Lecture 38

Compression

CS61B, Spring 2024 @ UC Berkeley

Slide Credit: Josh Hug

Today's Goal: Compression

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Zip Files, How Do They Work?

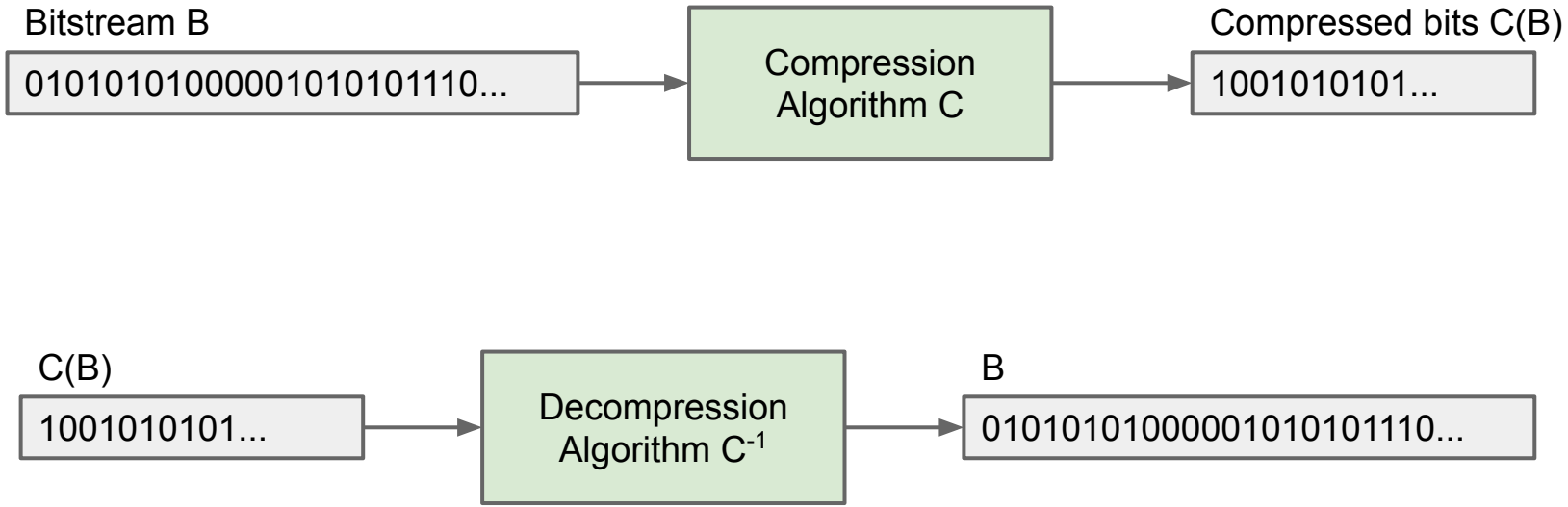
```
$ zip mobydick.zip mobydick.txt
  adding: mobydick.txt (deflated 59%)
$ ls -l
-rw-rw-r-- 1 jug jug 643207 Apr 24 10:55 mobydick.txt
-rw-rw-r-- 1 jug jug 261375 Apr 24 10:55 mobydick.zip
```

Size in Bytes

File is unchanged by zipping / unzipping.


```
$ 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
$
```

Compression Model #1: Algorithms Operating on Bits



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

- Formally, C needs to be **injective**: If A != B, then C(A) != C(B)
- Text files often compressible by 70% or more.

Information Theory

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Information Theory

Before we talk about compression, it's useful to see how much "information" is in our data.

Intuitively, the compressed bitstring should convey the same amount of information, and the more information we have, the harder it should be to compress our bitstring.

One useful test of the amount of information something has is how easy it is to memorize it; less information = easier to memorize (generally).

Memorize a 10-Character String

BXX ONHP WTP

Memorize a String

Memorize a 47-Character String

ONE A POEM A RAVEN
MIDNIGHTS SO DREARY TIRED
AND WEARY

Memorize a String

Memorize a 10000-Character String

Shannon Entropy

Recall: In CS, a "bit" is a single binary digit (1 or 0), which (most) computers use to store information.

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

However, the amount of *information* stored in data can be much less than the raw number of bits

- In general, the more predictable a dataset is, the fewer bits of data it actually contains

Shannon Entropy

The **Shannon Entropy** of a dataset is a measure of how predictable a dataset is

- Formal definition: E(-log(p(X))), where E is expected value and p(X) is the probability that X is a given value (averaged over all possible values of X)
- Informally, it's a measure of how many possible strings exist with that characteristic
  - Ex. If a string has 10 bits of entropy, there are ~2^10 = 1024 possible strings it could have been.
  - Ex. If we take 10-char random strings of English letters, there are 26^10 possible such strings, which is about 47 bits of entropy ($\log_2(26) \sim 4.7004$)
  - On the other extreme, if we have a 10000-char string containing only the same letter, there's only 26 possible such strings, so we only have 4.7 bits of entropy there
  - English text is somewhere in the middle: it's not completely predictable, but not completely random

Measuring the Shannon Entropy of Standard English

WH_N _R_T__G _NG__SH T_X_

__ST C__R__T__S C_N __

OM___ED W___O__ L_S_NG

M__N_NG

WHEN WRITING ENGLISH TEXT MOST CHARACTERS CAN BE OMITTED WITHOUT LOSING MEANING

Shannon Entropy

Through experiments such as on the previous page, it was determined that standard English has ~1 bit of entropy per character. (More info at http://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf)

So a 47-char string of English text should have around 47 bits of entropy (about the same as the 10-char string of random English letters)

This means that (in theory) an optimal compression algorithm should be able to compress a 47-char string of English text to (on average) 47 bits (87% reduction).

