Lecture 10

Universal Compression with LZ77 + Practical tips on lossless compression

Announcements

- HW2 due Wed (Nov 1) midnight
  - Ed clarifications
- Project proposal - ~~Nov 1~~ Nov. 3
  - Sheet
- OH Wed. - Pulkit

Recap

- Entropy rate - fundamental limit of lossless compression
  - Special cases: entropy (iid), conditional entropy (Markov)

$$H(\mathbf{U}) = \lim_{n\to\infty} H(U_{n+1}|U_1, U_2, \ldots, U_n) = \lim_{n\to\infty} \frac{H(U_1, U_2, \ldots, U_n)}{n}$$

- Context-based arithmetic coding
  - Prediction implies compression
  - kth order adaptive arithmetic coding (and more advanced models)
  - Extreme text compression with LLMs

Quiz - Q2 First order adaptive arithmetic coding

Consider first order adaptive arithmetic coding with $\mathcal{X} = \{A, B, C\}$. The initial counts are set to 1, i.e., $c(A, A) = c(A, B) = \cdots = c(C, C) = 1$. You are encoding $X_1X_2X_3 = BAB$, and assume that for encoding the first symbol $X_1 = B$ you take $X_0 = A$ (padding).

1. What is $\hat{P}(X_2 = A|X_0X_1 = AB)$?
2. What is $\hat{P}(X_3 = B|X_0X_1X_2 = ABA)$?
3. Assuming arithmetic coding uses exactly $\sum_i \log_2 \frac{1}{\hat{P}(X_i|X_0,\ldots,X_{i-1})}$, what is the encoded size for the sequence $X_1X_2X_3 = BAB$?

	$i=1$	$i=2$	$i=3$
X_{i-1}	A	B	A
X_i	B	A	B
before	$c(A,B)=1$	$c(B,A)=1$	$c(A,B)=2$
$\sum_{x\in\{A,B,C\}} c(x,X_i)$	3	3	4
$\hat{P}(X_i\|\text{past})$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{2}{4}$
after	$c(A,B)=2$	$c(B,A)=2$	$c(A,B)=3$

1. $\frac{1}{3}$
2. $\frac{2}{4}$
3. $\log_2(\frac{1}{3})+\log_2(\frac{1}{3})+\log_2(\frac{1}{2})\approx 4.17$

Quiz

Q1

Which lossless compressor is most suitable under following situations:

1. You know that the data is roughly 2nd order Markov but do not know the transition probabilities.

   ◯ context-based arithmetic coding
   ◯ context-based adaptive arithmetic coding

Quiz

Q1

Which lossless compressor is most suitable under following situations:

2. You know that the data is 3rd order Markov and know the exact distribution.
   - ◉ context-based arithmetic coding
   - ○ context-based adaptive arithmetic coding

Quiz

Q1

We didn't ask this in the quiz, but now is a good time

Which lossless compressor is most suitable under following situations:

3. You know nothing about the input data.

   ◯ LZ77
   ◯ context-based arithmetic coding
   ◯ context-based adaptive arithmetic coding

Universal Compression with Lempel-Ziv compression

Study *universal* compressors - a scheme that does *well* on **any** stationary input without prior knowledge of the source distribution.

As part of this - explore one of the most common schemes used in practical compressors!

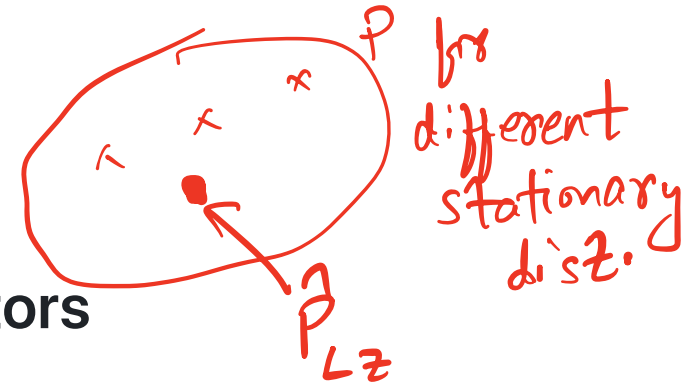
Universal compressor

Consider a compressor C that works on arbitrary length inputs and has length function $l(x^n)$.

