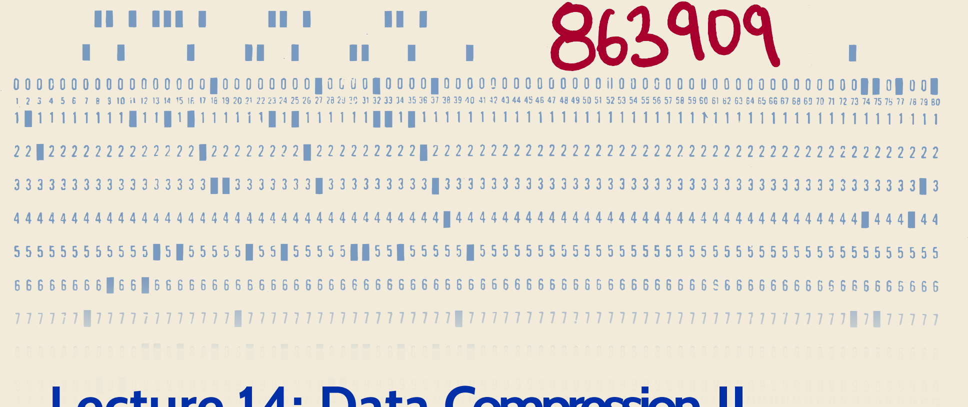


Lecture 14: Data Compression II

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: November 19, 2024

Announcements

1. Programming Assignment 2 posted
   - Due 29 November
2. Quiz 6 due Friday
   - Covers Lecture 13 material
   - 1 Question, Short Answer
   - Usual rules apply
3. Attendance Code:

863909

Meeting Goals

1. Introduce Programming Assignment 2
2. Discuss limitations of general compression
3. Introduce compression techniques that exploit redundancy in texts
   3.1 Run length encoding
   3.2 Elias codes
   3.3 Lempel-Ziv-Welch (LZW) encoding

Programming Assignment 2

The Setup

So. You've decided to **cheat** on the final exam for COMP666.

The Setup

So. You've decided to **cheat** on the final exam for COMP666.

- Final exam consists of 20 true/false questions
- Your hacker friend:
  - learns correct answers immediately before exam
  - can relay *some* information to you
  - limited to a single 10 bit message
- Before the exam:
  - figure out how to get the most out of a 10 bit message

The Setup

So. You've decided to **cheat** on the final exam for COMP666.

- Final exam consists of 20 true/false questions
- Your hacker friend:
  - learns correct answers immediately before exam
  - can relay *some* information to you
  - limited to a single 10 bit message
- Before the exam:
  - figure out how to get the most out of a 10 bit message

Goal: figure out a scheme to get the highest possible **guaranteed** score (without knowing how to answer any questions correctly yourself)

- For all possible (correct) exam solutions, maximize the *worst* score you receive

The Problem, Formalized

Three Pieces:

- $B = B[0..20)$ the correct solutions to the exam
  - expressed in binary 1 for true, 0 for false
  - known only to your hacker friend
- $M = M[0..10)$ the message your friend sends you
  - also expressed in binary
- $A = A[0..20)$ the answers your record for the exam, in binary

The Problem, Formalized

Three Pieces:

- $B = B[0..20)$ the correct solutions to the exam
  - expressed in binary 1 for true, 0 for false
  - known only to your hacker friend
- $M = M[0..10)$ the message your friend sends you
  - also expressed in binary
- $A = A[0..20)$ the answers your record for the exam, in binary

Two Procedures:

- *Encode* the correct exam solutions B to a message M
  - preformed by your hacker friend
- *Decode* the message M to exam solutions A
  - performed by you during the exam

The Problem, Formalized

Three Pieces:

- $B = B[0..20)$ the correct solutions to the exam
  - expressed in binary 1 for true, 0 for false
  - known only to your hacker friend
- $M = M[0..10)$ the message your friend sends you
  - also expressed in binary
- $A = A[0..20)$ the answers your record for the exam, in binary

Two Procedures:

- *Encode* the correct exam solutions B to a message M
  - preformed by your hacker friend
- *Decode* the message M to exam solutions A
  - performed by you during the exam

correct answers

all possible exam solutions

One Goal: Achieve the maximum *guaranteed* score.

