R A B R A B R A

key	value
⋮	⋮
A	41
B	42
C	43
D	44
⋮	⋮

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

codeword table

Lempel-Ziv-Welch compression

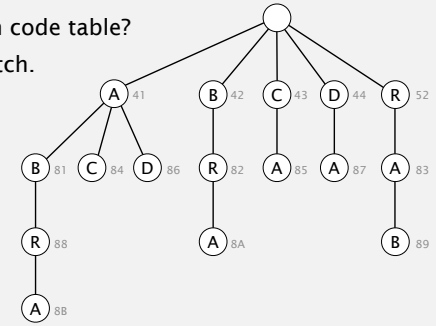
LZW compression.

- Create ST associating W-bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string s in ST that is a prefix of unscanned part of input.
- Write the W-bit codeword associated with s.
- Add $s + c$ to ST, where c is next char in the input.

Q. How to represent LZW compression code table?

A. A trie to support longest prefix match.

key	value
AB	81
BR	82
RA	83
AC	84



longest prefix match

R A B R A B R A

LZW expansion example

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	A B					

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

key	value
87	DA

codeword table

Q: Next two entries?

88: AB, 89: BR [787394]
88: AB, 89: BRA [787396]
88: ABR, 89: BRA [787397]
88: ABR, 89: RAB [787398]

pollEv.com/jhug text to 37607

LZW expansion example

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	A	B	R	A	C	A	D	A B	R A	B R	A B R	A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

key	value
87	DA
88	ABR
89	RAB
8A	BRA
8B	ABRA

codeword table

D. 88: ABR, 89: RAB

LZW expansion

LZW expansion.

- Create ST associating string values with W-bit keys.
- Initialize ST to contain single-char values.
- Read a W-bit key.
- Find associated string value in ST and write it out.
- Update ST.

Q. How to represent LZW expansion code table?

A. An array of size 2^W.

136 ⟶ ABR

key	value
⋮	⋮
65	A
66	B
67	C
68	D
⋮	⋮
129	AB
130	BR
131	RA
132	AC
133	CA
134	AD
135	DA
136	ABR
137	RAB
138	BRA
139	ABRA
⋮	⋮

LZW example: tricky case

input	A	B	A	B	A	B	A
matches	A	B	A B		A B A		
value	41	42	81		83		80

LZW compression for ABABABA

key	value
⋮	⋮
A	41
B	42
C	43
D	44
⋮	⋮

key	value
AB	81
BA	82
ABA	83

codeword table

LZW example: tricky case

value	41	42	81	83	80
output	A	B	A B	A B A	

need to know which key has value 83 before it is in ST!

LZW expansion for 41 42 81 83 80

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BA
83	ABA

codeword table

LZW implementation details

How big to make ST?

- How long is message?
- Whole message similar model?
- [many other variations]

What to do when ST fills up?

- Throw away and start over. [GIF]
- Throw away when not effective. [Unix compress]
- [many other variations]

Why not put longer substrings in ST?

- [many variations have been developed]

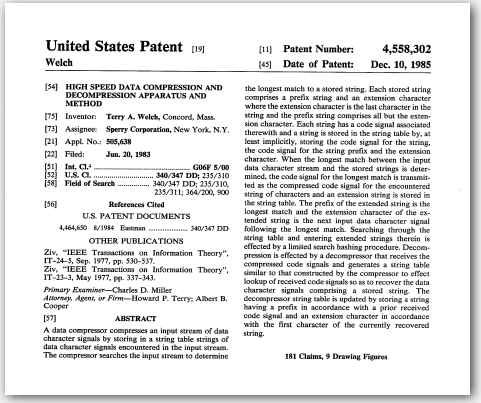
LZW in the real world

Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.

