CS61B

Lectures 38: Compression

- Prefix Free Codes
- Huffman Coding
- Theory of Compression
- LZW (Extra)
- Lossy Compression (Extra)

Zip Files, How Do They Work?

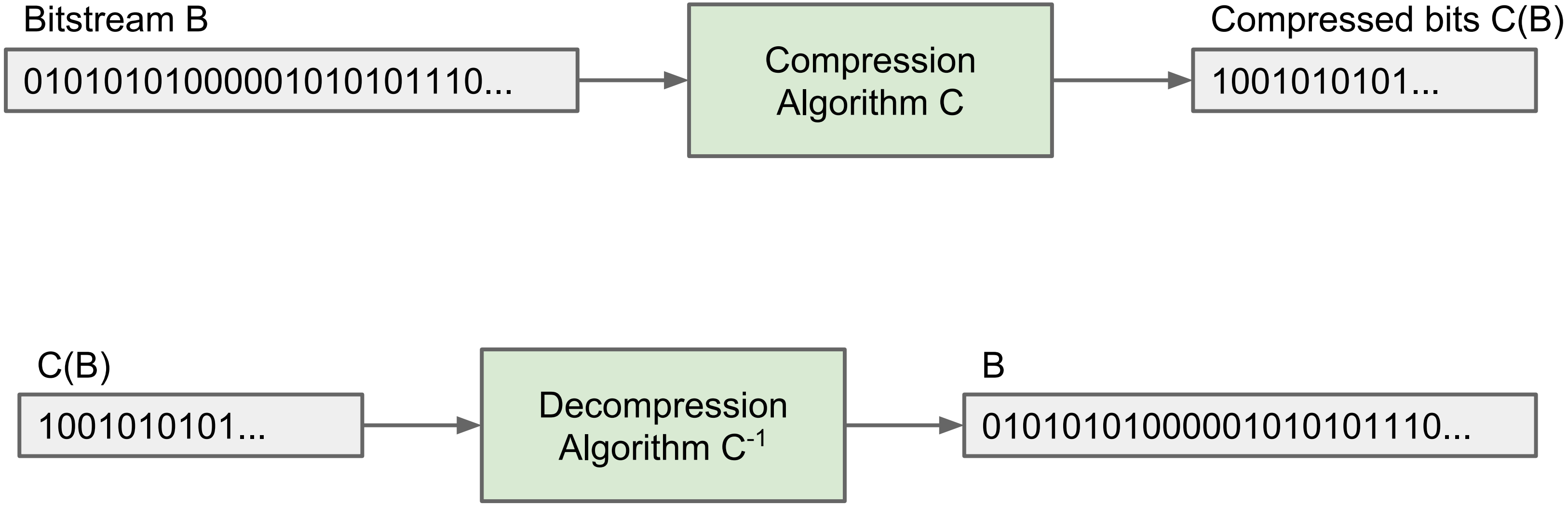
```
$ zip mobydick.zip mobydick.txt
  adding: mobydick.txt (deflated 59%)
$ ls -l
-rw-rw-r-- 1 jug jug 643207 Apr 24 10:55 mobydick.txt
-rw-rw-r-- 1 jug jug 261375 Apr 24 10:55 mobydick.zip
```

Size in Bytes

```
$ unzip mobydick.zip
replace mobydick.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename: r
new name: unzipped.txt
  inflating: unzipped.txt
$ diff mobydick.txt unzipped.txt
$
```

File is unchanged by zipping / unzipping.

Compression Model #1: Algorithms Operating on Bits



Bitstream B

01010101000001010101110...

Compression Algorithm C

Compressed bits C(B)

1001010101...

C(B)

1001010101...

Decompression Algorithm C^{-1}

B

01010101000001010101110...

In a ***lossless*** algorithm we require that no information is lost.

- Text files often compressible by 70% or more.

Prefix Free Codes

Increasing Optimality of Coding

By default, English text is usually represented by sequences of characters, each 8 bits long, e.g. ‘d’ is 01100100.

word	binary	hexadecimal
dog	01100100 01101111 01100111	64 6f 67

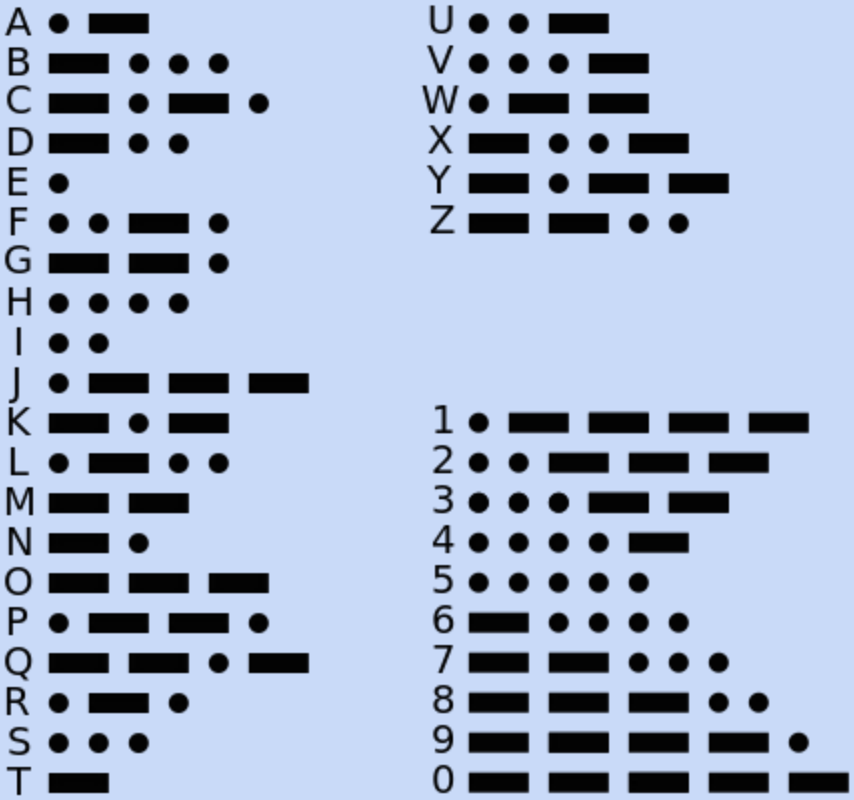
Easy way to compress: Use fewer than 8 bits for each letter.

- Have to decide which bit sequences should go with which letters.
- More generally, we’d say which ***codewords*** go with which ***symbols.***

More Code: Mapping Alphanumeric Symbols to Codewords

Example: Morse code.

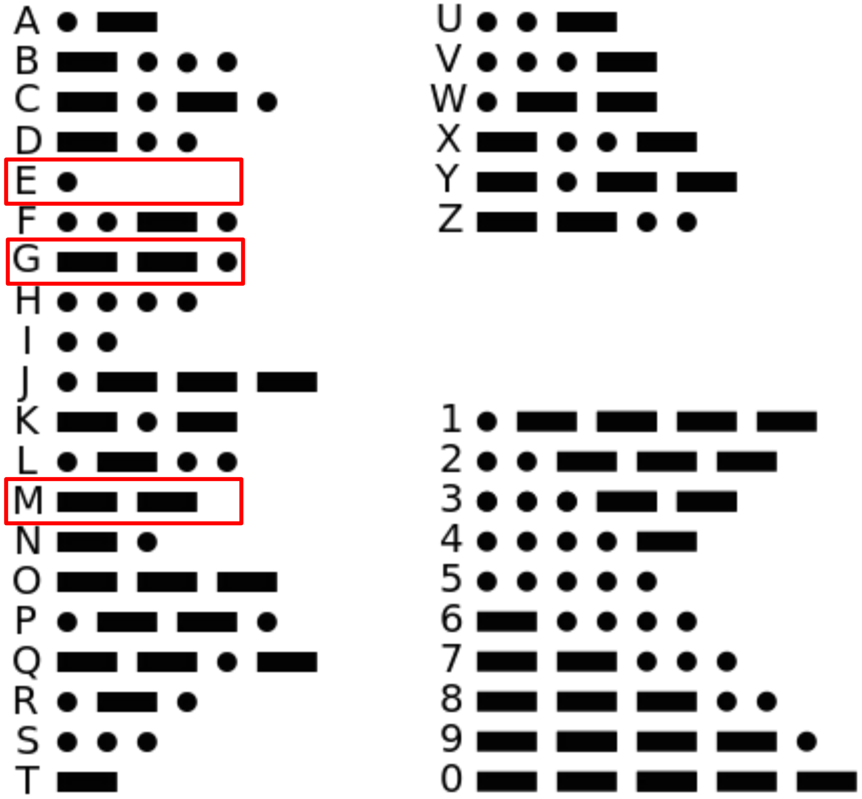
- Goal: Compact representation.
- What is – – • – – •?