- In practice, hard to get to that theoretical limit
  - Just looking at the relative frequencies of English characters gives us a Shannon entropy of 4.1 bits, and looking at relative word frequencies brings us to 2.6 bits.

As we compress data, the entropy of the data is constant. Thus, the more compressed our data becomes, the more uniformly distributed the underlying bits should become.

Prefix Free Codes

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Increasing Optimality of Coding

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

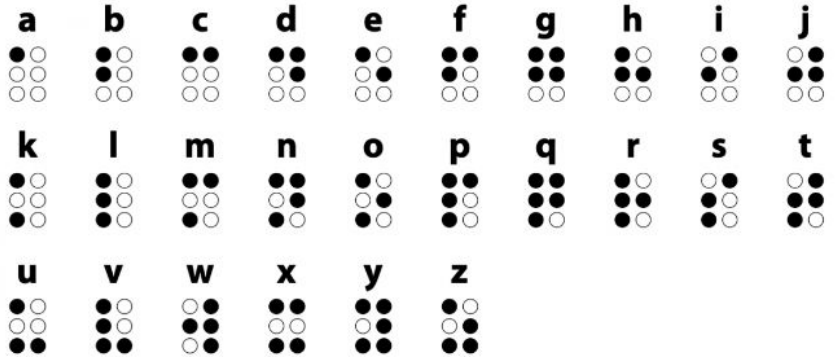
word	binary	hexadecimal
dog	01100100 01101111 01100111	64 6f 67

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.***

Braille

One option is to use Braille:



If we treat a black/white dot as 1/0, we can store each letter in 6 bits instead of 8

Going further, we can get to 5 bits with this, **but no further**. Why?

- 26 letters, 4 bits can only handle 16 letters at a time

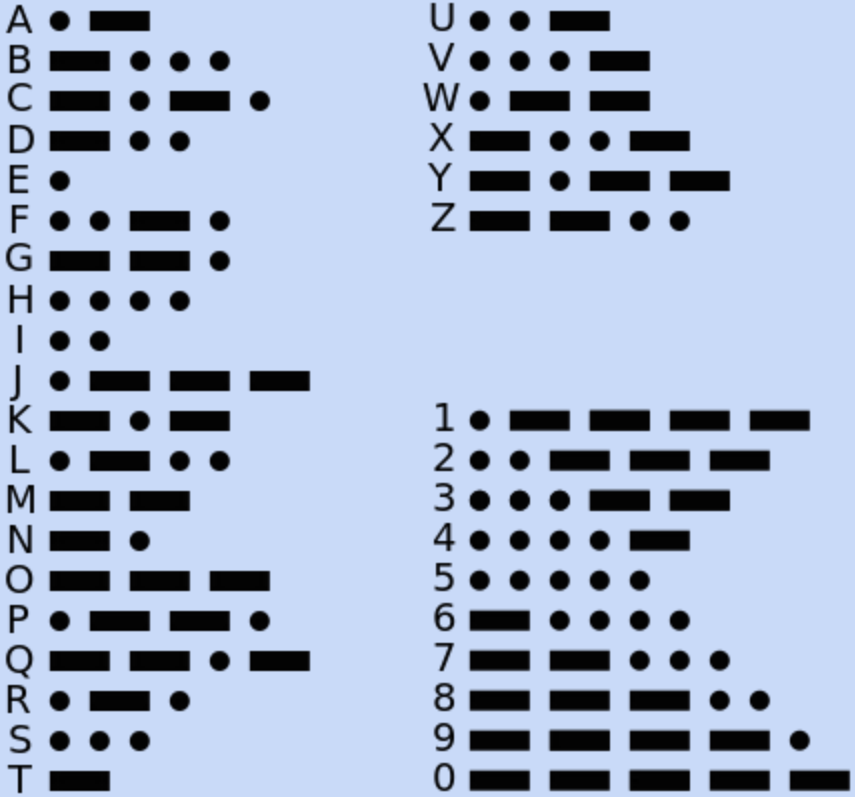
One way forward: use *variable lengths of bits* for each letter.

- Can you think of a cipher that does that?

More Code: Mapping Alphanumeric Symbols to Codewords

Example: Morse code.

- Dash = 1, dot = 0
- Each letter is between 1 and 4 bits
- Useful: More common letters are shorter
  - So average length of each letter is as short as possible
- Problem: What is – – • – – •?