LZ77 not patented ⇒ widely used in open source

LZW patent #4,558,302 expired in U.S. on June 20, 2003



United States Patent [19]
Welch

[11] Patent Number: 4,558,302
[45] Date of Patent: Dec. 10, 1985

[54] HIGH SPEED DATA COMPRESSION AND DECOMPRESSION APPARATUS AND METHOD

[75] Inventor: Terry A. Welch, Concord, Mass.
[73] Assignee: Sperry Corporation, New York, N.Y.
[21] Appl. No.: 505,638
[22] Filed: Jun. 20, 1983

[51] Int. Cl.[4] G06F 5/00
[52] U.S. Cl. ..... 340/347 DD; 235/310
[58] Field of Search ..... 340/347 DD; 235/310, 235/311; 364/200, 900

[56] References Cited
U.S. PATENT DOCUMENTS
4,464,650 8/1984 Eastman ..... 340/347 DD
OTHER PUBLICATIONS
Ziv, "IEEE Transactions on Information Theory", IT-24-5, Sep. 1977, pp. 530-537.
Ziv, "IEEE Transactions on Information Theory", IT-23-3, May 1977, pp. 337-343.

Primary Examiner—Charles D. Miller
Attorney, Agent, or Firm—Howard P. Terry; Albert B. Cooper

[57] ABSTRACT

A data compressor compresses an input stream of data character signals by storing in a string table strings of data character signals encountered in the input stream. The compressor searches the input stream to determine the longest match to a stored string. Each stored string comprises a prefix string and an extension character where the extension character is the last character in the string and the prefix string comprises all but the extension character. Each string has a code signal associated therewith and a string is stored in the string table by, at least implicitly, storing the code signal for the string, the code signal for the string prefix and the extension character. When the longest match between the input data character stream and the stored strings is determined, the code signal for the longest match is transmitted as the compressed code signal for the encountered string of characters and an extension string is stored in the string table. The prefix of the extended string is the longest match and the extension character of the extended string is the next input data character signal following the longest match. Searching through the string table and entering extended strings therein is effected by a limited search hashing procedure. Decompression is effected by a decompressor that receives the compressed code signals and generates a string table similar to that constructed by the compressor to effect lookup of received code signals so as to recover the data character signals comprising a stored string. The decompressor string table is updated by storing a string having a prefix in accordance with a prior received code signal and an extension character in accordance with the first character of the currently recovered string.

181 Claims, 9 Drawing Figures

LZW in the real world

Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate / zlib = LZ77 variant + Huffman.

Unix compress, GIF, TIFF, V.42bis modem: LZW.
zip, 7zip, gzip, jar, png, pdf: deflate / zlib.
iPhone, Sony Playstation 3, Apache HTTP server: deflate / zlib.

Lossless data compression benchmarks

year	scheme	bits / char
1967	ASCII	7.00
1950	Huffman	4.70
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.30
1987	move-to-front	3.24
1987	LZB	3.18
1987	gzip	2.71
1988	PPMC	2.48
1994	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29 ← Coursera programming assignment
1997	BOA	1.99
2010	PAQ	1.30

data compression using Calgary corpus

http://algs4.cs.princeton.edu

5.5 Data Compression

- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*
- *Kolmogorov complexity*

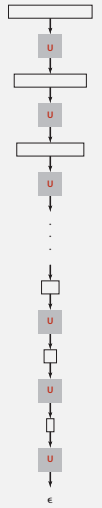
Andrej Kolmogorov

Universal data compression

Proposition. No algorithm can compress every bitstring.

Pf. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bitstring B_0, compress it to get smaller bitstring B_1.
- Compress B_1 to get a smaller bitstring B_2.
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed to 0 bits!



Universal data compression?

Compression

Compression likelihood

- How many different bitstreams are there with 5 bits?
  - 2^5=32
- How many of these sequences can be uniquely represented with EXACTLY 4 bits?
  - 2^4=16
- What fraction of our sequences can be represented with EXACTLY 2 bits?
  - $2^2/2^5 = 1/8$
- What fraction can be represented with 3 bits or fewer?
  - Number of 3 bit or fewer sequences: $1+2+2^2+2^3 = 2^4 - 1$
  - Fraction: 15/32
- What fraction of the sequences of N bits can be represented by N/2 bits?
  - Approximately: $2^{N/2+1} / 2^N$
  - 1 out of $2^{N/2+1}$
  - For N=1000, this is 1 out of 2^{501}