- $\underline{20} - \max\left\{ d_H(A, B) \,\middle|\, \boxed{B \in \{0,1\}^{20}} \right\}$
- $d_H(A, B)$ is **Hamming distance** = number of indices where solutions differ

your answers

Your Task

Main Task. Implement functions to compute & decode the message M

- complete `exam_cheat_code.py`
- `encode(solutions:  list[int]) -> list[int]`
  - input: list of binary values, B, length 20
  - output: list of binary values, M, length 10
- `decode(message:  list[int]) -> list[int]`
  - input: list of binary values, M, length 10
  - output: list of binary values, A, length 20

Your Task

Main Task. Implement functions to compute & decode the message M

- complete `exam_cheat_code.py`
- `encode(solutions:  list[int]) -> list[int]`
  - input: list of binary values, B, length 20
  - output: list of binary values, M, length 10
- `decode(message:  list[int]) -> list[int]`
  - input: list of binary values, M, length 10
  - output: list of binary values, A, length 20

Given. Testing program `exam_tester.py`

- computes the the worst guaranteed score from your scheme

Your Task

Main Task. Implement functions to compute & decode the message M

- complete `exam_cheat_code.py`
- `encode(solutions: list[int]) -> list[int]`
  - input: list of binary values, B, length 20
  - output: list of binary values, M, length 10
- `decode(message: list[int]) -> list[int]`
  - input: list of binary values, M, length 10
  - output: list of binary values, A, length 20

Given. Testing program `exam_tester.py`

- computes the the worst guaranteed score from your scheme

Secondary Task. Explain how your scheme works!

- complete a single page PDF (typed) explaining your approach

Your Task

Main Task. Implement functions to compute & decode the message M

- complete `exam_cheat_code.py`
- `encode(solutions: list[int]) -> list[int]`
  - input: list of binary values, B, length 20
  - output: list of binary values, M, length 10
- `decode(message: list[int]) -> list[int]`
  - input: list of binary values, M, length 10
  - output: list of binary values, A, length 20

Given. Testing program `exam_tester.py`

- computes the the worst guaranteed score from your scheme

Secondary Task. Explain how your scheme works!

- complete a single page PDF (typed) explaining your approach

Optional Task. Prove an **upper bound** on the best achievable score for **any** cheating scheme.

Evaluation

Total marks: 100

- Main Task (code): 70 marks
  - higher guaranteed test score = more marks!
  - $>$ 70 marks possible if guaranteed score is $\boxed{> 16}$
  - more marks for **simpler** solutions (tie breaking)
- Secondary Task (explanation): 30 marks
  - Concise and clear explanation of approach
  - Sensible/systematic approach

  ~ 1 quiz
- Optional Task (upper bound proof): up to $\boxed{\text{20 marks}}$ **extra credit**

Evaluation

Total marks: 100

- Main Task (code): 70 marks
  - higher guaranteed test score = more marks!
  - > 70 marks possible if guaranteed score is > 16
  - more marks for **simpler** solutions (tie breaking)
- Secondary Task (explanation): 30 marks
  - Concise and clear explanation of approach
  - Sensible/systematic approach
- Optional Task (upper bound proof): up to 20 marks **extra credit**

Admininstration

- Full instructions on course website: https://willrosenbaum.com/teaching/2024f-comp-526/
- Submission through Canvas
- **Due 29 November** (next Friday)

Limits of Compression

From Last Time

Introduced lossless compression task

$S \xrightarrow{\text{compress}} C$

- Compression ratio
  - $\frac{|C|\cdot\log|\Sigma_C|}{\left|S\cdot\log|\Sigma_S|\right|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S|\cdot\log|\Sigma_S|}$

length of encoded text (in bits)

smaller

- Character encoding
  - encode characters in binary

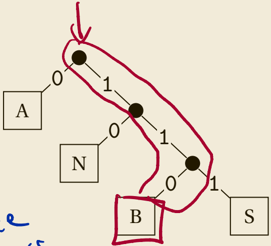
$|S|$ = # chars in S

$|\Sigma_S|$ = size of alphabet (possible chars)

From Last Time

Introduced lossless compression task

- Compression ratio
  - $\frac{|C|\cdot\log|\Sigma_C|}{\big|S\cdot\log|\Sigma_S|\big|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S|\cdot\log|\Sigma_S|}$
- Character encoding
  - encode characters in binary
- Prefix coding
  - ensures unambiguous decoding



Source chars

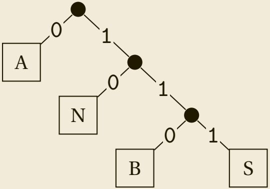
c	A	N	B	S
$E(c)$	0	10	110	111

encoded chars
"codewords"