More Code: Mapping Alphanumeric Symbols to Codewords

Example: Morse code.

- Goal: Compact representation.
- What is – – • – – •? It’s ambiguous!
  - MEME
  - GG

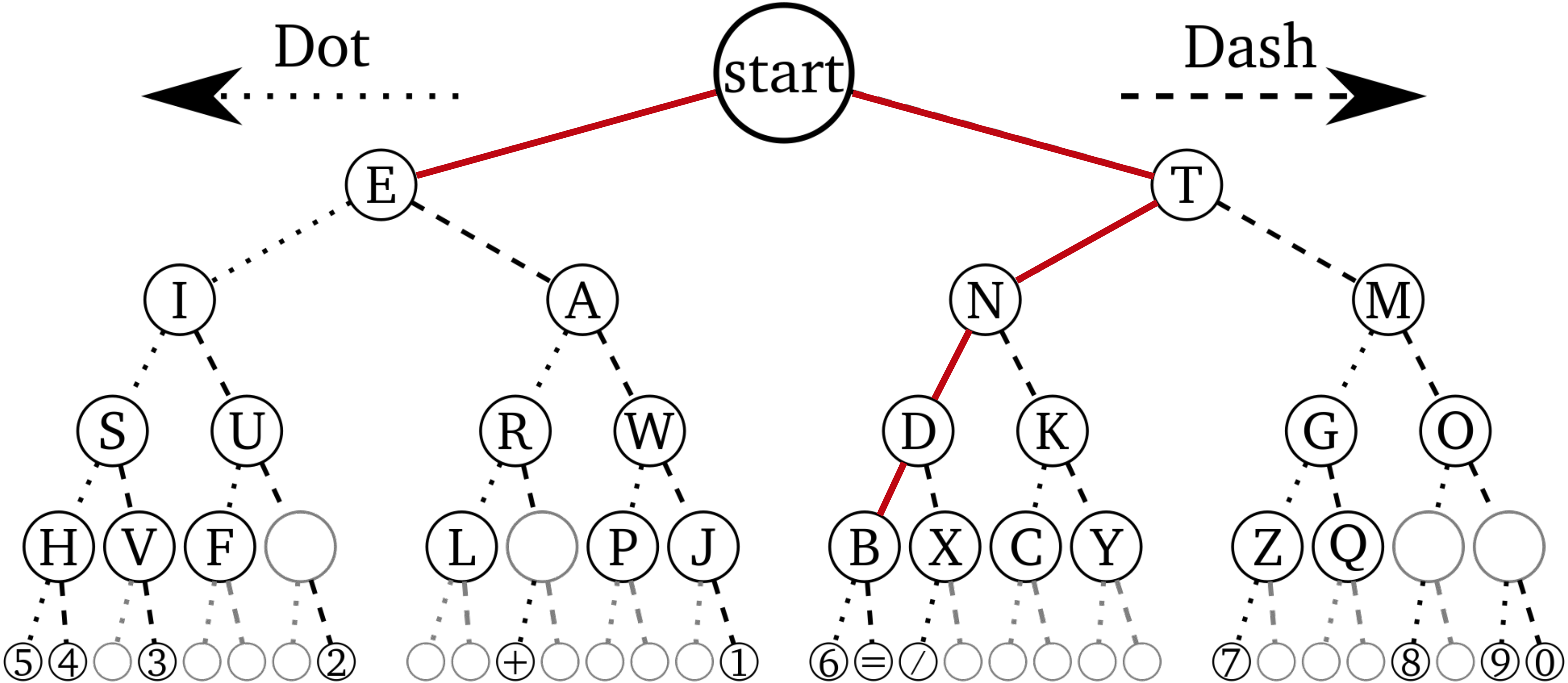


Note:

- Can think of dot as 0, dash as 1.
- Operators pause between codewords to avoid ambiguity.
  - Pause acts as a 3rd symbol.

Alternate strategy: Avoid ambiguity by making code ***prefix free***.

Morse Code (as a Tree)

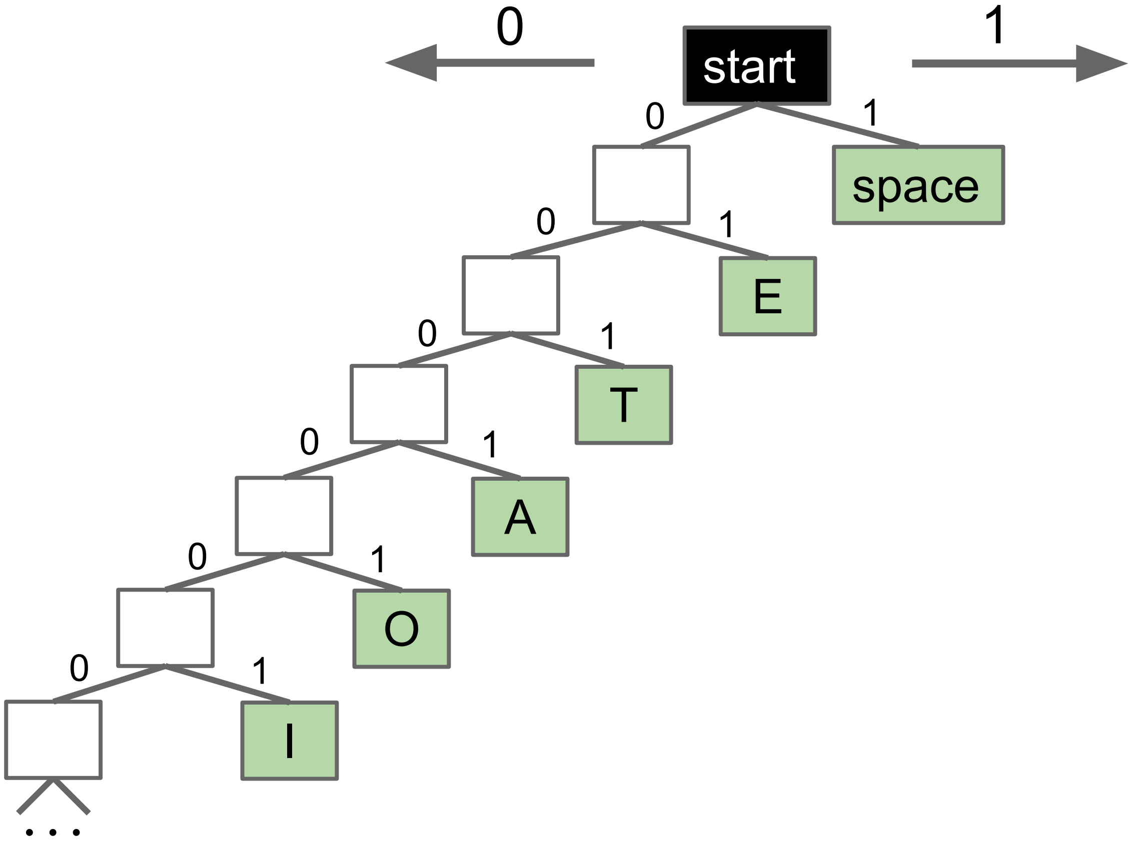


From Wikimedia

Prefix-Free Codes [Example 1]

A prefix-free code is one in which no codeword is a prefix of any other. Example for English:

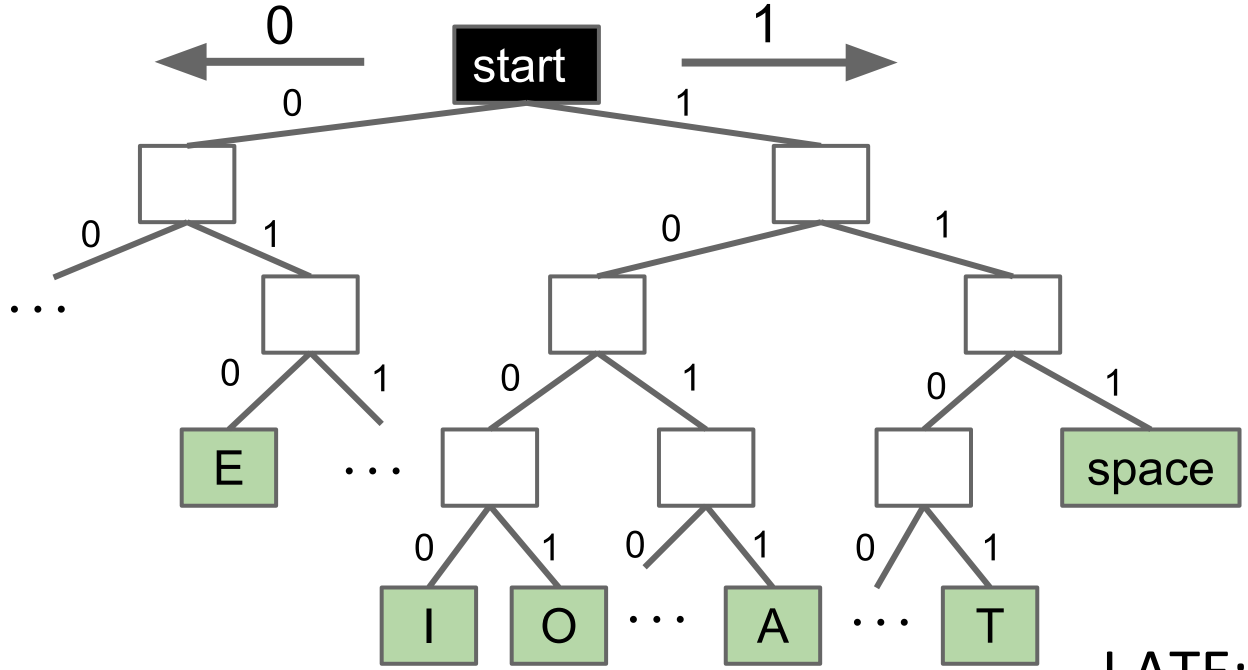
space	1
E	01
T	001
A	0001
O	00001
I	000001
...	



I ATE: 0000011000100101

Prefix-Free Codes [Example 2]

A prefix-free code is one in which no codeword is a prefix of any other. Example for English:



space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	

I ATE: 100011110111101010

Prefix Free Code Design

Observation: Some prefix-free codes are better for some texts than others.

Better for EEEEAT
(8+3+4 = 15 bits).

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

Much worse for JOSH
(25+5+8+10 = 48 bits).

Worse for EEEEAT
(12+4+4 = 20 bits).

space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	

Better for JOSH
(7+4+6+6 = 23 bits).

Observation: It’d be useful to have a procedure that calculates the “best” code for a given text.

Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
    - Left half gets a leading zero. Right half gets a leading one.
    - Repeat.

	Symbol	Frequency
Left half	我	0.35
	爸	0.17
Right half	是	0.17
	李	0.16
	刚	0.15

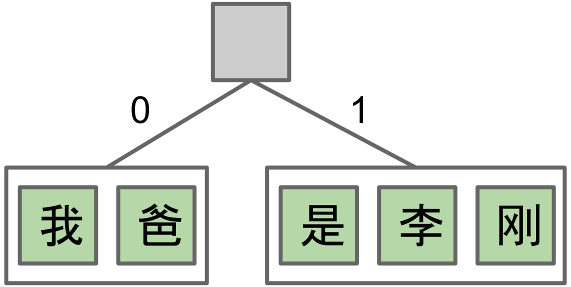
35% of all characters are 我

我 爸 是 李 刚

Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
  - Left half gets a leading zero. Right half gets a leading one.
  - Repeat.

	Symbol	Frequency	Code
Left half	我	0.35	0...
	爸	0.17	0...
Right half	是	0.17	1...
	李	0.16	1...
	刚	0.15	1...



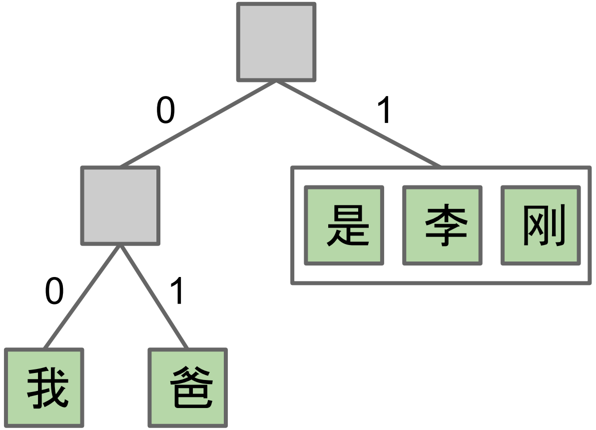
Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
    - Left half gets a leading zero. Right half gets a leading one.
    - Repeat.

Left half

Right half

Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	1...
李	0.16	1...
刚	0.15	1...



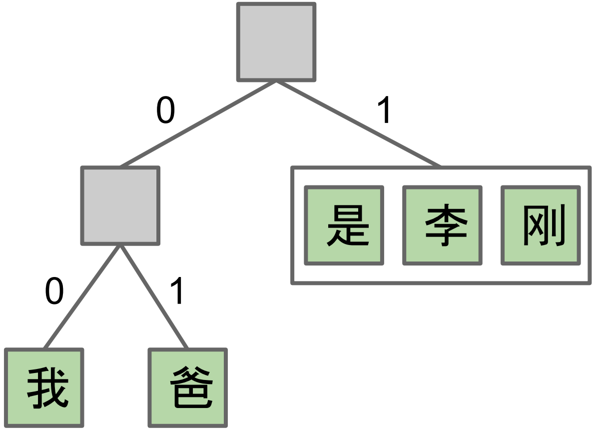
Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
  - Left half gets a leading zero. Right half gets a leading one.
  - Repeat.

Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	1...
李	0.16	1...
刚	0.15	1...

Left half

Right half



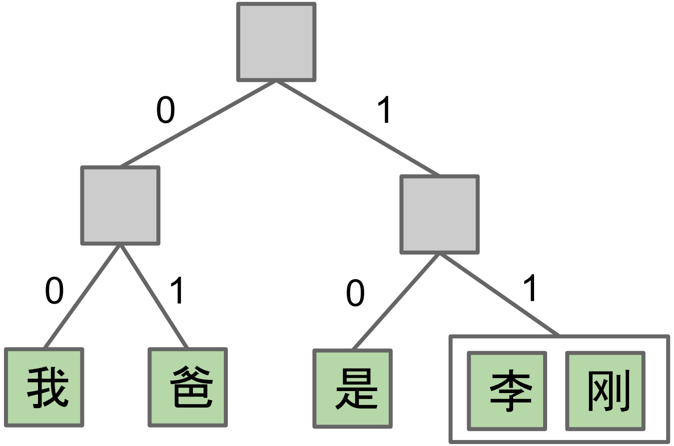
Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
    - Left half gets a leading zero. Right half gets a leading one.
    - Repeat.

Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	11...
刚	0.15	11...

Left half

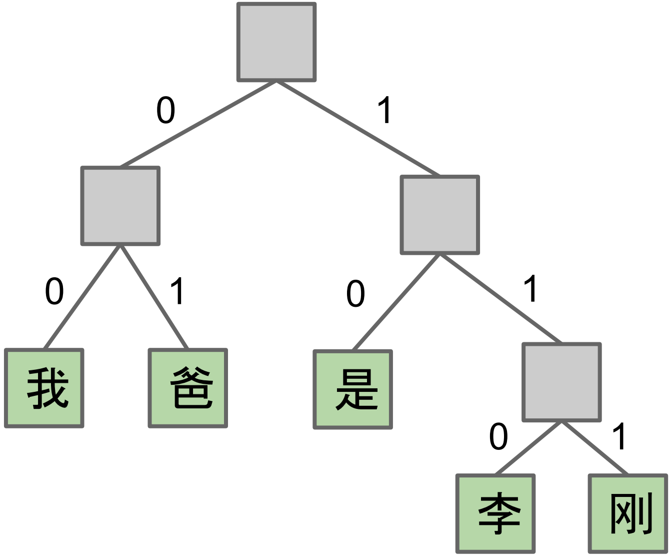
Right half



Code Calculation Approach #1 (Shannon-Fano Coding)

- Count relative frequencies of all characters in a text.
- Split into ‘left’ and ‘right halves’ of roughly equal frequency.
  - Left half gets a leading zero. Right half gets a leading one.
  - Repeat.

Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111



Efficiency Assessment: http://shoutkey.com/deep

How many bits per symbol do we need to compress a file with the character frequencies listed below using the Shannon-Fano code that we created?