This plant

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits

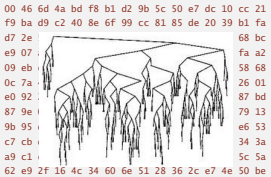
Run length encoding:
4224800 bits

hugPlant.bmp → hugPlant.huf

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits

Huffman → Total: 1994024 bits



```
00 46 6d 4a bd f8 b1 d2 9b 5c 50 e7 dc 10 cc 21
f9 ba d9 c2 40 8e 6f 99 cc 81 85 de 20 39 b1 fa
d7 2e                                       68 bc
e9 07                                       fa a2
09 eb                                       58 68
0c 7a                                       26 01
e0 92                                       87 bd
87 9e                                       79 13
9b 95                                       e6 53
c7 cb                                       34 3a
a9 c1                                       5c 5a
62 e9 2f 16 4c 34 60 6e 51 28 36 2c e7 4e 50 be
c0 15 1b 01 d9 c0 bd b4 20 87 42 be d4 e2 23 a2
b6 84 22 4c cf 74 cd 4f 23 06 54 e6 c2 0f 2d bd
e5 81 f4 c6 de 15 59 f1 68 a4 a5 88 16 b0 7f bf
8a 1d 98 bd 33 b4 d5 71 22 93 81 af e0 cc ce 12
57 23 62 3a e4 3d 8c f1 12 8d a5 40 3b 70 d6 9b
12 49 62 8d 6f d4 52 f6 7f d5 11 7c ca 07 dd e3
dc 1c 7f c4 a4 69 77 6e 5e 60 db 5a 69 01 95 c8
cc 16 b4 0e 64 34 68 b4 6a 8c 50 78 6c cf 9f fe
```

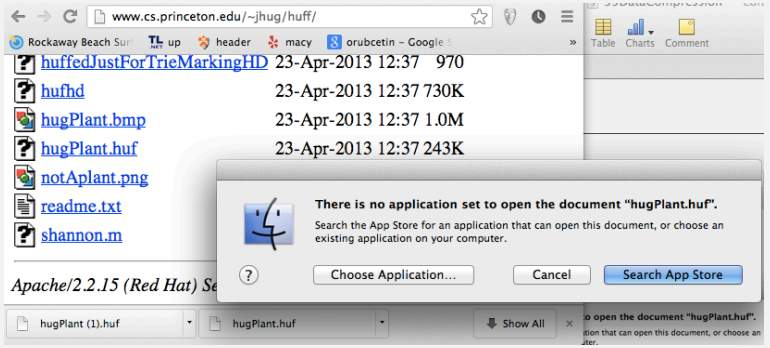
Trie: 2560 bits

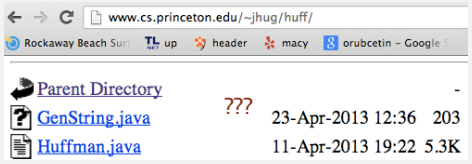
```
00 20 00 f4 c3 b7 6d c2 31 24 92 dc 24 a7 c9 25
ae 24 b5 c4 85 88 40 be c4 92 46 25 79 2f c4 af
25 f8 92 49 24 92 64 c9 92 49 30 b1 24 92 49 24
2c 49 24 92 49 0b 12 49 24 92 42 c4 92 49 24 92
49 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ...
```

Image data: 1991432 bits
+ 32 bits for length

25% ratio!