From Last Time

Introduced lossless compression task

- Compression ratio
  - $\frac{|C|\cdot\log|\Sigma_C|}{\left|S\cdot\log|\Sigma_S|\right|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S|\cdot\log|\Sigma_S|}$
- Character encoding
  - encode characters in binary
- Prefix coding
  - ensures unambiguous decoding



c	A	N	B	S
$E(c)$	0	10	110	111

From Last Time

Introduced lossless compression task

- Compression ratio
  - $\frac{|C|\cdot\log|\Sigma_C|}{\left|S\cdot\log|\Sigma_S|\right|} \overset{\Sigma_C=\{0,1\}}{=} \frac{|C|}{|S|\cdot\log|\Sigma_S|}$
- Character encoding
  - encode characters in binary
- Prefix coding
  - ensures unambiguous decoding
- Huffman codes
  - most efficient possible prefix code

char encoding
non-adaptive

From Last Time

Introduced lossless compression task

- Compression ratio
  - $\frac{|C| \cdot \log|\Sigma_C|}{|S \cdot \log|\Sigma_S||} \overset{\Sigma_C=\{0,1\}}{=} \boxed{\frac{|C|}{|S| \cdot \log|\Sigma_S|}}$
- Character encoding
  - encode characters in binary

PollEverywhere Question

Suppose S is a text of length n over an alphabet Σ_S of size 8. What is the **smallest** possible compression ratio of any character encoding of S?

pollev.com/comp526

Must write n codeword

$n = |S|$

$|\Sigma_S| = 8$

n ← smallest possible encoding 1 bit per char

$$\frac{n}{n \log|\Sigma_S|} = \frac{1}{\boxed{\log 8}} = \boxed{\frac{1}{3}}$$

Character Encoding Limitations

An Issue with character encodings:

- Only single characters are encoded *in isolation*
- Cannot exploit *larger patterns* in text

Example. Huffman encoding doesn't distinguish between the following texts:

- $S =$ `AAAAAAAAAAAAAAAAAAAABBBBBBBBBBCCCCCCCCCC`
- $T =$ `ACBBAAACAABAABABCAACCAABBACCAAAACBBAABCC`

Character Encoding Limitations

An Issue with character encodings:

- Only single characters are encoded *in isolation*
- Cannot exploit *larger patterns* in text

Example. Huffman encoding doesn't distinguish between the following texts:

- $S =$ `AAAAAAAAAAAAAAAAAAAABBBBBBBBBBCCCCCCCCCC`
- $T =$ `ACBBAAACAABAABABCAACCAABBACCAAAACBBAABCC`

But evidently, strings like S are admit simpler descriptions than T:

- Print 20 `A`s followed by 10 `B`'s followed by 10 `C`s.

Character Encoding Limitations

An Issue with character encodings:

- Only single characters are encoded *in isolation*
- Cannot exploit *larger patterns* in text

Example. Huffman encoding doesn't distinguish between the following texts:

- $S =$ `AAAAAAAAAAAAAAAAAAAABBBBBBBBBBCCCCCCCCCC`
- $T =$ `ACBBAAACAABAABABCAACCAABBACCAAAACBBAABCC`

But evidently, strings like S are admit simpler descriptions than T:

- Print 20 `A`s followed by 10 `B`'s followed by 10 `C`s.

Question. How can we generalize our notation of encoding to compress texts further?

Character Encoding Limitations

An Issue with character encodings:

- Only single characters are encoded *in isolation*
- Cannot exploit *larger patterns* in text

Example. Huffman encoding doesn't distinguish between the following texts:

- $S =$ `AAAAAAAAAAAAAAAAAAAABBBBBBBBBBCCCCCCCCCC`
- $T =$ `ACBBAAACAABAABABCAACCAABBACCAAAACBBAABCC`

But evidently, strings like S are admit simpler descriptions than T:

- Print 20 `A`s followed by 10 `B`'s followed by 10 `C`s.

Question. How can we generalize our notation of encoding to compress texts further?

- One idea: use a larger source alphabet—e.g., use pairs of characters

General Compression

High Level View. A compressed representation of S is a **program** whose output is S.

General Compression

High Level View. A compressed representation of S is a **program** whose output is S.

- Huffman codes are very restricted programs represented by the Huffman tree
- Why restrict ourselves?
- Fix syntax and semantics for general decoding
- Any valid program that outputs S is an encoding of S

General Compression

High Level View. A compressed representation of *S* is a **program** whose output is *S*.

- Huffman codes are very restricted programs represented by the Huffman tree
- Why restrict ourselves?
- Fix syntax and semantics for general decoding
- Any valid program that outputs *S* is an encoding of *S*

Example.

```
s = ‘’
for i in range(1000000):
  s = s + ‘A’
print(s)
```

$$S = \underbrace{AA\cdots A}_{1\text{ million}}$$

Quantifying General Compression

Definition. Suppose we fix a (programming) language L (e.g., Python). Given a source text S, the **Kolmogorov complexity** of S (relative to L), denoted $K(S)$ is the length of the shortest program whose output is S.