A. (2*3 + 3*2) / 5
= 2.4 bits per symbol

B. (0.35 + 0.17 + 0.17) * 2 + (0.16 + 0.15) * 3
= 2.31 bits per symbol

C. Not enough information, we need to know the exact characters in the file being compressed.

Symbol	Frequency	S-F Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111

Efficiency Assessment of Shannon-Fano Coding

How many bits per symbol do we need to compress a file with the character frequencies listed below using the Shannon-Fano code that we created?

B. **(0.35 + 0.17 + 0.17) * 2 + (0.16 + 0.15) * 3 = 2.31 bits per symbol.**

Symbol	Frequency	S-F Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111

Example assuming we have 100 symbols:

- 35 * 2 = 70 bits
- 17 * 2 = 34 bits
- 17 * 2 = 34 bits
- 16 * 3 = 48 bits
- 15 * 3 = 45 bits

Total: 231 bits
231 / 100 = 2.31 bits/symbol

Efficiency Assessment of Shannon-Fano Coding

If we had a file with 350 我 characters , 170 爸 characters , 170 是 characters, 160 李 characters, and 150 刚 characters, how many total bits would we need to encode this file using 32 bit Unicode? Using our Shannon-Fano code?

You don’t need a calculator.

Symbol	Frequency	S-F Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111

2.31 bits per symbol for texts with this distribution

Efficiency Assessment of Shannon-Fano Coding

If we had a file with 350 我 characters , 170 爸 characters , 170 是 characters, 160 李 characters, and 150 刚 characters, how many total bits would we need to encode this file using 32 bit Unicode? Using our Shannon-Fano code?

1000 total characters.

Space used:

- 32 bit Unicode: 32,000 bits.
- S-F code: 2,310 bits.

Our code is 14 times as efficient!

- Can only encode strings with these 5 symbols.

Symbol	Frequency	S-F Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111

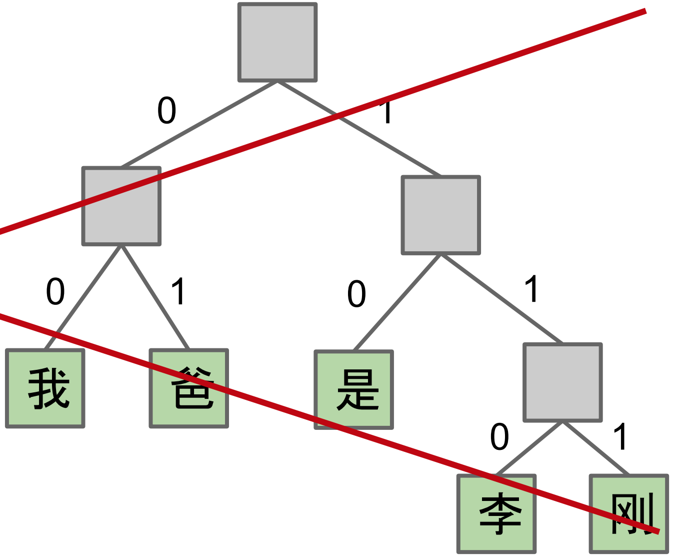
2.31 bits per symbol for texts with this distribution

Code Calculation Approach #1 (Shannon-Fano Coding)

Shannon-Fano coding is NOT optimal. Does a good job, but possible to find ‘better’ codes (see CS170).

- Optimal solution assigned (and solved) as alternative to a final exam: http://www.huffmancoding.com/my-uncle/scientific-american

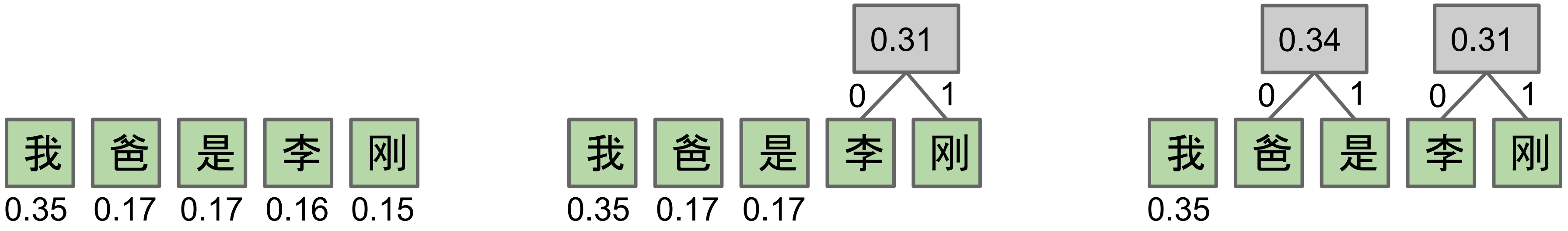
Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111



Code Calculation Approach #2: Huffman Coding

As before, calculate relative frequencies.

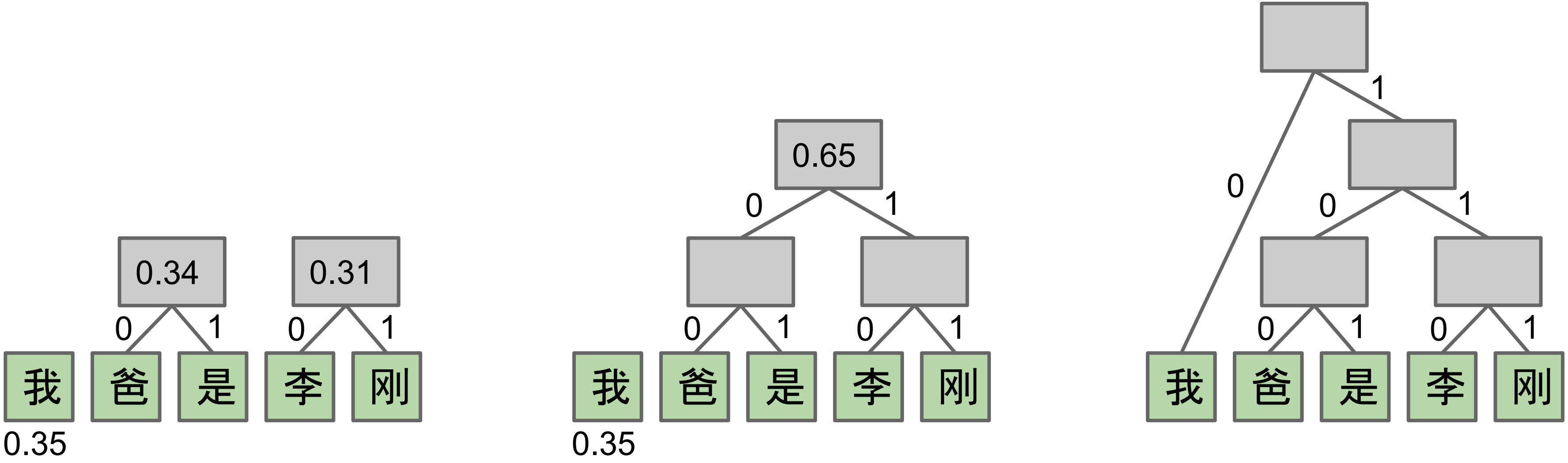
- Assign each symbol to a node with weight = relative frequency.
- Take the two smallest nodes and merge them into a super node with weight equal to sum of weights.
- Repeat until everything is part of a tree.



Code Calculation Approach #2: Huffman Coding

As before, calculate relative frequencies.

- Assign each symbol to a node with weight = relative frequency.
- Take the two smallest nodes and merge them into a super node with weight equal to sum of weights.
- Repeat until everything is part of a tree.



Huffman vs. Shannon-Fano

Shannon-Fano code below results in an average of 2.31 bits per symbol, whereas Huffman is only 2.3 bits per symbol.

- Huffman coded file is 0.35*1 + 0.65*3 = 2.3 bits per symbol.

Symbol	Frequency	S-F Code	Huffman Code
我	0.35	00	0
爸	0.17	01	100
是	0.17	10	101
李	0.16	110	110
刚	0.15	111	111

Strictly better than Shannon-Fano coding. There is NO downside to Huffman coding instead.

Prefix-Free Codes

Question: For encoding (bitstream to compressed bitstream), what is a natural data structure to use? Assume characters are of type Character, and bit sequences are of type BitSequence.

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	0111

I ATE: 0000011000100101

I ATE: 1000111101111101010

Prefix-Free Codes

Question: For encoding (bitstream to compressed bitstream), what is a natural data structure to use? chars are just integers, e.g. ‘A’ = 65.