Decoding





hugPlant.bmp → hugPlant.huf

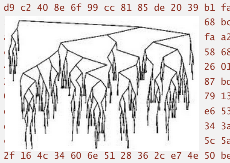
```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits

Huffman

Total: 2037424 bits

```
00 46 6d 4a bd f8 b1 d2 9b 5c 50 e7 dc 10 cc 21
f9 ba d9 c2 40 8e 6f 99 cc 81 85 de 20 39 b1 fa
d7 2e 68 bc
e9 07 fa a2
09 eb 58 68
0c 7a 26 01
e0 92 87 bd
87 9e 79 13
9b 95 e6 53
c7 cb 34 3a
a9 c1 5c 5a
62 e9 2f 16 4c 34 60 6e 51 28 36 2c e7 4e 50 be
c0 15 1b 01 d9 c0 bd b4 20 87 42 be d4 e2 23 a2
b6 84 22 4c cf 74 cd 4f 23 06 54 e6 c2 0f 2d bd
e5 81 f4 c6 de 15 59 f1 68 a4 a5 88 16 b0 7f bf
8a 1d 98 bd 33 b4 d5 71 22 93 81 af e0 cc ce 12
57 23 62 3a e4 3d 8c f1 12 8d a5 40 3b 70 d6 9b
12 49 62 8d 6f d4 52 f6 7f d5 11 7c ca 07 dd e3
dc 1c 7f c4 a4 69 77 6e 5e 60 db 5a 69 01 95 c8
cc 16 b4 0e 64 34 68 b4 6a 8c 50 78 6c cf 9f fe
```



Trie: 2560 bits

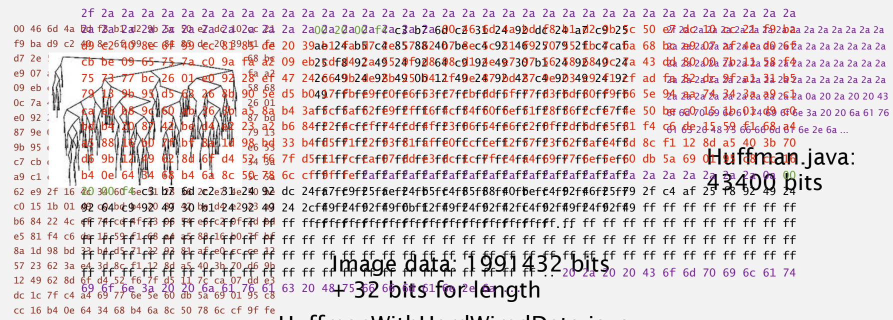
```
00 20 00 f4 c3 b7 6d c2 31 24 92 dc 24 a7 c9 25
ae 24 b5 c4 85 88 40 be c4 92 46 25 79 2f c4 af
25 f8 92 49 24 92 64 c9 92 49 30 b1 24 92 49 24
2c 49 24 92 49 0b 12 49 24 92 42 c4 92 49 24 92
49 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ...
```

Image data: 1991432 bits
+ 32 bits for length

```
2f 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 0a 20 2a 20 20 43
6f 6d 70 69 6c 61 74 69 6f 6e 3a 20 20 6a 61 76
61 63 20 48 75 66 66 6d 61 6e 2e 6a ...
```

Huffman.java:
43400 bits

Data is bits. Code is bits.



HuffmanWithHardWiredData.java:
~2000000 bits

Code	(purple)
Trie	(red)
ImageData	(green/black)

A program unambiguously describes a bitstream