S (desired output)

Programs = Text (compressed)

Code whose output is S

Quantifying General Compression

Definition. Suppose we fix a (programming) language L (e.g., Python). Given a source text S, the **Kolmogorov complexity** of S (relative to L), denoted $K(S)$ is the length of the shortest program whose output is S.

- This is a *very* general notion of encoding of S
- S may admit a huge amount of compression
  - $S = \underbrace{\texttt{AAA...A}}_{n \text{ times}}\underbrace{\texttt{BBB...B}}_{2n \text{ times}}$
  - $S = 31415926535\ldots$
  - $S = 12345678910111213141516\ldots$

Digits of π

Count & print

Size of program $O(1)$
+ $\log n$ bits to specify where to stop.

String of length n $\rightsquigarrow$ $O(\log n)$ bits

Quantifying General Compression

Definition. Suppose we fix a (programming) language L (e.g., Python). Given a source text S, the **Kolmogorov complexity** of S (relative to L), denoted $K(S)$ is the length of the shortest program whose output is S.

- This is a *very* general notion of encoding of S
- S may admit a huge amount of compression
  - $S = \underbrace{\texttt{AAA...A}}_{n \text{ times}}\underbrace{\texttt{BBB...B}}_{2n \text{ times}}$
  - $S = 31415926535\ldots$
  - $S = 12345678910111213141516\ldots$
  - $S = 01101001100101101001011001101001 0110\ldots$

  ...though it may not be obvious how.

What simple program generates this?

Quantifying General Compression

Definition. Suppose we fix a (programming) language L (e.g., Python). Given a source text S, the **Kolmogorov complexity** of S (relative to L), denoted $K(S)$ is the length of the shortest program whose output is S.

- This is a *very* general notion of encoding of S
- S may admit a huge amount of compression
  - $S = \underbrace{\texttt{AAA...A}}_{n\text{ times}}\underbrace{\texttt{BBB...B}}_{2n\text{ times}}$
  - $S = 31415926535\ldots$
  - $S = 12345678910111213141516\ldots$
  - $S = 0110100110010110100101100110100101 10\ldots$

…though it may not be obvious how.

Question. How much compression can we achieve in this way?

- How close to $K(S)$ can we get?

Limits of General Compression

Fact 1. Suppose $\Sigma = \{0, 1\}$ and fix any language L. Then for every positive integer n, there exists a source text $S \in \Sigma^n$ for which $K(S) \geq n$.

- Interpretation: some input text is not compressible at all

Limits of General Compression

Fact 1. Suppose $\Sigma = \{0, 1\}$ and fix any language L. Then for every positive integer n, there exists a source text $S \in \Sigma^n$ for which $K(S) \geq n$.

- Interpretation: some input text is not compressible at all
- Reason is pretty simple: there are more possible texts of length n than all possible texts of length up to n

Limits of General Compression

Fact 1. Suppose $\Sigma = \{0, 1\}$ and fix any language L. Then for every positive integer n, there exists a source text $S \in \Sigma^n$ for which $K(S) \geq n$.

- Interpretation: some input text is not compressible at all
- Reason is pretty simple: there are more possible texts of length n than all possible texts of length up to n
- "No free lunch" theorem for compression
- Explore in tutorials this week

Limits of General Compression

Fact 1. Suppose $\Sigma = \{0, 1\}$ and fix any language L. Then for every positive integer n, there exists a source text $S \in \Sigma^n$ for which $K(S) \geq n$.

- Interpretation: some input text is not compressible at all
- Reason is pretty simple: there are more possible texts of length n than all possible texts of length up to n
- "No free lunch" theorem for compression
- Explore in tutorials this week

Well we can't compress *everything*, but how well can we do?

- Can we generally find an optimal compression of a string in a given language?
- We did manage this for prefix codes! (Huffman codes)

Impossibility and Compression

call functions, do iteration

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes* $K(S)$
- *distinguishes* $K(S) = |S|$ *from* $K(S) < |S|$.

Impossibility and Compression

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes* $K(S)$
- *distinguishes* $K(S) = |S|$ *from* $K(S) < |S|$.

Proof (sketch). Argue by contradiction

- Suppose P is a program that computes $K(S)$.
- By Fact 1, for every n, there is some S_n with $K(S_n) \geq n$.