- Array of BitSequence[], to retrieve, can use character as index.
- How is this different from a HashMap<Character, BitSequence>? Lookup in a hashmap consists of:
  - Compute hashCode.
  - Mod by number of buckets.
  - Look in a linked list.

Compared to HashMaps, Arrays are faster (just get the item from the array) and only use more memory if some characters in the alphabet are unused.

I ATE: 0000011000100101

I ATE: 100011110111101010

Prefix-Free Codes

Question: For decoding (compressed bitstream back to bitstream), what is a natural data structure to use?

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	0111

I ATE: 0000011000100101

I ATE: 1000111101111101010

Prefix-Free Codes

Question: For decoding (compressed bitstream back to bitstream), what is a natural data structure to use?

- We need to lookup **longest matching prefix**, an operation that Tries excel at.

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	0111

I ATE: 0000011000100101

I ATE: 100011110111101010

Huffman Coding in Practice

Huffman Compression

Two possible philosophies for using Huffman Compression:

1. For each input type (English text, Chinese text, images, Java source code, etc.), assemble huge numbers of sample inputs for that category. Use each corpus to create a standard code for English, Chinese, etc.
2. For every possible input file, create a unique code just for that file. Send the code along with the compressed file.

What are some advantages/disadvantages of each idea? Which is better?

```
$ java HuffmanEncodePh1 ENGLISH mobydick.txt
$ java HuffmanEncodePh1 BITMAP horses.bmp
```

```
$ java HuffmanEncodePh2 mobydick.txt
$ java HuffmanEncodePh2 horses.bmp
```

Huffman Compression (Your Answers)

Two possible philosophies for using Huffman Compression:

1. Build one corpus per input type.
2. For every possible input file, create a unique code just for that file. Send the code along with the compressed file.

What are some advantages/disadvantages of each idea? Which is better?

- #2 has the advantage that the code is tailor made and more efficient. Weird books like Gadsby which has no letter ‘e’.
- #2 has the advantage that it feels kind of encryptedy?
- #2 would additional to build codeword table.
- #2 has the disadvantage that the code itself takes up space, which is what we care about.
- #1 you can have one translator, but #2 takes many translators (sending the code makes it so you really just need one decoder, though, not a big deal)

Huffman Compression

Two possible philosophies for using Huffman Compression:

1. For each input type (English text, Chinese text, images, Java source code, etc.), assemble huge numbers of sample inputs for that category. Use each corpus to create a standard code for English, Chinese, etc.
2. For every possible input file, create a unique code just for that file. Send the code along with the compressed file.

In practice, Philosophy 2 is used in the real world.

Huffman Compression Example [Demo Link]

Given **input text**: 我我刚刚刚是我是我刚李刚我李是爸李爸李是李我我李刚是我是刚爸是刚我爸我李是是李是我我刚爸是李我我我是爸我是我爸是我爸是我是刚我是爸刚爸我刚我我刚爸我我爸我刚爸爸李李李李我我爸李我我刚爸李我我李我爸我我

See optional textbook for how to do this.

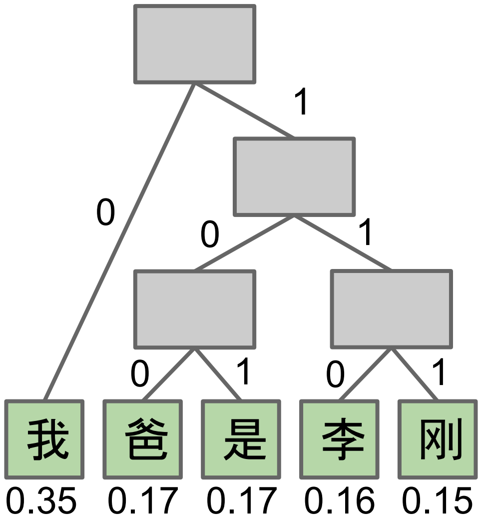
Step 1: Count frequencies.

Step 2: Build encoding array and decoding trie.

Step 3: Write decoding trie to output.huf.

Step 4: Write codeword for each symbol to output.huf.

Output bits: $\underbrace{\texttt{010101010101001...}}_{\text{Decoding Trie}}\underbrace{\texttt{00111111111101...}}_{\text{Codewords}}$



Decoding Trie

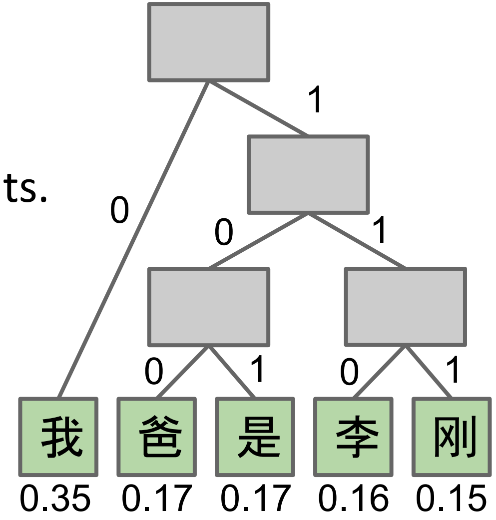
Huffman Decompression Example [Demo Link]

Given **input bitstream**: $\underbrace{\texttt{010101010101001...}}_{\text{Decoding Trie}}\underbrace{\texttt{001111111111101}}_{\text{Codewords}}\texttt{...}$

Step 1: Read in decoding trie.

Step 2: Use codeword bits to walk down the trie, outputting symbols every time you reach a leaf.

- Note: Symbols are really just bits!
  - 我 is 111001011000100010010001 in Unicode.
  - “Outputting 我” actually means outputting these 32 bits.



Output symbols: 我我刚刚刚是...

- Output bits: 111001011000100010010001...

Huffman Coding Summary

Given a file X.txt that we'd like to compress into X.huf:

- Consider each b-bit symbol (e.g. 8-bit chunks, Unicode characters, etc.) of X.txt, counting occurrences of each of the 2^b possibilities, where b is the size of each symbol in bits.
- Use Huffman code construction algorithm to create a decoding trie and encoding map. Store this trie at the beginning of X.huf.
- Use encoding map to write codeword for each symbol of input into X.huf.

To decompress X.huf:

- Read in the decoding trie.
- Repeatedly use the decoding trie's longestPrefixOf operation until all bits in X.hug have been converted back to their uncompressed form.

See Huffman.java for an example implementation on 8-bit symbols.

Compression Algorithms (General)

The big idea in Huffman Coding is representing common symbols with small numbers of bits.

Many other approaches, e.g.

- Run-length encoding: Replace each character by itself concatenated with the number of occurrences.
  - Rough idea: XXXXXXXXXYYYYXXXXX -> X10Y4X5
- LZW: Search for common repeated patterns in the input. See extra slides.

General idea: Exploit redundancy and existing order inside the sequence.

- Sequences with no existing redundancy or order may actually get enlarged.

Compression Theory

Comparing Compression Algorithms

Different compression algorithms achieve different compression ratios on different files.

We'd like to try to compare them in some nice way.

- To do this, we'll need to refine our model from slide 3 to be a bit more sophisticated.

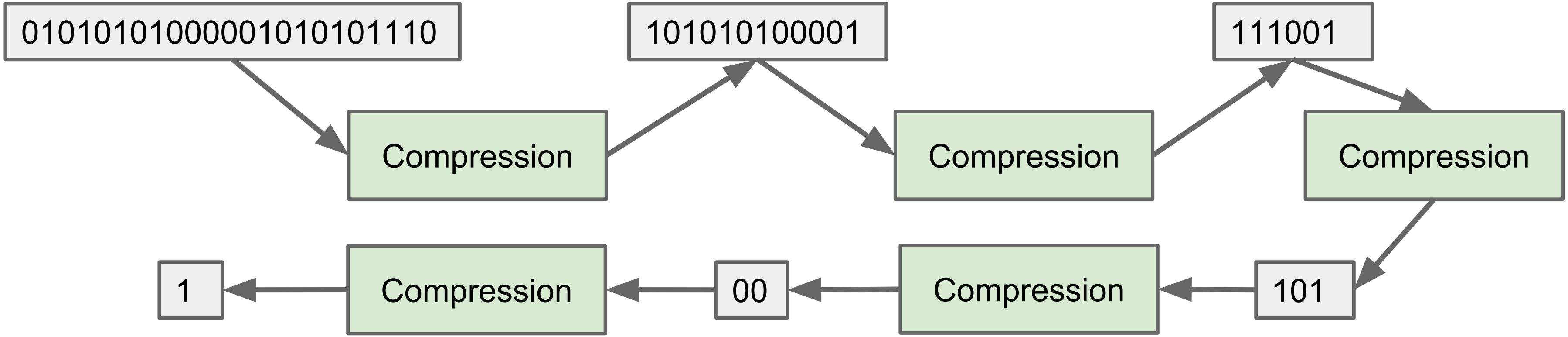
Let's start with a straightforward puzzle.