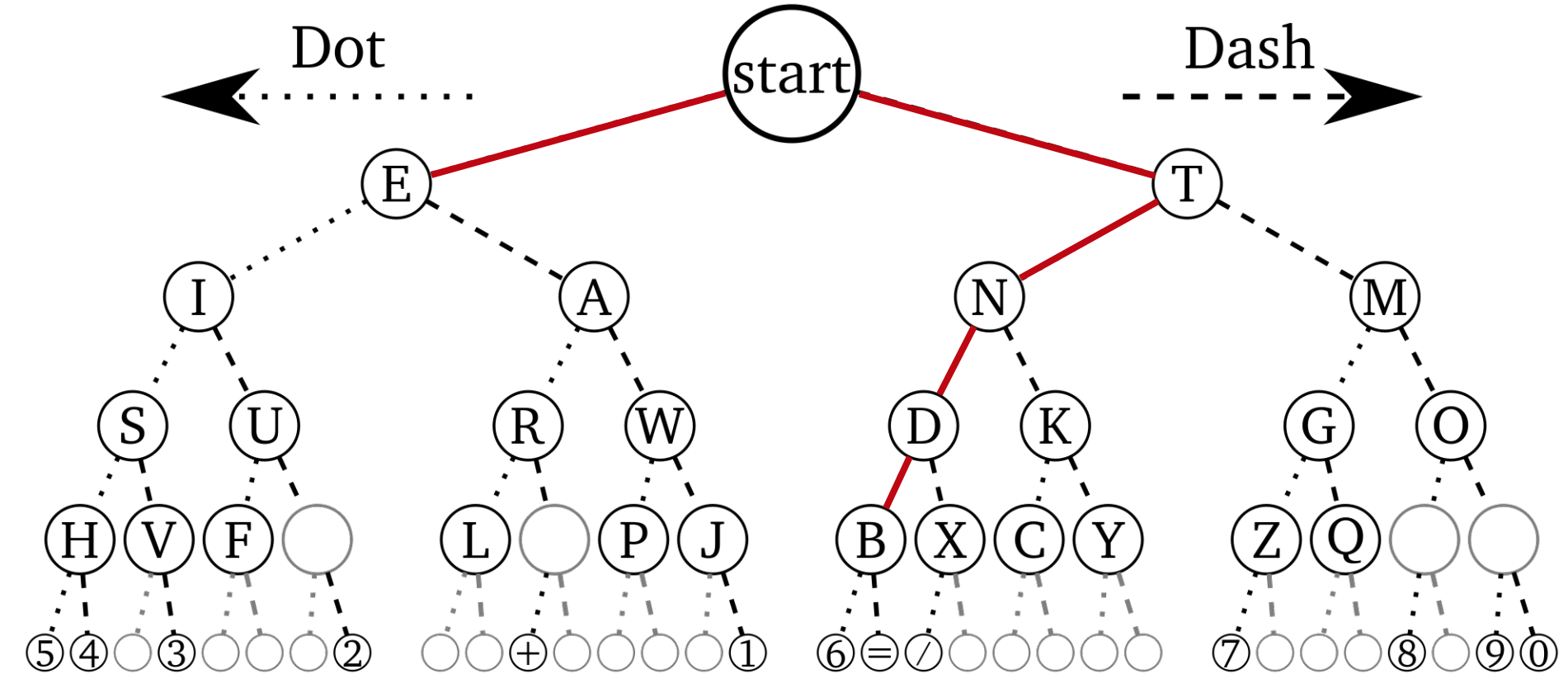
More Code: Mapping Alphanumeric Symbols to Codewords

Example: Morse code.

- What is – – • – – •? It's ambiguous!
  - MEME
  - GG
  - MATE
  - MAN
- Operators pause between codewords to avoid ambiguity.
  - Pause acts as a 3rd symbol.
- For those who are curious, the most ambiguous string I could find is – • • • • – • • • •, which has 23 English representations

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

Morse Code (as a Tree)

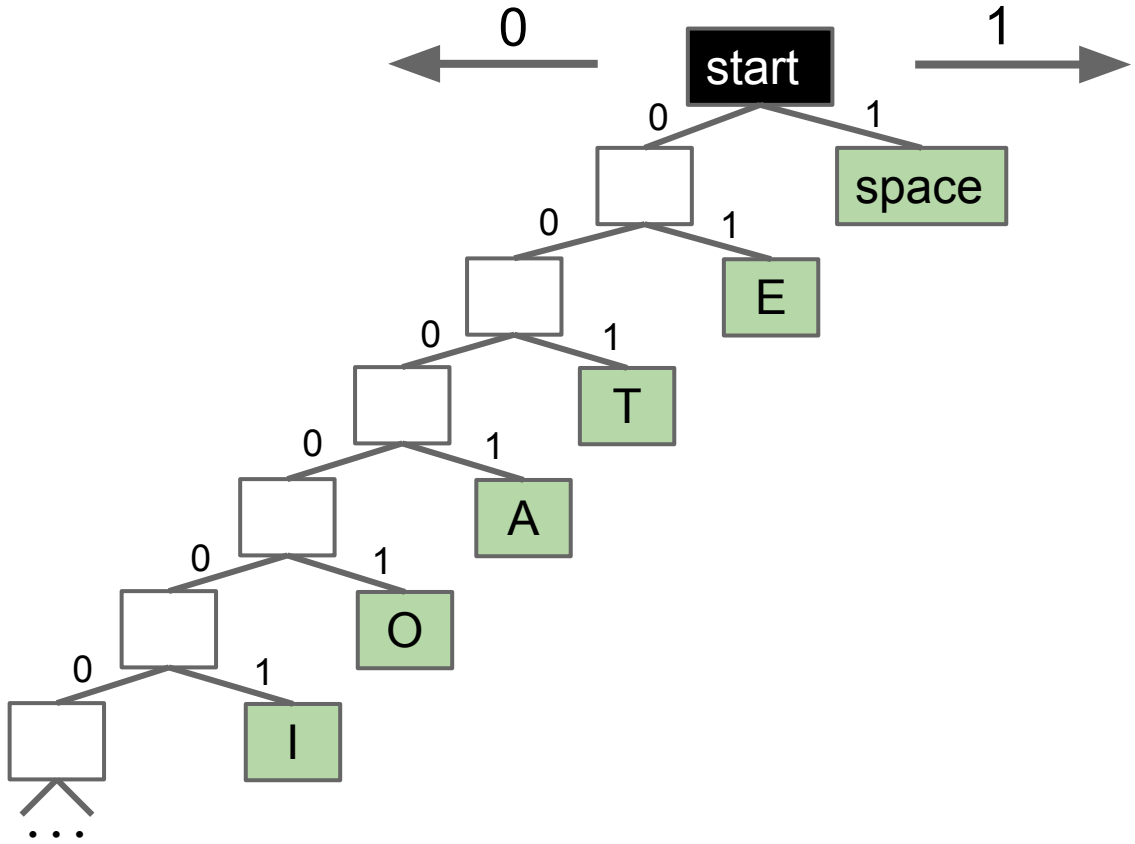


From Wikimedia

Prefix-Free Codes [Example 1]