```
2f 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a
2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 00 46 6d 4a bd f8 b1 d2 9b 5c 50 e7 dc 10 cc 21 f9 ba
d9 c2 40 8e 6f 99 cc 81 85 de 20 39 b1 fa d7 2e 57 62 b7 8e 5c 51 f9 70 55 1b 7c ba 68 bc e9 07 af 4e d0 6f
cb be 09 65 75 7a c0 9a fa a2 09 eb cd d4 2a 55 9f d8 98 d1 4e e7 97 56 58 68 0c 7a 43 dd 80 00 7b 11 58 f4
75 73 77 bc 26 01 e0 92 28 ef 47 24 66 9b de 8b 25 04 1f 0e 87 bd 87 9e 03 c9 f1 cf ad fa 82 dc 9f a1 31 b5
79 13 9b 95 d5 63 26 8b 90 5e d5 b0 17 fb e9 c0 e6 53 c7 cb dd 5f 77 d3 bd 80 f9 b6 5e 94 aa 74 34 3a a9 c1
ca e6 b8 9c 60 ab 36 3b a5 8a b4 3a 5c 5a 62 e9 2f 16 4c 34 60 6e 51 28 36 2c e7 4e 50 be c0 15 1b 01 d9 c0
bd b4 20 87 42 be d4 e2 23 a2 b6 84 22 4c cf 74 cd 4f 23 06 54 e6 c2 0f 2d bd e5 81 f4 c6 de 15 59 f1 68 a4
a5 88 16 b0 7f bf 8a 1d 98 bd 33 b4 d5 71 22 93 81 af e0 cc ce 12 57 23 62 3a e4 3d 8c f1 12 8d a5 40 3b 70
d6 9b 12 49 62 8d 6f d4 52 f6 7f d5 11 7c ca 07 dd e3 dc 1c 7f c4 a4 69 77 6e 5e 60 db 5a 69 01 95 c8 cc 16
b4 0e 64 34 68 b4 6a 8c 50 78 6c cf 9f fe 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 2a 0a 00
20 00 f4 c3 b7 6d c2 31 24 92 dc 24 a7 c9 25 ae 24 b5 c4 85 88 40 be c4 92 46 25 79 2f c4 af 25 f8 92 49 24
92 64 c9 92 49 30 b1 24 92 49 24 2c 49 24 92 49 0b 12 49 24 92 42 c4 92 49 24 92 49 ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff .. 20 2a 20 20 43 6f 6d 70 69 6c 61 74
69 6f 6e 3a 20 20 6a 61 76 61 63 20 48 75 66 66 6d 61 6e 2e 6a ...
```

HuffmanWithHardWiredData.java:
~2000000 bits

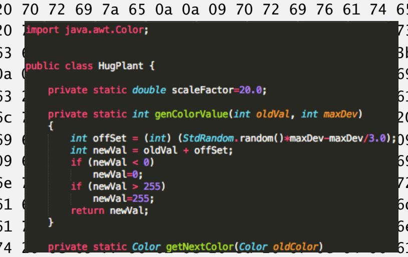
Java

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

8389584 bits

HugPlant.java → hugPlant.bmp

```
69 6d 70 6f 72 74 20 6a 61 76 61 2e 61 77 74 2e
43 6f 6c 6f 72 3b 0a 0a 0a 70 75 62 6c 69 63 20
63 6c 61 73 73 20 48 75 67 50 6c 61 6e 74 20 7b
0a 09 2f 2f 73 65 6e 64 20 65 6d 61 69 6c 20 74
6f 20 70 72 69 7a 65 40 6a 6f 73 68 68 2e 75 67
20 74 6f 20 72 65 63 65 69 76 65 20 79 6f 75 72
20 70 72 69 7a 65 0a 0a 09 70 72 69 76 61 74 65
20 73
63 3b
0a 69
63 61
6c 20
69 09
09 69
6e 72
61 6d
61 6e
74 61
6c 20 2b 20 6f 66 66 53 65 74 3b ...
```



```
import java.awt.Color;

public class HugPlant {

    private static double scaleFactor=20.0;

    private static int genColorValue(int oldVal, int maxDev)
    {
        int offSet = (int) (StdRandom.random()*maxDev-maxDev/3.0);
        int newVal = oldVal + offSet;
        if (newVal < 0)
            newVal=0;
        if (newVal > 255)
            newVal=255;
        return newVal;
    }

    private static Color getNextColor(Color oldColor)
```

Total: 29432 bits

Java

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits

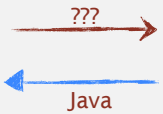
Of Note:

- 0.35% compression ratio!
- Of the 2^(8389584) possible bit streams, only roughly one in 2^(8360151) can be compressed this well.

So what?

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits




```
69 6d 70 6f 72 74 20 6a 61 76 61 2e 61 77 74 2e
43 6f 6c 6f 72 3b 0a 0a 70 75 62 6c 69 63 20
63 6c 61 73 73 20 48 75 67 50 6c 61 6e 74 20 7b
0a 09 2f 2f 73 65 6e 64 20 65 6d 61 69 6c 20 74
6f 20 70 72 69 7a 65 40 6a 6f 73 68 68 2e 75 67
20 74 6f 20 72 65 63 65 69 76 65 20 79 6f 75 72
20 70 72 69 7a 65 0a 0a 09 70 72 69 76 61 74 65
20 73 74 61 74 69 63 20 64 6f 75 62 6c 65 20 73
63 61 6c 65 46 61 63 74 6f 72 3d 32 30 2e 30 3b
0a 0a 09 70 72 69 76 61 74 65 20 73 74 61 74 69
63 20 69 6e 74 20 67 65 6e 43 6f 6c 6f 72 56 61
6c 75 ...
```

Total: 29432 bits

Interesting questions:

- Do different programming languages have different sets of 'easy' bitstreams?
- Is there some way to find the smallest Java program that generates a given bitstream?

Kolmogorov Complexity

Definition

- Given a bitstream x, the Java Kolmogorov complexity ***K(x)*** is the length in bits of the shortest Java program (also a bitstream) that outputs x.

Claim

- Given any arbitrary Turing complete programming language P.
  - $K(x) < K_P(x) + C_P$
  - Shortest program in Java is no more than a constant ADDITIVE factor larger than the shortest program in P.

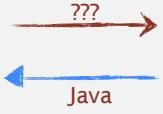
- Proof:
  - Since Java and P are both Turing complete languages, we can write a compiler for P in Java.
  - The size of this compiler is independent of x, and is given by C_P.

$K($ $) \leq 29432$

So what?

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits




```
69 6d 70 6f 72 74 20 6a 61 76 61 2e 61 77 74 2e
43 6f 6c 6f 72 3b 0a 0a 70 75 62 6c 69 63 20
63 6c 61 73 73 20 48 75 67 50 6c 61 6e 74 20 7b
0a 09 2f 2f 73 65 6e 64 20 65 6d 61 69 6c 20 74
6f 20 70 72 69 7a 65 40 6a 6f 73 68 68 2e 75 67
20 74 6f 20 72 65 63 65 69 76 65 20 79 6f 75 72
20 70 72 69 7a 65 0a 0a 09 70 72 69 76 61 74 65
20 73 74 61 74 69 63 20 64 6f 75 62 6c 65 20 73
63 61 6c 65 46 61 63 74 6f 72 3d 32 30 2e 30 3b
0a 0a 09 70 72 69 76 61 74 65 20 73 74 61 74 69
63 20 69 6e 74 20 67 65 6e 43 6f 6c 6f 72 56 61
6c 75 ...
```

Total: 29432 bits

Interesting questions:

- Do different programming languages have different sets of 'easy' bitstreams? No!
- Is there some way to find the smallest Java program that generates a given bitstream? No, but we won't discuss the proof in class. See following slides or come to flipped lecture. No proving this won't be on the exam. Yes, it is a super cool proof.

K(X)

Upper bound is easy

- x = "What is the shortest Java program that can output exactly this Stringgggggggggggggggggggggggggggggggggggggggggggggggg??? ??? ??????????????????"

```
public class PrintX {
  public static void main(String[] args) {
     String x = "What is the shortest Java program that
  can output exactly this Stringgggggggggggggggggggggg
  gggggggggggggggggggggggggg?????????????????????????
  ??????????????????????????????????????????????????
  ??????????????????????????????";
     System.out.println(x);
   }
}
```

- Might be able to do better, but no reason to do worse than ***K(x)*** ≤ 2680.

StringGenerator - a tool for exploring random strings

```
StringGenerator sg = new StringGenerator();
String s0 = sg.next();
String s1 = sg.next();
String s2 = sg.next();
...
String s65538 = sg.next();
...
```

s0	0000
s1	0001
s2	0002
...	
s65536	FFFF
s65536	0000 0000
s65537	0000 0001
s65538	0000 0002
...	
s131071	0000 FFFF
s131072	0001 0000

Brute force approach for finding K(x)