SuperZip

Suppose an algorithm designer says their algorithm SuperZip can compress any bitstream by 50%. Why is this impossible?

Universal Compression: An Impossible Idea

Argument 1: If true, they'd be able to compress any bitstream down to a single bit. Interpreter would have to be able to do the following (impossible) task for ANY output sequence.



01010101000001010101110

Compression

101010100001

Compression

111001

Compression

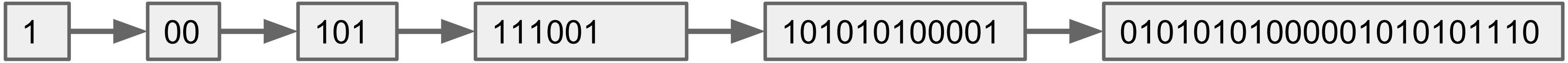
101

Compression

00

Compression

1



1

00

101

111001

101010100001

01010101000001010101110

Universal Compression: An Impossible Idea

Argument 2: There are far fewer short bitstreams than long ones. Guaranteeing compression even once by 50% is impossible. Proof:

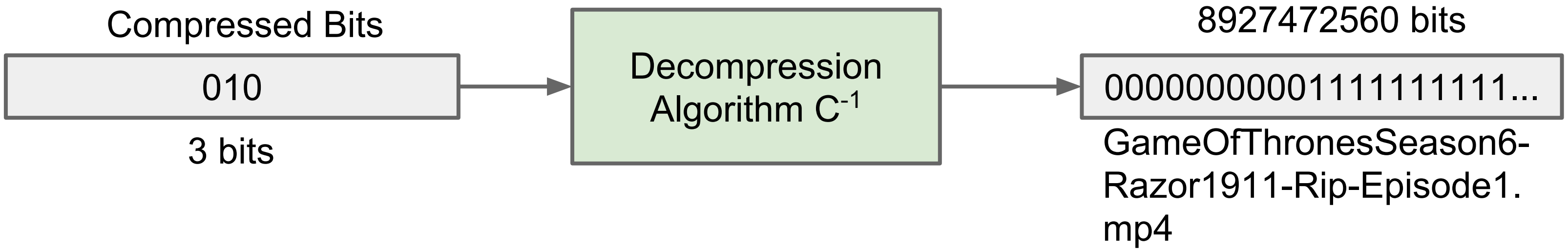
- There are 2^{1000} 1000-bit sequences.
- There are only $1+2+4+...+2^{500} = 2^{501} - 1$ bit streams of length ≤ 500.
- In other words, you have 2^{1000} things and only $2^{501} - 1$ places to put them.
- Of our 1000-bit inputs, only roughly 1 in 2^{499} can be compressed by 50%!

A Sneaky Situation

Universal compression is impossible, but the following example implies that comparing compression algorithms could still be quite difficult.

Suppose we write a special purpose compression algorithm that simply hardcodes small bit sequences into large ones.

- Example, represent GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4 as 010.

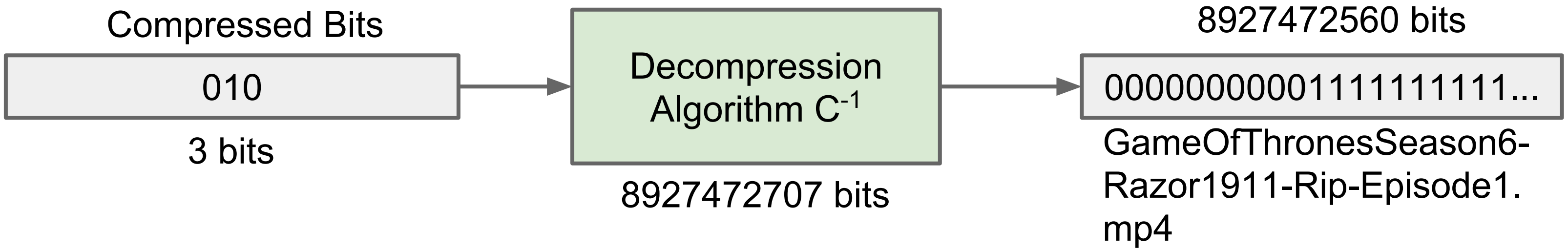


A Sneaky Situation

Suppose we write a special purpose compression algorithm that simply hardcodes small bit sequences into large ones.

- Example, represent GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4 as 010.

To avoid this sort of trickery, we should include the bits needed to encode the decompression algorithm itself.



Compressed Bits

010

3 bits

Decompression Algorithm C^{-1}

8927472707 bits

8927472560 bits

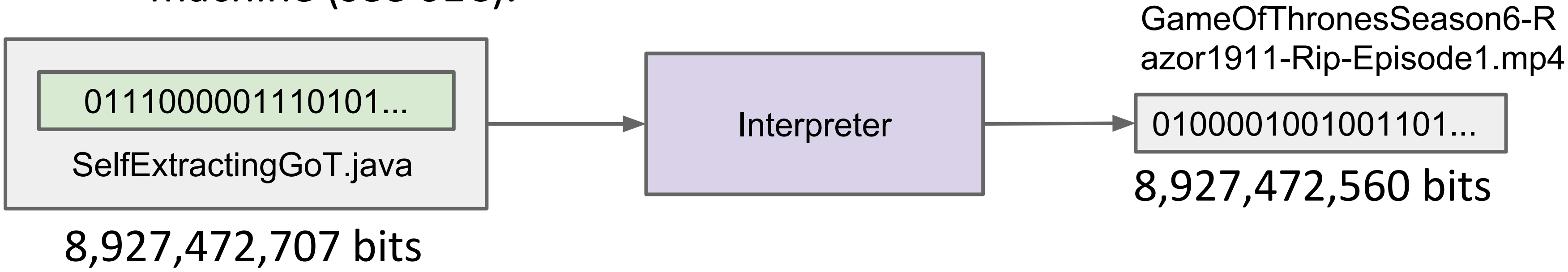
0000000000111111111...

GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4

Compression Model 2: Self-Extracting Bits

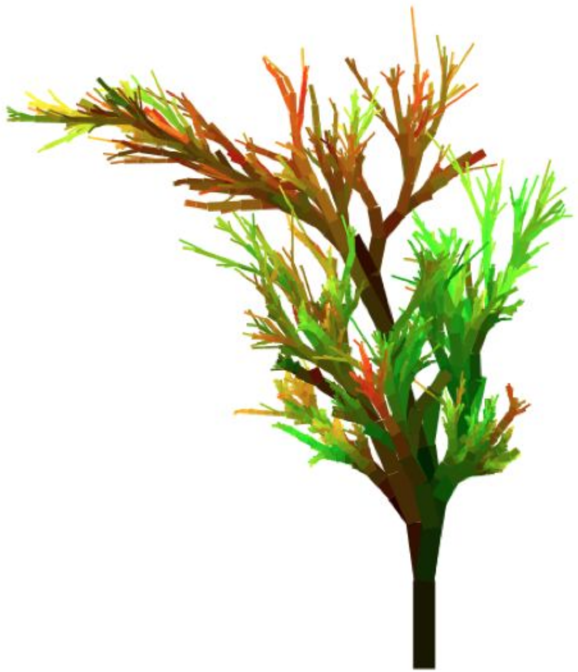
As a model for the decompression process, let’s **treat the algorithm and the compressed bitstream as a single sequence of bits.**

- If you want a concrete idea to hold on to, imagine storing the compressed bitstream as a byte[] variable in a .java file. We’ll show an example on the coming slides involving compressing an image.
- Can think of the algorithm + compressed bitstream as an input to an interpreter. Interpreter somehow executes those bits (see 61A)
  - At the very “bottom” of these abstractions is some kind of physical machine (see 61C).



HugPlant

Huffman Coding can be used to compress any data, not just text. In bitmap format, the plant below is simply the stream of bits shown on the right.