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

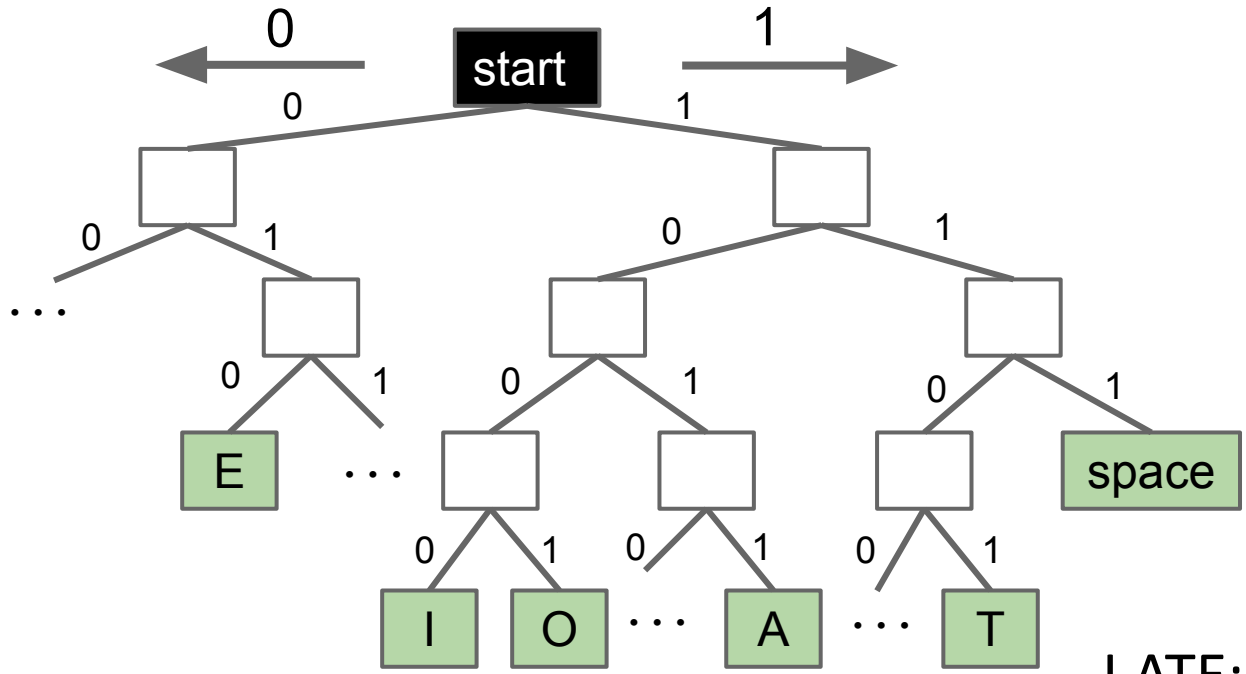
space	1
E	01
T	001
A	0001
O	00001
I	000001
...	



I ATE: 000001100010010 1

Prefix-Free Codes [Example 2]

With a prefix-free code, we are guaranteed to have at most one valid way of interpreting a bitstring (no ambiguous strings!)



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

I ATE: 10001111010111101010

Prefix Free Code Design

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

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

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

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

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

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

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.

Shannon Fano Codes

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Code Calculation Approach #1 (Shannon-Fano Coding)

- Main idea: Since we want to maximize entropy per bit, we want ~50% 0s and ~50% 1s
- 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.

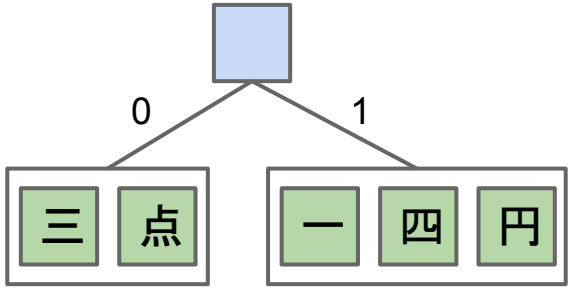
	Symbol	Frequency
Left half	三	0.35
	点	0.17
Right half	一	0.17
	四	0.16
	円	0.15

35% of all characters are 三

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	三	0.35	0...
	点	0.17	0...
Right half	一	0.17	1...
	四	0.16	1...
	円	0.15	1...



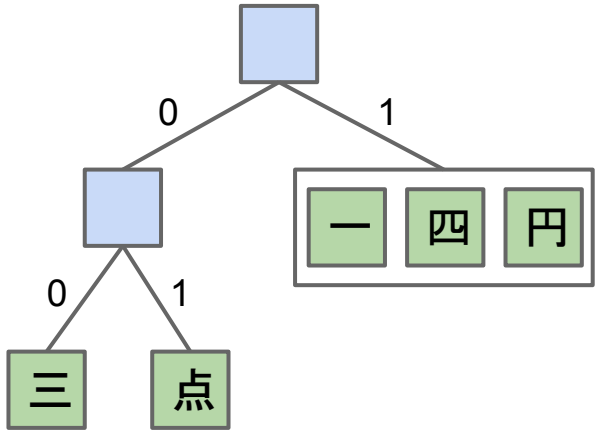
Code Calculation Approach #1 (Shannon-Fano Coding)

- 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
三	0.35	00
点	0.17	01
一	0.17	1...
四	0.16	1...
円	0.15	1...