> **Definition: Universal Compressor**
> C is universal if
>
> $$\lim_{n\to\infty} \frac{1}{n} E[l(X^n)] = H(\mathbf{X})$$
>
> for any stationary ergodic source.

So a single compressor C is asymptotically optimal for every stationary distribution without prior knowledge of the source distribution!

Thinking in terms of universal predictors



- Recall from last lecture that a compressor induces a distribution via it's length function: $\hat{p}(x^n) = 2^{-l(x^n)}$.
- A universal compressor's $\hat{p}$ approximates any stationary distribution arbitrarily closely as n grows!
- In particular a universal compressor is a universal predictor!

Thinking in terms of universal predictors

- Recall from last lecture that a compressor induces a distribution via it's length function: $\hat{p}(x^n) = l(x^n)$.
- A universal compressor's $\hat{p}$ approximates any stationary distribution arbitrarily closely as n grows!
- In particular a universal compressor is a universal predictor!

All this needs to be rigorously formulated, e.g., see the reference below, talk to Tsachy, and take EE 376C!

Ref: M. Feder, N. Merhav and M. Gutman, "Universal prediction of individual sequences," in IEEE Transactions on Information Theory, vol. 38, no. 4, pp. 1258-1270, July 1992, doi: 10.1109/18.144706.

Lempel-Ziv universal algorithms

- LZ77: in gzip, zstd, png, zip, lz4, snappy
- LZ78: strong theoretical guarantees
- LZW (Lempel-Ziv-Welch) (LZ78 variant): in linux compress utility, GIF
- LZMA (Lempel–Ziv–Markov chain algorithm) (LZ77 variant): 7-Zip, xz

References:

1. **LZ77:** Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." IEEE Transactions on information theory 23.3 (1977): 337-343.
2. **LZ78:** Ziv, Jacob, and Abraham Lempel. "Compression of individual sequences via variable-rate coding." IEEE transactions on Information Theory 24.5 (1978): 530-536.
3. **LZW:** Welch, Terry A. "A technique for high-performance data compression." Computer 17.06 (1984): 8-19.

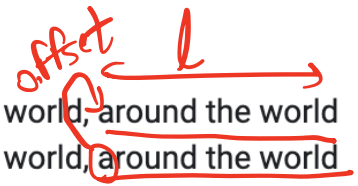
LZ77 algorithm

Simple idea: Replace repeated segments in data with pointers and lengths!

Around the World

Song by Daft Punk

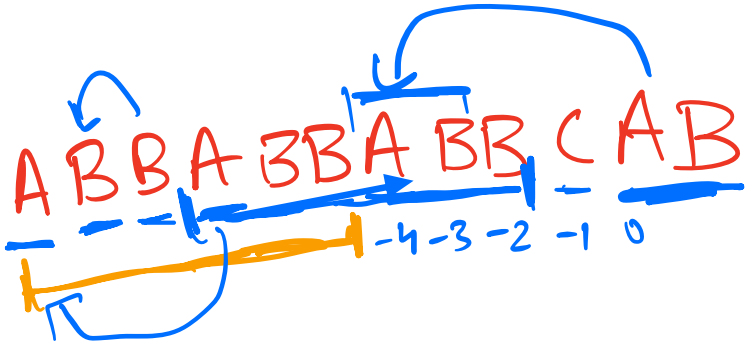
Lyrics



Around the world, around the world
Around the world, around the world
Around the world, around the world
Around the world, around the world
Around the world, around the world
Around the world, around the world
Around the world, around the world
Around the world, around the world