Impossibility and Compression

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes $K(S)$*
- *distinguishes $K(S) = |S|$ from $K(S) < |S|$.*

$\Sigma = \{0,1\}$

2^n possible texts of length n

Proof (sketch). Argue by contradiction

- Suppose P is a program that computes $K(S)$.
- By Fact 1, for every n, there is some S_n with $K(S_n) \geq n$.
- Consider the following program, P':
  - On input n, iterate over all source texts S of length n
  - Apply P, and return the first S_n with $\boxed{P(S_n) \geq n}$.

Impossibility and Compression

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes $K(S)$*
- *distinguishes $K(S) = |S|$ from $K(S) < |S|$.*

$O(1)$ size

Proof (sketch). Argue by contradiction

- Suppose $\boxed{P}$ is a program that computes $K(S)$.
- By Fact 1, for every n, there is some S_n with $K(S_n) \geq n$.
- Consider the following program, P':
  - On input n, iterate over all source texts S of length n
  - Apply P, and return the first S_n with $P(S_n) \geq \boxed{n.}$
- **Claim.** $K(S_n) = O(\log n)$.

Description size $O(\log n)$ only input value n

$\log n$ bits to write this

P'(n) is a program that outputs S_n and siz of prog description is $O(\log n)$

Impossibility and Compression

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes* $K(S)$
- *distinguishes* $K(S) = |S|$ *from* $K(S) < |S|$.

Proof (sketch). Argue by contradiction

- Suppose P is a program that computes $K(S)$.
- By Fact 1, for every n, there is some S_n with $K(S_n) \geq n$.
- Consider the following program, P':
    - On input n, iterate over all source texts S of length n
    - Apply P, and return the first S_n with $P(S_n) \geq n$.
- **Claim.** $K(S_n) = O(\log n)$.
- This contradicts the assumption that P computed $K(S)$. $\square$

Impossibility and Compression

Theorem

Suppose L is a "sufficiently rich" programming language (e.g. Python). Then there is no algorithm/program that for any string S:

- *computes* $K(S)$
- *distinguishes* $K(S) = |S|$ *from* $K(S) < |S|$.

Moral. There is no general method for determining how compressible a source text might be.

- Most texts are not very compressible.
  - Generalization of Fact 1.
- But many "interesting" source texts are compressible.
- Can still exploit features of common texts
  - most "interesting" source texts obey some patterns

Run Length Encoding

A Simple Idea

Example. How could we compress the following source text?

```
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000101100100000111111000000000011111000
001111111110001111111110000001111111000
001111011010001110001111000001110000000
001100000000000000000111000111000000000
001100000000000000000011001110000000000
001100000000000000000011001110000000000
001101100000000000000111001100111110000
001111111100000000000111001111111111000
001110111111000000001110001111110011100
000000000111000000011100001110000001110
000000000111000000011000001110000001100
000000000011000000110000000110000001110
000000000011000001110000001110000001100
000000000111000111000000000110000001110
000000000110000111000000000111000011100
001101111110001111011101000011111111000
011111111100011111111111100001111110000
000101100000001010011001000000100100000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

TIFF

Simple Setting. $\Sigma_S = \{0, 1\}$.

A Simple Idea

Example. How could we compress the following source text?

```
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000101100100000111111000000000011111000
001111111110001111111110000001111111000
001111011010001110001111000011100000000
001100000000000000000111000111000000000
001100000000000000000011001110000000000
001100000000000000000011001110000000000
001101100000000000000111001100111110000
001111111100000000000111001111111111000
001110111110000000001110001111100111100
000000000111000000011100001110000001110
000000000111000000011000001110000001100
000000000011000000110000000110000001110
000000000011000001110000001110000001100
000000000111000111000000000110000001110
000000000110000111000000000111000011100
001101111110001111011101000011111111000
011111111100011111111111100001111111000
000101100000001010011001000000100100000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

Simple Setting. $\Sigma_S = \{0, 1\}$.

Simpler Example. How could we compress $S = 1111110000111000000000$?

A Simple Idea

Example. How could we compress the following source text?

```
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000101100100000111111000000000011111000
001111111110001111111110000001111111000
001111011010001110001111000011100000000
001100000000000000000111000111000000000
001100000000000000000011001110000000000
001100000000000000000011001110000000000
001101100000000000000111001100111110000
001111111100000000000111001111111111000
001110111110000000001110001111100111100
000000000111000000011100001110000001110
000000000111000000011000001110000001100
000000000011000000110000000110000001110
000000000011000001110000001110000001100
000000000111000111000000000110000001110
000000000110000111000000000111000011100
001101111110001111011101000011111111000
011111111110001111111111100001111110000
000101100000001010011001000000100100000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

Simple Setting. $\Sigma_S = \{0, 1\}$.

Simpler Example. How could we compress $S = 11111100001111000000000$?

Idea. Store **runs**:

$$\underbrace{111111}_{6}\underbrace{0000}_{4}\underbrace{1111}_{4}\underbrace{000000000}_{9}$$

A Simple Idea

Example. How could we compress the following source text?