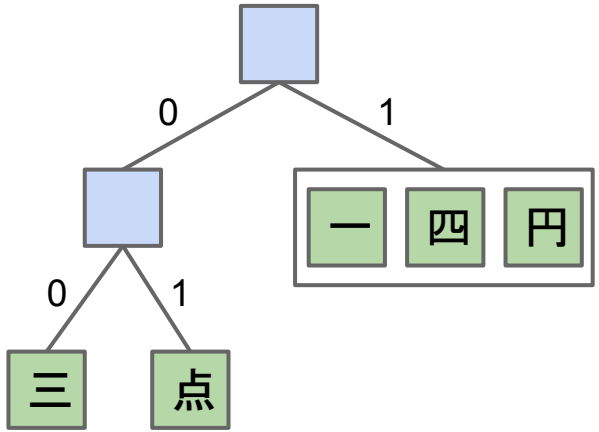
Code Calculation Approach #1 (Shannon-Fano Coding)

- 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
三	0.35	00
点	0.17	01
一	0.17	1...
四	0.16	1...
円	0.15	1...

Left half

Right half



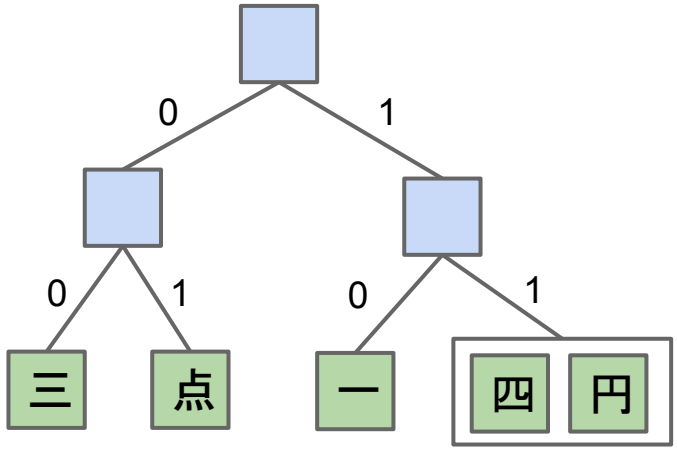
Code Calculation Approach #1 (Shannon-Fano Coding)

- 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
三	0.35	00
点	0.17	01
一	0.17	10
四	0.16	11...
円	0.15	11...

Left half

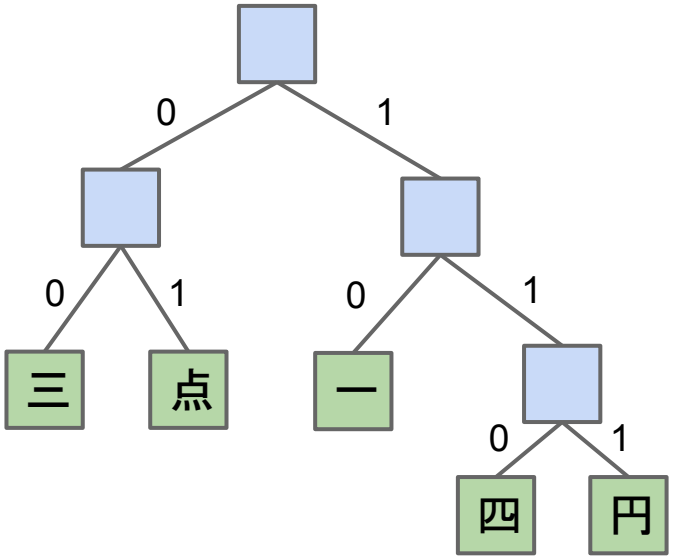
Right half



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
三	0.35	00
点	0.17	01
一	0.17	10
四	0.16	110
円	0.15	111

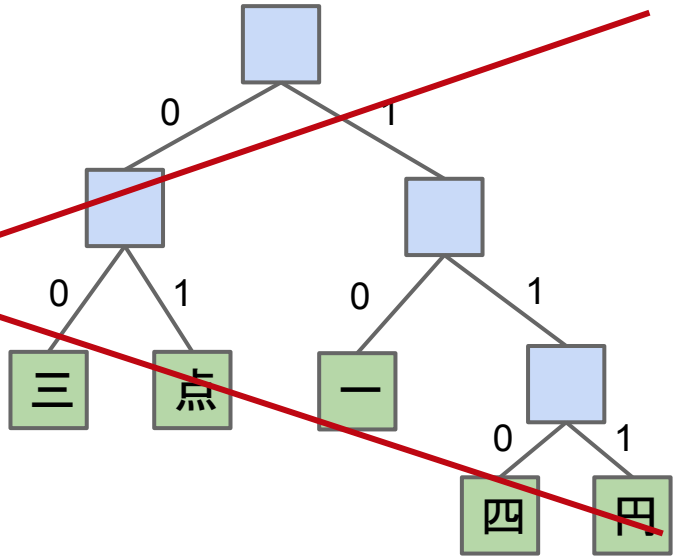


Code Calculation Approach #1 (Shannon-Fano Coding)

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
三	0.35	00
点	0.17	01
一	0.17	10
四	0.16	110
円	0.15	111



Huffman Coding: Core Idea

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- **Core Idea**
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Code Calculation Approach #2: Huffman Coding

Suppose I have a text file with the 5 Kanji shown.

- 35% of the characters are 三.