Original Uncompressed Bits B

```
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 ...
```

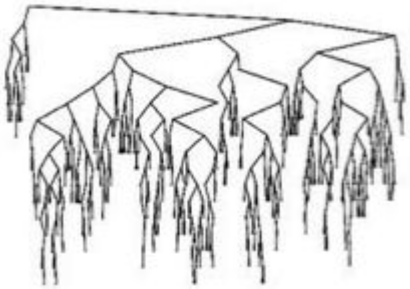
Total: 8389584 bits

HugPlant Compressed

```
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.java compress()



```
74 68 65 20 70 61 73 73 63 6F 64 65 20 69 73 20
68 75 67 39 31 38 32 37 78 79 7A 2E 65 75 7a c0
09 eb cd d4 2a 55 9f d8 98 d1 4e e7 97 56 58 68
0c 7a 43                            7 bc 26 01
e0 92 28                            f 0e 87 bd
87 9e 03                            1 b5 79 13
9b 95 d5                            9 c0 e6 53
c7 cb dd                            a 74 34 3a
a9 c1 ca                            4 3a 5c 5a
62 e9 2f                            7 4e 50 be
c0 15 1b                            4 e2 23 a2
b6 84 22                            2 0f 2d bd
e5 81 f4 c6 de 15 59 f1 68 a4 a5 88 16 b0 7f bf
8a 1d                                     ce 12
57 23                                     d6 9b
12 49                                     dd e3
dc 1c 7f c4 a4 69 77 6e 5e 60 db 5a 69 01 95 c8
d7 2e 57 62 b7 8e 5c 51 f9 70 55 1b 7c ba 68 bc
```

Decoding Trie: 2560 bits

Total: 1994024 bits

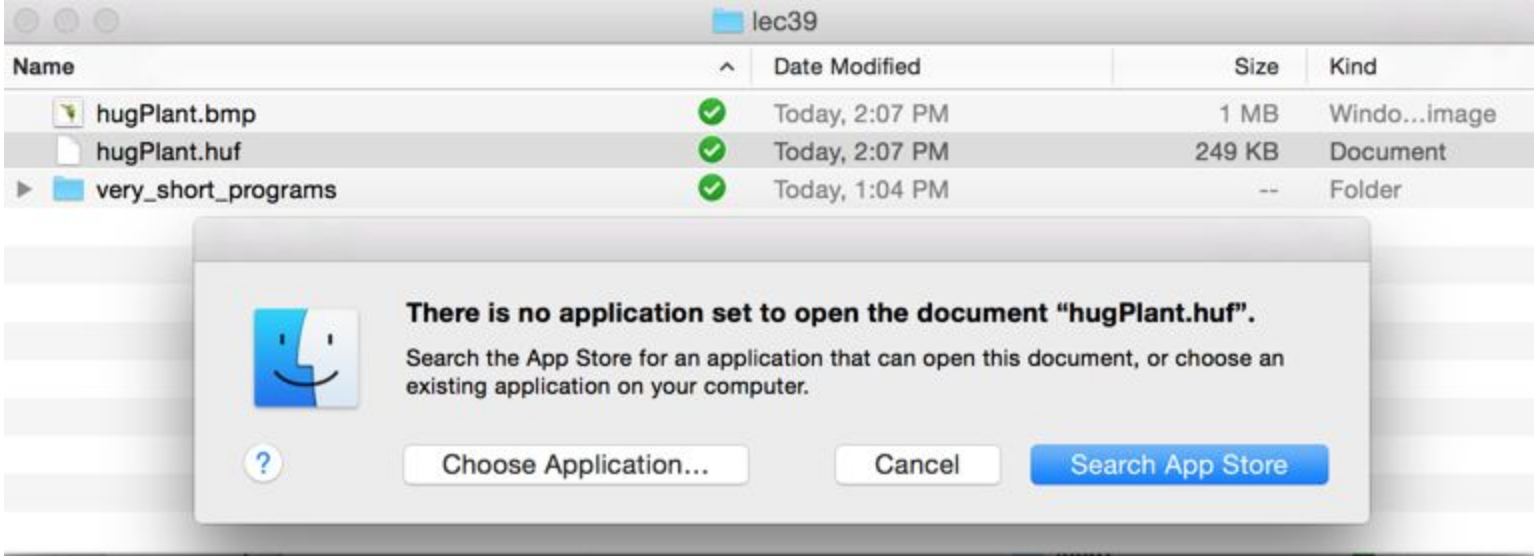
```
00 20 00 f4 c3 b7 6d c2 31 24 92 dc 24 a7 c9 25 ae 24
b5 c4 85 88 40 be c4 92 46 25 79 2f c4 af 25 f8 92 49
24 92 64 c9 92 49 30 b1 24 92 49 24 2c 49 24 92 49 0b
12 49 24 92 42 c4 92 49 24 92 49 ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff 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                                        f ff
ff ff ff                                        f ff
ff ff ff                                        f ff
ff ff ff                                        f ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff 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: 1991464 bits

Opening a .huf File

Of course, major operating systems have no idea what to do with a .huf file.

- Have to send over the 43,400 bits of Huffman.java code as well.
- Total size (including .java file): 2,037,424 bits.



Name	Date Modified	Size	Kind
hugPlant.bmp	Today, 2:07 PM	1 MB	Windo...image
hugPlant.huf	Today, 2:07 PM	249 KB	Document
very_short_programs	Today, 1:04 PM	--	Folder

There is no application set to open the document "hugPlant.huf".

Search the App Store for an application that can open this document, or choose an existing application on your computer.

Or an Even Simpler View

To keep things conceptually simpler, let’s package the compressed data plus decoder into a single self-extracting .java file.

- Bitstream on the left generates bitstream on the right.

`SelfExtractingHugPlant.java`

```
70 75 62 6c 69 63 20 63 6c 61 73 73 20 53 65 6c
66 45 78 74 72 61 63 74 69 6e 67 48 75 67 50 6c
61 6e 74 20 7b 0d 0a ...   74 68 65 20 70 61 73
73 63 6F 64 65 20 69 73 20 68 75 67 39 31 38 32
37 78 79 7A 2E 65 75 7a c0 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 ...   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                          f ff ff ff
ff ff ff ff                          f ff ff ff
ff ff ff ff                          f ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ...
```

2,037,424 bits

`HugPlant.bmp`

```
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 00 00 00 00 00 00 00 00 00 00 01 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff f                      f ff ff ff ff
ff ff ff ff ff f                      f ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff...
```

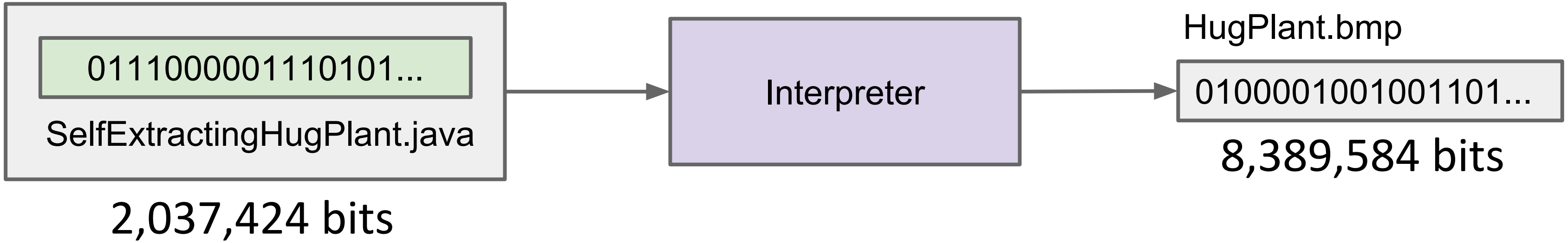
8,389,584 bits

Compression Model #2: Self-Extracting Bits

As a model for the decompression process, let's **treat the algorithm and the compressed bitstream as a single sequence of bits.**

- We've now seen an example: SelfExtractingHugPlant.

Will discuss the implications of this model next time.



LZW Style Compression (Extra)

Thought Experiment

How might we compress the following bitstreams (underlines for emphasis only)?

- B=”aababcabcdabcdeabcdefabcdefgabcdefgh”?
- B=”ababababababababababababababab”?
- B=”aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”?

The LZW Approach

Key idea: Each ***codeword*** represents multiple symbols.

- Start with ‘trivial’ codeword table where each codeword corresponds to one ASCII symbol.
- Every time a codeword X is used, record a new codeword Y corresponding to X concatenated with the next symbol.