```
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
000101100100000111111000000000011111000
001111111110001111111110000001111111000
001111011010001110001111000011100000000
001100000000000000000111000111000000000
001100000000000000000011001110000000000
001100000000000000000011001110000000000
001101100000000000000111001100111110000
001111111110000000000111001111111111000
001110111110000000001110001111100111100
000000000111000000011100001110000001110
000000000111000000011000001110000001100
000000000011000000110000000110000001110
000000000011000001110000001110000001100
000000000111000111000000000110000001110
000000000110000111000000000111000011100
001101111110001111011101000011111111000
011111111110001111111111000011111110000
000101100000001010011001000000100100000
000000000000000000000000000000000000000
000000000000000000000000000000000000000
```

Simple Setting. $\Sigma_S = \{0, 1\}$.

Simpler Example. How could we compress $S = \texttt{1111110000111000000000}$?

Idea. Store **runs**:

$$\underbrace{\texttt{111111}}\underbrace{\texttt{0000}}\underbrace{\texttt{1111}}\underbrace{\texttt{000000000}}$$

Run Length Encoding. For binary alphabet, store

- the first bit (0 or 1)
- the lengths of the runs

Example. 1111110000111000000000 becomes 1 , 6 , 4 , 4 , 9

A Simple Idea

Run Length Encoding. For binary alphabet, store

- the first bit (0 or 1)
- the lengths of the runs

Example. 1111110000111100000000 becomes $\boxed{1, 6, 4, 4, 9}$

Question. What is wrong with this encoding?

still must represent run lengths

$m \rightsquigarrow \log m$ bits (binary)

Commas are not accounted for

binary value

1011

⇓

01 00 01 01

comma = 11

$\underset{1}{01}$ $\underset{\text{comma}}{11}$ $\underbrace{01\ 01}_{3}$

A Simple Idea

Run Length Encoding. For binary alphabet, store

- the first bit (`0` or `1`)
- the lengths of the runs

Example. `1111110000111000000000` becomes `1,6,4,4,9`

Question. What is wrong with this encoding?

Issues:

- The alphabet is no longer binary!
- Even if we express run lengths in binary, we still need an extra symbol for the comma!

Representing Lists

Generic Problem. Given only a binary alphabet, how can we express a *list* of numbers efficiently?

- A single m can be represented with $\log m$ bits.
- Can we represent k such numbers with $\approx k \log m$ bits?

Representing Lists

Generic Problem. Given only a binary alphabet, how can we express a *list* of numbers efficiently?

- A single m can be represented with $\log m$ bits.
- Can we represent k such numbers with $\approx k \log m$ bits?

Two approaches.

- Represent list lengths in *unary*:

$$m = \underbrace{\mathtt{000\cdots0}}_{m \text{ times}}\mathtt{1}$$

Sentinel 1 denotes the end of a number

Representing Lists

Generic Problem. Given only a binary alphabet, how can we express a *list* of numbers efficiently?

- A single m can be represented with $\log m$ bits.
- Can we represent k such numbers with $\approx k \log m$ bits?

Two approaches.

- Represent list lengths in *unary*:

$$m = \underbrace{000\cdots0}_{m \text{ times}}1$$

  Sentinel 1 denotes the end of a number

- Represent values in binary, and concatenate encoded values

2 1 11

$$5, 2, 3 \longmapsto \boxed{10}\boxed{1}\boxed{1011}$$

101 10 11

Representing Lists

Generic Problem. Given only a binary alphabet, how can we express a *list* of numbers efficiently?

- A single m can be represented with $\log m$ bits.
- Can we represent k such numbers with $\approx k \log m$ bits?

Two approaches.

- Represent list lengths in *unary*:

$$m = \underbrace{\texttt{000}\cdots\texttt{0}}_{m \text{ times}}\texttt{1}$$

  Sentinel `1` denotes the end of a number
- Represent values in binary, and concatenate encoded values

$$5, 2, 3 \longmapsto \texttt{1011011}$$

Question. How to address these shortcomings?

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two
approaches:

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two approaches:

- Express m in binary (using $\boxed{\log m}$ bits)
- Write the length of m's binary representation (less 1) in unary
- Concatenate unary then binary parts

Elias Encoding

Goal. A *prefix code* for long runs (of 0s or 1s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two approaches:

- Express m in binary (using $\log m$ bits)
- Write the length of m's binary representation (less 1) in unary
- Concatenate unary then binary parts

Example. $m = 21$

- $21 = 10101_2$ (binary)
- length $\ell = 4 = 0000_1$
- encoding $21 \mapsto 000010101$

Example

$\boxed{000}\underbrace{1011}_{\ell+1 \text{ bits encode } \#}$

11

ℓ

read until first 1

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two approaches:

- Express m in binary (using $\log m$ bits)
- Write the length of m's binary representation (less 1) in unary
- Concatenate unary then binary parts

Example. $m = 21$

- $21 = \texttt{10101}_2$ (binary)
- length $\ell = 4 = \texttt{0000}_1$
- encoding $21 \mapsto \texttt{00010101}$

Question. Why is this a prefix code?

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two approaches:

- Express m in binary (using $\log m$ bits)
- Write the length of m's binary representation (less 1) in unary
- Concatenate unary then binary parts

Example. $m = 21$

- $21 = \texttt{10101}_2$ (binary)
- length $\ell = 4 = \texttt{0000}_1$
- encoding $21 \mapsto \texttt{00010101}$

Question. Why is this a prefix code?

- Binary representations start with 1
- ℓ 0's followed by $\ell + 1$ bits starting with a 1.

Elias Encoding

Goal. A *prefix code* for long runs (of `0`s or `1`s)

Two approaches.

- Represent lengths in *unary*
  - too long!
- Represent values in binary
  - not prefix code!

Idea. Combine the two approaches:

- Express m in binary (using $\log m$ bits)