Symbol	Frequency
三	0.35
点	0.17
一	0.17
四	0.16
円	0.15

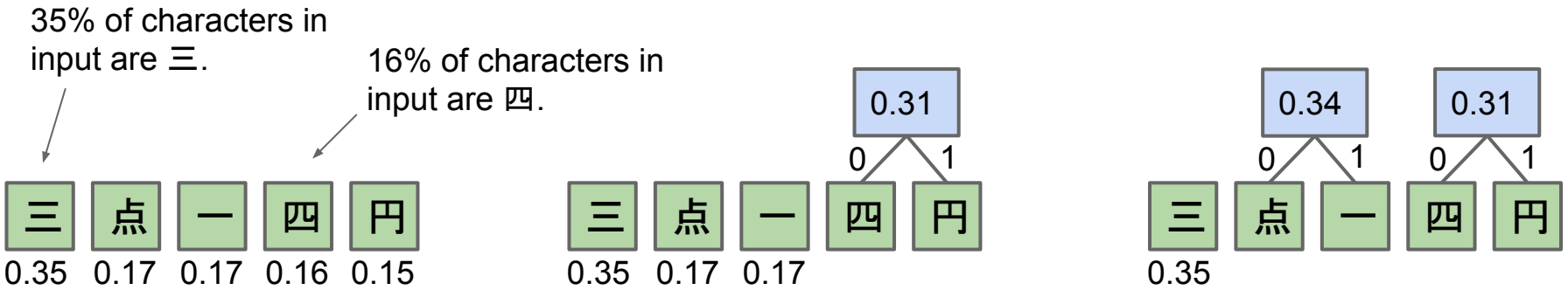
35% of all characters are 三

三 点 一 四 円

Code Calculation Approach #2: Huffman Coding

Calculate relative frequencies.

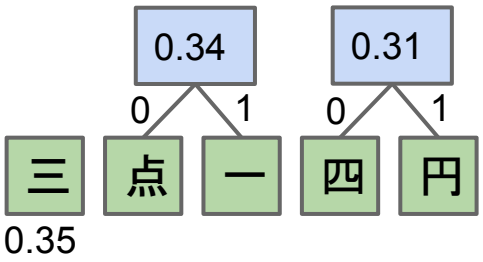
- Assign each symbol to a node with weight = relative frequency.
- Take the two smallest nodes and merge them into a super node with weight equal to sum of weights.
- Repeat until everything is part of a tree.

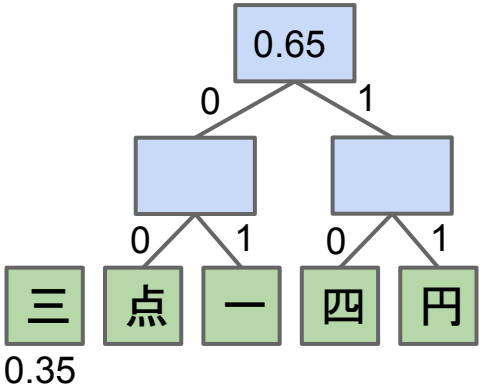


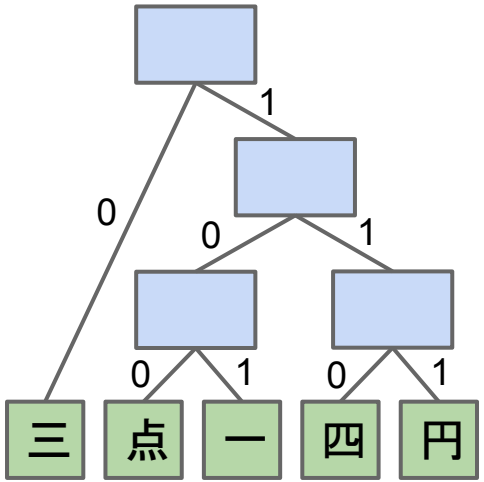
Code Calculation Approach #2: Huffman Coding

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.







Efficiency Assessment: http://yellkey.com/true

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

A. (1*1 + 4*3) / 5
= 2.6 bits per symbol

B. (0.35) * 1 + (0.17 + 0.17 + 0.16 + 0.15) * 3
= 2.3 bits per symbol

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

Symbol	Frequency	Huffman Code
三	0.35	0
点	0.17	100
一	0.17	101
四	0.16	110
円	0.15	111

Efficiency Assessment of Huffman Coding

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

B. (0.35) * 1 + (0.17 + 0.17 + 0.16 + 0.15) * 3 = 2.3 bits per symbol.

Symbol	Frequency	Huffman Code
三	0.35	0
点	0.17	100
一	0.17	101
四	0.16	110
円	0.15	111

Example assuming we have 100 symbols:

- 35 * 1 = 35 bits
- 17 * 3 = 51 bits
- 17 * 3 = 51 bits
- 16 * 3 = 48 bits
- 15 * 3 = 45 bits

Total: 230 bits
230 / 100 = 2.3 bits/symbol

Efficiency Assessment of Huffman Coding

If we had a file with 350 三 characters , 170 点 characters , 170 一 characters, 160 四 characters, and 150 円 characters, how many total bits would we need to encode this file using 32 bit Unicode? Using our Huffman code?

You don’t need a calculator.

Symbol	Frequency	Huffman Code
三	0.35	0
点	0.17	100
一	0.17	101
四	0.16	110
円	0.15	111

2.30 bits per symbol for texts with this distribution

Efficiency Assessment of Huffman Coding

If we had a file with 350 三 characters , 170 点 characters , 170 一 characters, 160 四 characters, and 150 円 characters, how many total bits would we need to encode this file using 32 bit Unicode? Using our Huffman code?

1000 total characters.