```
public class FindBestClass {
   public static void main(String[] args) {
      String x = args[0];
      StringGenerator sg = new StringGenerator();
       while(true) {
          String sourceCode = sg.next();
          String output = CompileRunAndReturn(sourceCode);
          if (output.equals(x)) {
             System.out.println(sourceCode);
             return;
          }
    }
}
```

- Prints source code of shortest program that outputs x.

Brute force approach for finding K(x)

```
public class FindBestClass {
   public static void main(String[] args) {
      String x = args[0];
      StringGenerator sg = new StringGenerator();
       while(true) {
          String sourceCode = sg.next();
          String output = CompileRunAndReturn(sourceCode);
          if (output.equals(x)) {
             System.out.println(sourceCode);
             return;
          }
    }
}
```

- Prints source code of shortest program that outputs x.
- K(x): Length of source code.

Brute force approach for finding K(x)

```
public class FindBestClass {
   public static void main(String[] args) {
      String x = args[0];
      StringGenerator sg = new StringGenerator();
       while(true) {
          String sourceCode = sg.next();
          String output = CompileRunAndReturn(sourceCode);
          if (output.equals(x)) {
             System.out.println(sourceCode);
             return;
          }
    }
}
```

- Prints source code of shortest program that outputs x.
- K(x): Length of source code.
- Will this approach work (if we're willing to wait a long long time)?

Brute force approach for finding K(x)

```
public class FindBestClass {
   public static void main(String[] args) {
      String x = args[0];
      StringGenerator sg = new StringGenerator();
       while(true) {
          String sourceCode = sg.next();
          String output = CompileRunAndReturn(sourceCode);
          if (output.equals(x)) {
             System.out.println(sourceCode);
             return;
          }
    }
}
```

- Prints source code of shortest program that outputs x.
- K(x): Length of source code.

- Will this approach work (if we're willing to wait a long long time)?
  - No! Some of the random programs will go into an infinite loop.

Is computing K(x) theoretically possible?

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)
```

- Gives ***K(x)***, but does not provide the actual Java source code!

public class ComplexStringFinder

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)
```

```
public String GenerateComplexString(int n) {
   int sLength = 0;

  while (true) {
      i++;
      StringGenerator sg = new StringGenerator(i);
      while (sg.hasNext()) {
         String complexString = sg.next();
         if (KolmogorovComplexity(complexString) >= n)
            return complexString;
      }
   }
}
```

- If there is a String of length `i` with complexity n, the inner loop returns this String.
- `GenerateComplexString(n)` gives shortest string of complexity n.

public class ComplexStringFinder

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)
```

```
//Returns a string with Kolmogorov Complexity at least n
public String GenerateComplexString(int n)
```

G bits

Observations

- `GenerateComplexString(n)`
  - G total bits for the code (including both methods).
  - `lg n` bits for parameter n.
- Java program of length `lg n` + G bits provides an x such that ***K(x)*** ≥ *n*.

public class ComplexStringFinder

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)

//Returns a string with Kolmogorov Complexity at least n
public String GenerateComplexString(int n)
```

G bits

Observations

- GenerateComplexString(n)
  - G total bits for the code (including both methods).
  - lg n bits for parameter n.
- Java program of length lg n + G bits provides an x such that ***K(x)*** ≥ *n.*
  - ***DANGER!!***

public class ComplexStringFinder

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)

//Returns a string with Kolmogorov Complexity at least n
public String GenerateComplexString(int n)
```

G bits

Observations

- GenerateComplexString(n)
  - G total bits for the code (including both methods).
  - lg n bits for parameter n.
- Java program of length lg n + G bits provides an x such that ***K(x)*** ≥ *n.*
  - ***DANGER!!***
- Our Java program of length lg n_0 + G bits generates strings that can only be generated by Java programs of length n_0.

public class ComplexStringFinder

```
//Returns Kolmogorov Complexity of a string x
public int KolmogorovComplexity(String x)

//Returns a string with Kolmogorov Complexity at least n
public String GenerateComplexString(int n)
```

G bits

Observations

- GenerateComplexString(n)
  - G total bits for the code (including both methods).
  - lg n bits for parameter n.
- Java program of length lg n + G bits provides an x such that ***K(x)*** ≥ *n.*
  - ***DANGER!!***
- Our Java program of length lg n_0 + G bits generates strings that can only be generated by Java programs of length n_0.
  - Example: Suppose G is 100,000 bits. If we pick n_0=1,000,000, then Java program is of length 100,020, but ***K(x)*** ≥ 1,000,000.