- Write the length of m's binary representation (less 1) in unary
- Concatenate unary then binary parts

Example. $m = 21$

- $21 = \mathtt{10101}_2$ (binary)
- length $\ell = 4 = \mathtt{0000}_1$
- encoding $21 \mapsto \mathtt{00010101}$

Question. Why is this a prefix code?

- Binary representations start with 1
- ℓ 0's followed by $\ell + 1$ bits starting with a 1.

This encoding of positive integers is called the **Elias gamma code**.

Unique Decodability

Observation. If the Elias gamma code is a prefix code, then we should be able to unambiguously decode a sequence of concatenated encoded strings.

Unique Decodability

Observation. If the Elias gamma code is a prefix code, then we should be able to unambiguously decode a sequence of concatenated encoded strings.

PollEverywhere Question

What is the first value stored in the following encoded text:

$\ell = 4$

`000011010000101111001111`

pollev.com/comp526

$$11010_2 = \boxed{26}$$

16 8 4 2 1

Run Length Encoding (RLE)

Fix this

Encoding procedure. To compute the RLE of a binary source text *S*:

- Write the first bit of *S*.
- For each run, write the length of the run using the Elias gamma code

Example. Encode $\overbrace{111111}^{\substack{110\\6}}\overbrace{0000}^{\substack{100\\4}}\overbrace{111}^{\substack{100\\4}}\overbrace{000000000}^{\substack{1001\\9}}$

1 $\boxed{00110}$ $\boxed{00100}$ 00100 0001001

Run Length Encoding (RLE)

Encoding procedure. To compute the RLE of a binary source text *S*:

- Write the first bit of *S*.
- For each run, write the length of the run using the Elias gamma code

Example. Encode `11111100001111000000000`

$\Longrightarrow$ ~~`1 00110 010 010 0001001`~~

Run Length Encoding (RLE)

Encoding procedure. To compute the RLE of a binary source text S:

- Write the first bit of S.
- For each run, write the length of the run using the Elias gamma code

Decoding procedure. To decode an RLE encoded text C:

- Write the first bit b_0 of C
- Parse code word starting at index 1 of C and repeat b_0 that many times
- Parse next coded value of C and write $1 - b_0$ that many times
- Repeat until done

Run Length Encoding (RLE)

Encoding procedure. To compute the RLE of a binary source text S:

- Write the first bit of S.
- For each run, write the length of the run using the Elias gamma code

Decoding procedure. To decode an RLE encoded text C:

- Write the first bit b_0 of C
- Parse code word starting at index 1 of C and repeat b_0 that many times
- Parse next coded value of C and write $1 - b_0$ that many times
- Repeat until done

first bit

Example. Decode 1001100100100001001.

6 2

11111100 11 0000000000

RLE Discussion

Generalizations and Applications.

- Can be extended to larger alphabets
  - write next character before run length
- Useful for some image formats (TIFF)

RLE Discussion

Generalizations and Applications.

- Can be extended to larger alphabets
  - write next character before run length
- Useful for some image formats (TIFF)

Evaluation.

- Fairly simple and fast!
- Can compress n bits to $\Theta(\log n)$ bits (extreme best case!)
- Not good compression for many common datatypes
  - No compression for run lengths ≤ 6
  - Expansion for run lengths $k = 2, 6$.

Lempel-Ziv-Welch Encoding

Lempel-Ziv Compression

Compression so far: Exploit frequently repeated *single characters*

- Huffman: globally frequent characters (large alphabet)
- RLE: repeated characters (binary alphabet)

Lempel-Ziv Compression

Compression so far: Exploit frequently repeated *single characters*

- Huffman: globally frequent characters (large alphabet)
- RLE: repeated characters (binary alphabet)

Observation. In many contexts, some substrings are much more frequent than others

- short words in English text (the, be, to, of, and, a, in, that)
- tags in HTML (`<div>`, `<a href`, ...)

Lempel-Ziv Compression

Compression so far: Exploit frequently repeated *single characters*

- Huffman: globally frequent characters (large alphabet)
- RLE: repeated characters (binary alphabet)

Observation. In many contexts, some substrings are much more frequent than others

- short words in English text (the, be, to, of, and, a, in, that)
- tags in HTML (`<div>`, `<a href`, `...`)

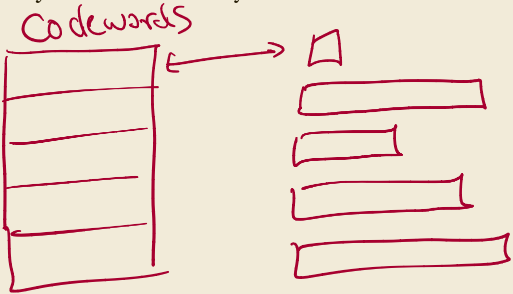
Lempel-Ziv covers a family of *adaptive* compression algorithms

- encode (frequently repeated) substrings of text with codewords
  - not just individual characters!
- Several variations of this idea
- Lempel-Ziv-Welch is a clean one (that is used in practice!)

LZW Idea

Codewords for different *strings* of text

- *Variable-to-fixed* encoding
  - all codewords have k bits (typical $k \approx 12$)
  - size of substring represented by each codeword varies
- Maintain a dictionary D (map) with $\underline{2^k}$ entries — codewords
  - codewords are *values* in the dictionary
  - text strings are *keys* in the dictionary



LZW Idea

Codewords for different *strings* of text

- *Variable-to-fixed* encoding
  - all codewords have k bits (typical $k \approx 12$)
  - size of substring represented by each codeword varies
- Maintain a dictionary D (map) with 2^k entries
  - codewords are *values* in the dictionary
  - text strings are *keys* in the dictionary

Encoding Idea.

- Initialize D with single characters $\boxed{\Sigma}$
- Start reading characters from S building up "words" (substrings) $\boxed{x}$
- If D contains x and next character is c, check if D contains $\boxed{xc}$
- If D does not contain xc, write $D(x)$ to C, **add** $\boxed{xc}$ **to** D, and start building next word from c

Example

Consider $S =$ N A N A S B A N A N A S

Dictionary

code	string
0000	A
0001	B
0010	N
0011	S
0100	NA
0101	AN
0110	NAS
0111	SB
1000	BA
1001	ANA
1010	ANAS
1011	

C = 0010 0000 0100 0011 0001 0101 1001

LZW in Pseudocode