Space used:

- 32 bit Unicode: 32,000 bits.
- Huffman code: 2,300 bits.

Our code is 14 times as efficient!

- Can only encode strings with these 5 symbols.
- Only efficient for this particular frequency

Symbol	Frequency	Huffman Code
三	0.35	0
点	0.17	100
一	0.17	101
四	0.16	110
円	0.15	111

2.30 bits per symbol for texts with this distribution

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.
- In comparison, the Shannon entropy of the dataset is 2.233 bits

Symbol	Frequency	S-F Code	Huffman Code
三	0.35	00	0
点	0.17	01	100
一	0.17	10	101
四	0.16	110	110
円	0.15	111	111

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

Data Structures for Huffman Coding

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- **Data Structures for Huffman Coding**
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

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
...	

I ATE: 0000011000100101

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

I ATE: 100011110111101010

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. Two approaches:

- 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), but use more memory if some characters in the alphabet are unused.

I ATE: 0000011000100101

I ATE: 1000111101111010101

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
...	

I ATE: 0000011000100101

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

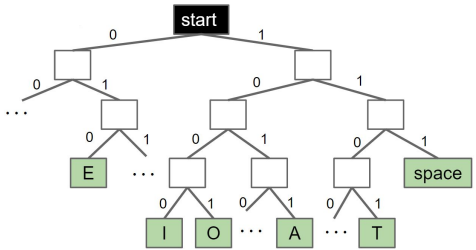
I ATE: 10001111010111101010

Prefix-Free Codes

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

- We need to look up **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: 000001100010010 1

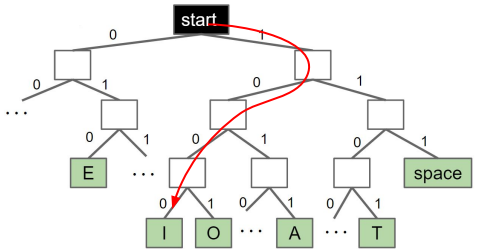
I ATE: 1000111101111010 10

Prefix-Free Codes

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

- We need to look up **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: 000001100010010 1

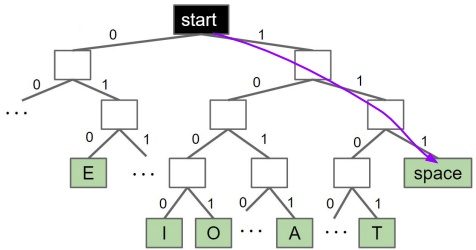
I ATE: ~~1000~~ 111101111010 10

Prefix-Free Codes

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

- We need to look up **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: 000001100010010 1

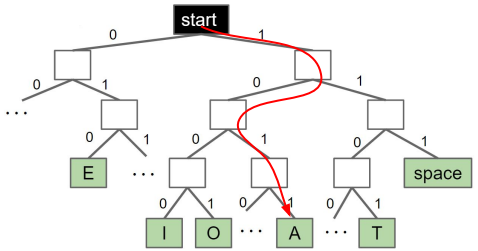
I ATE: ~~1000111~~10111101010

Prefix-Free Codes

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

- We need to look up **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: 000001100010010 1

I **A**TE: ~~1000~~ ~~111~~ ~~1011~~ 1101010

Huffman Coding in Practice

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- **Huffman Coding in Practice**

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

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?

Huffman Compression (My 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?

- Approach 1 will result in suboptimal encoding.
- Approach 2 requires you to use extra space for the codeword table in the compressed bitstream.

For very large inputs, the cost of including the codeword table will become insignificant.

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**: 三三円円円一三一三円四円三四一点四点四一四三三四円一三一円点一円三点三四一一四一三三円点一四三三三一点三一三点一三点一三一円三一点円点三円三三円点三三点三円点点四四四四三三点四三三円点四三三四三点三三

Step 1: Count frequencies.

Step 2: Build encoding array and decoding trie.

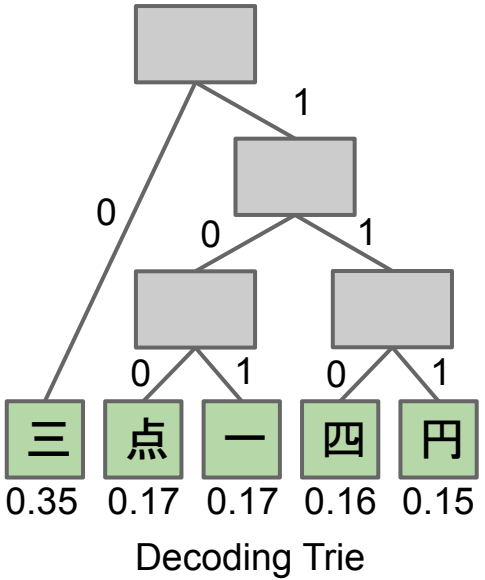
Step 3: Write decoding trie to output.huf.

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

→ See `writeTrie` in this code if you're curious.

Output bits: 010101010101001…00111111111101…

Decoding Trie | Codewords



Decoding Trie

Huffman Decompression Example [Demo Link]

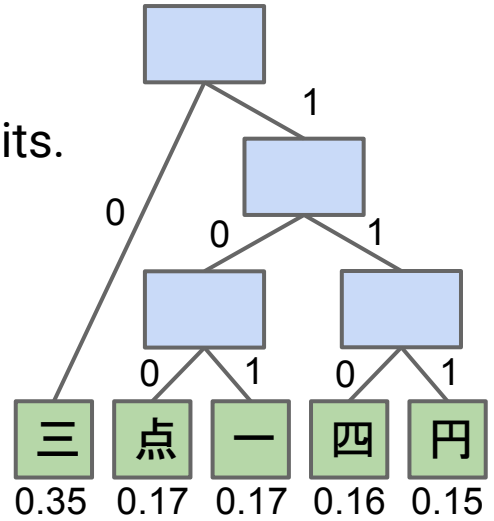
Given **input bitstream**: `010101010101001…`**`00111111111101…`**

Decoding Trie | Codewords

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!
  - 三 is `0100111000001001` in Unicode.
  - “Outputting 三” actually means outputting these 32 bits.