Berry Paradox

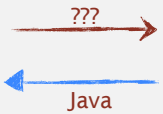
Let n be "the smallest number requiring nine words to express"

- But this word can be expressed in eight words, namely:
  "the smallest number requiring nine words to express".
  1 2 3 4 5 6 7 8
- There is no such number!
- Not quite the same thing as our Kolmogorov paradox (Java is a better defined language than English).
  - Kolmogorov complexity IS a specific number.
  - You just can't find it!

So what?

```
42 4d 7a 00 10 00 00 00 00 00 7a
00 00 00 6c 00 00 00 00 02 00 00
00 02 00 00 01 00 20 00 03 00 00
00 00 00 10 00 12 0b 00 00 12 0b
00 00 00 00 00 00 00 00 00 00 00
00 ff 00 00 ff 00 00 ff 00 00 00
00 00 00 ff 01 00 00 00 00 00 00
00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 01 00 ...
```

Total: 8389584 bits




```
69 6d 70 6f 72 74 20 6a 61 76 61 2e 61 77 74 2e
43 6f 6c 6f 72 3b 0a 0a 70 75 62 6c 69 63 20
63 6c 61 73 73 20 48 75 67 50 6c 61 6e 74 20 7b
0a 09 2f 2f 73 65 6e 64 20 65 6d 61 69 6c 20 74
6f 20 70 72 69 7a 65 40 6a 6f 73 68 68 2e 75 67
20 74 6f 20 72 65 63 65 69 76 65 20 79 6f 75 72
20 70 72 69 7a 65 0a 0a 09 70 72 69 76 61 74 65
20 73 74 61 74 69 63 20 64 6f 75 62 6c 65 20 73
63 61 6c 65 46 61 63 74 6f 72 3d 32 30 2e 30 3b
0a 0a 09 70 72 69 76 61 74 65 20 73 74 61 74 69
63 20 69 6e 74 20 67 65 6e 43 6f 6c 6f 72 56 61
6c 75 ...
```

Total: 29432 bits

Interesting questions:

- Do different programming languages have different sets of 'easy' bitstreams? No!
- Is there some way to find the smallest Java program that outputs a given bitstream? No!

An uncomputable function

Neat Facts.

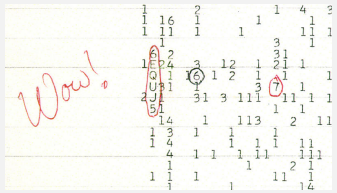
- There exists SOME function **K(X)** that gives the Java Kolmogorov complexity of a given bitstream X.
- We can compute this function for some bitstreams. For example, in Python, we can exactly compute $K_P(X)$ for some small X:
  - $K_P(6) = 56$: print 6
  - $K_P(387420489) = 80$: print 9**9
- It is **impossible** to compute **K(X)** for arbitrary X.

Kolmogorov complexity summary

Kolmogorov Complexity

- Uncomputable function.
- Represents the compressibility of a bitstream using ANY technique.
  - a.k.a. measures randomness of the bitstream.
- Most bitstreams cannot be compressed using any technique in any language.
- No Turing complete language is better at compressing any sufficiently long bitstream than any other.

hugPlant: Simple



SETI data: Complex

Data compression summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]
- Special purpose code for generating a specific bitstream. [hugPlant]

Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

Theoretical limits on compression. Shannon entropy: $H(X) = -\sum_i^n p(x_i) \lg p(x_i)$

- Better upper bounds on K(x)
  - Human experiments on English text: 1.1 bits/character

Practical compression. Use extra knowledge whenever possible.