Demo Example: http://goo.gl/68Dncw

LZW

Named for inventors Limpel, Ziv, Welch.

- Related algorithm used as a component in many compression tools, including .gif files, .zip files, and more.
- Once a hated algorithm because of attempts to enforce licensing fees. Patent expired in 2003.

Our version in lecture is simplified, for example:

- Assumed inputs were ≤ 0x7f (7 bit input) and also provided 8 bit outputs (real LZW can have variable length outputs).
- Didn’t say what happens when table is full (many variants exist).

LZW

Neat fact: You don’t have to send the codeword table along with the compressed bitstream.

- Possible to reconstruct codeword table from C(B) alone.

LZW decompression example:

http://goo.gl/fdYU9C

Side Trip: Lossy Compression

Lossy Compression

Most media formats lose information during compression process:

- .JPEG images.
- .MP3 audio.
- .MP4 video.

Why?

- MP4 video: 1920 x 1080 pixels, 60 times per second, 24 bits per pixel: 0.37 gigabytes per second, 1,343 gigabytes per hour.
- Downloading a movie: 30 days at 1 MB/second.

Lossy Compression

Basic idea: Throw away information that human sensory system doesn't care about.

Examples:

- Audio: High frequencies.
- Video: Subtle gradations of color (low frequencies).

See EE20 (or perhaps 16A/16B?) for more.

Summary

Compression: Make common bitstreams more compact.

Huffman coding:

- Represents common symbols as codeword with fewer bits.
- Uses something like Map<Character, BitSeq> for compression.
- Uses something like TrieMap<Character> for decompression.

LZW:

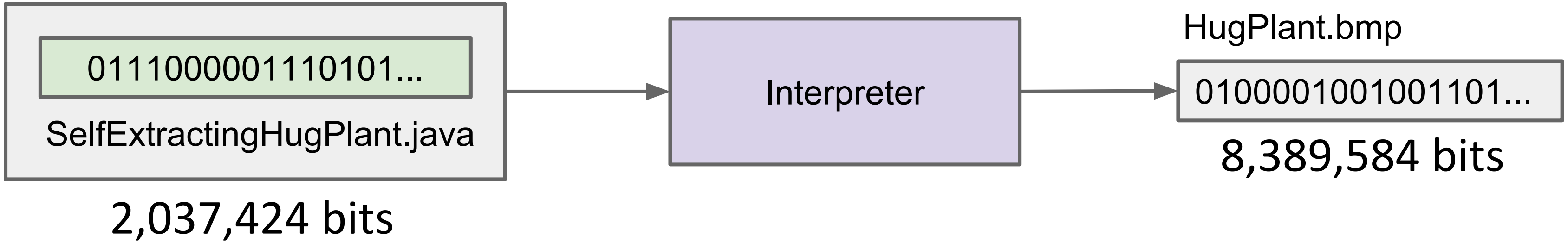
- Represents multiple symbols with a single codeword.
- Uses something like TrieMap<Integer> for compression.
- Uses something like Map<Character, SymbolSeq> for decompression.

Slides Moved to Next Lecture

TAANSTAFL: There ain’t no such thing as a free lunch.

Compression can be thought of as a ‘remapping’ of bitstreams.

- For very bitstream that has a “short” representation, some other bitstream must have a “long” representation by C.



01110000011101010...

SelfExtractingHugPlant.java

2,037,424 bits

Interpreter

HugPlant.bmp

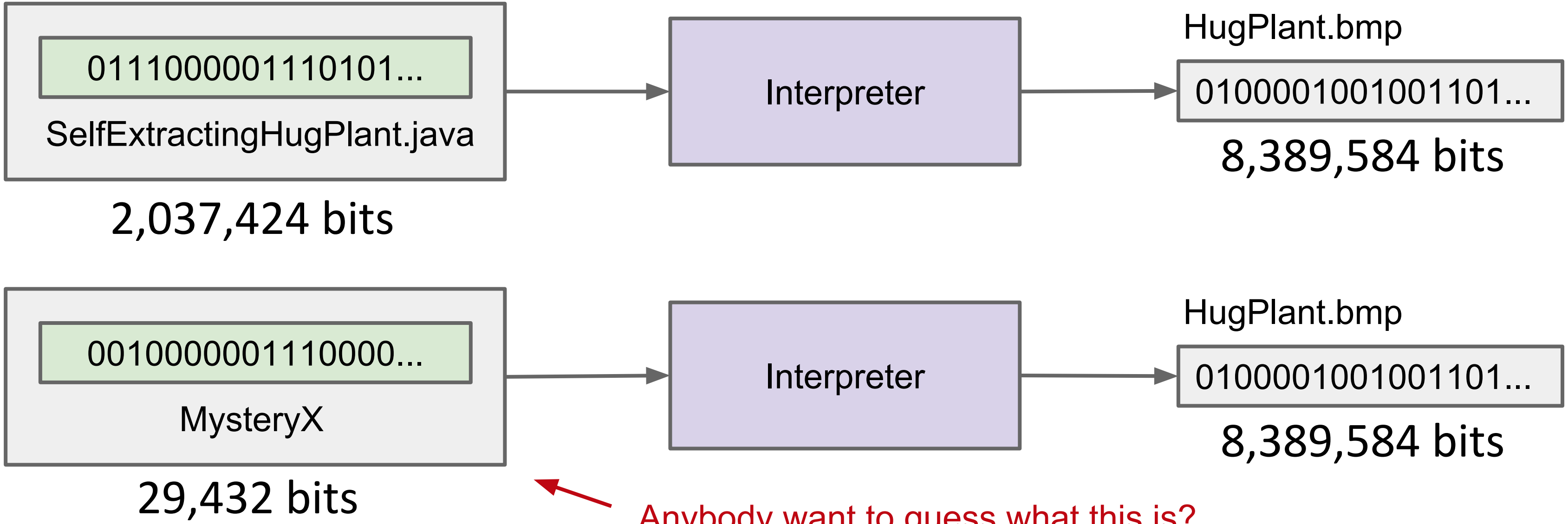
01000010010011010...

8,389,584 bits

Even Better Compression

Compression ratio of 25% is certainly very impressive, but we can do much better. MysteryX achieves a 0.35% compression ratio!

- Of the $2^{8389584}$ possible bit streams of length 8389584, only one in $2^{8360151}$ can be generated by our computer using an input of length 29,432 bits.



Anybody want to guess what this is?

MysteryX: HugPlant.java

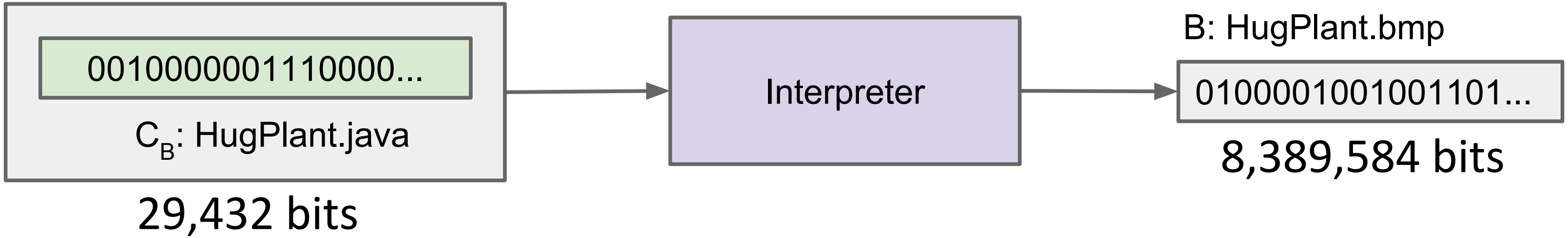
MysteryX is just HugPlant.java, the piece of code that I used to generate the .bmp file originally.

```
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                                  4 20 7b
0a 09 2f                                  c 20 74
6f 20 70                                  e 75 67
20 74 6f                                  f 75 72
20 70 72                                  1 74 65
20 73 74                                  5 20 73
63 61 6c                                  e 30 3b
0a 0a 09                                  1 74 69
63 20 69                                  2 56 61
6c 75 65 28 69 6e 74 20 6f 6c 64 56 61 6c 2c
```

```
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)
```

Interesting question:

- Given a target bitstream B, what is the shortest bitstream C_B that outputs B?
- More next time...



0010000001110000...

C_B: HugPlant.java

29,432 bits

Interpreter

B: HugPlant.bmp

0100001001001101...

8,389,584 bits