Output symbols: 三三円円円一…

- Output bits: `0100111000001001`...

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.huf have been converted back to their uncompressed form.

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

Compression Ratios

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- **Compression Ratios**

LZW Style Compression (Extra)

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 (low-entropy substrings) inside the sequence.

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

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.

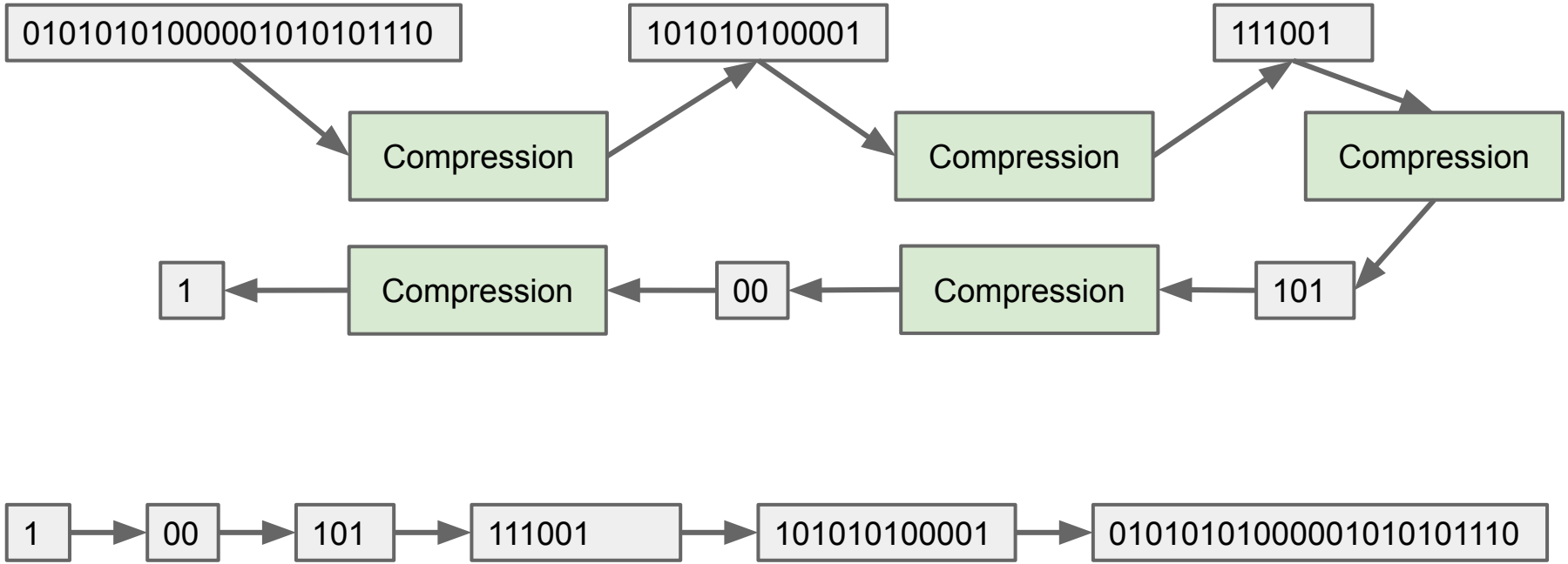
Let's start with a straightforward puzzle.

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.



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%!

Universal Compression: No Free Lunch Theorem

In general, no compression algorithm can compress below the entropy of a dataset.

- If we want to look at all possible strings of length N bits, this forms a dataset with the maximum N bits of entropy (effectively, we need this to work for random strings)

So any compression algorithm will on average keep the same size

- If we want to make one string shorter 1000 bytes, we need to make 1000 other strings one byte longer.

In fact, we can go even further: If we include the compression algorithm as part of our file size, any compression algorithm will on average **increase** the length of a random input (unless the algorithm just returns the original string)

But because English isn't a completely random string, we can make compression algorithms that make low-entropy (useful) strings shorter, and high-entropy (useless) strings longer.

Summary

The more predictable data is, the less information it actually carries, and therefore, the better we can compress that data

Most meaningful data is low-entropy, so we can generally compress text/images/videos to smaller sizes (at the expense of making random nonsense strings slightly longer)

Compression can be slow; in general, slower algorithms yield better results, but regardless, we can only compress down to the Shannon Entropy limit (times a fixed constant, depending on the computation model)

One might ask if it's possible to write the best possible compression algorithm, and if so, what the runtime of that compression algorithm is. We'll discuss this next time, and how it connects to P=NP and computability.

LZW Style Compression (Extra)

Lecture 38, CS61B, Spring 2024

Today's Goal: Compression

Information Theory

Prefix Free Codes

Shannon Fano Codes

Huffman Coding

- Core Idea
- Data Structures for Huffman Coding
- Huffman Coding in Practice

Compression Theory

- Compression Ratios

LZW Style Compression (Extra)

Thought Experiment

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

- B=”aababcabcdabcdeabcdefabcdefgabcdefgh”?
- B=”abababababababababababababababab”?
- B=”aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”?

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 to[illegible]
  .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