```
procedure LZWEncode(S[0..n))
    x ← ε                                   ▷ previous word, initially empty
    C ← ε                                   ▷ output, initially empty
    D ← all c ∈ Σ_S                         ▷ dictionary of codewords
    k ← |Σ_S|                               ▷ next free codeword
    for i = 0, 1, ..., n − 1 do
        c ← S[i]
        if D.ContainsKey(xc) then
            x ← xc
        else
            C ← C D.Get(x)                  ▷ append codeword for x
            D.Put(xc, k)
            k ← k + 1, x ← c
        end if
    end for
    C ← C D.Get(x)
end procedure
```

LZW in Pseudocode

```
procedure LZWENCODE(S[0..n))
    x ← ε                                   ▷ previous word, initially empty
    C ← ε                                   ▷ output, initially empty
    D ← all c ∈ Σ_S                         ▷ dictionary of codewords
    k ← |Σ_S|                               ▷ next free codeword
    for i = 0, 1, ..., n−1 do
        c ← S[i]
        if D.CONTAINSKEY(xc) then
            x ← xc
        else
            C ← C D.GET(x)                  ▷ append codeword for x
            D.PUT(xc, k)
            k ← k+1, x ← c
        end if
    end for
    C ← C D.GET(x)
end procedure
```

For next time. Given C and D, how to decompress?

Next Time

Decompression

- Decoding LZW Encoding
- Making Texts Compressible

Scratch Notes