Around the world, around the world
Around the world, around the world

LZ77 parsing example



ABBABBABBCAB

Unmatched literals	Match length	Match offset
AB	1	1
x	6	3
C	2	4

LZ77 parsing example

A[B]BABBABBCAB

Unmatched literals	Match length	Match offset
AB	1	1
-	-	-
-	-	-

LZ77 parsing example

[ABBABB]ABBCAB

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
-	-	-

LZ77 parsing example

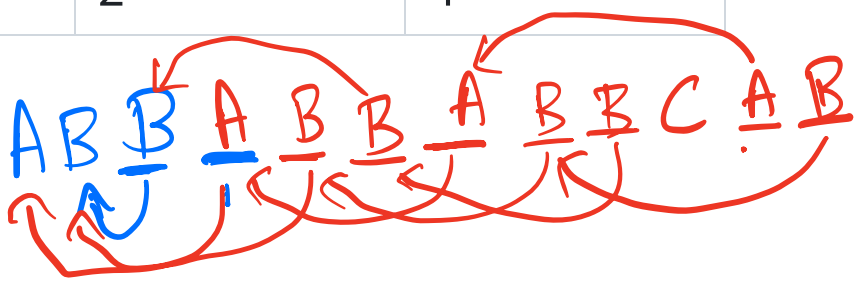
ABBABB[AB]BCAB

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
C	2	4

LZ77 unparsing example

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
C	2	4

Decoded:



LZ77 unparsing example

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
C	2	4

Decoded: ABB

LZ77 unparsing example

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
C	2	4

Decoded: ABBABBABB

LZ77 unparsing example

Unmatched literals	Match length	Match offset
AB	1	1
-	6	3
C	2	4

Decoded: ABBABBABBCAB

LZ77 parsing

Pseudocode:

```
For input sequence x[0], x[1], ...

Suppose we have parsed till x[i-1].

- Try to find largest k such that for some j < i
  x[j:j+k] = x[i:i+k]

- Then the match length is k and the match offset is i-j

  [note that the ranges j:j+k and i:i+k are allowed to overlap]

- If no match found, store as literal.
```

k = match length
i - j = offset

LZ77 unparsing

Pseudocode:

```
At each step:

- First read any literals and copy to output y.

- To decode a match with length l and offset o.
  - If l < o:
    - append y[-o:-o+l] to y
  - Else:
    // Need to be more careful with overlapping matches!
    - For _ in 0:l:
      - append y[-o] to y
```

Decompression is very fast since it just involves copying!

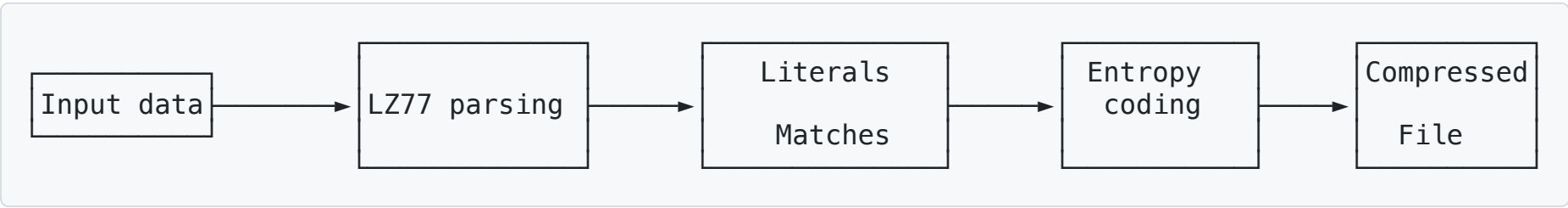
Quiz question

Apply the above parsing and unparsing algorithms for the following:

1. Parse **AABBBBBBBAABBBCDCDCD**.
2. Unparse the below table (note that this parsing was generated using a different parser than the one described above!):

Unmatched literals	Match length	Match offset
AABBB	4	1
-	5	9
CDCD	2	2

Encoding step



- Need to encode the literals, match lengths and match offsets.
- Implementations (gzip, zstd, etc.) differ in the approach.
- Typically use Huffman coding/ANS with some modifications to optimize for real-life data.
- More on this in a bit!

LZ77 universality proof idea

Question:

Consider iid sequence $\ldots, X_{-2}, X_{-1}, X_0, X_1, X_2, \ldots$

If symbol a has probability $P(a)$, what's the expected gap between consecutive occurences of a?

$n = 1000$ – total

$nP(a) = 200$ – a's

average gap $\rightarrow 5 = \frac{1000}{200}$

LZ77 universality proof idea

Question:

Consider iid sequence $\ldots, X_{-2}, X_{-1}, X_0, X_1, X_2, \ldots$

If symbol a has probability $P(a)$, what's the expected gap between consecutive occurences of a?

$nP(a)$

Hint: In a block of size n, how many times do you expect to see a? What's the average spacing between the occurences?

LZ77 universality proof idea

Question:

Consider iid sequence $\ldots, X_{-2}, X_{-1}, X_0, X_1, X_2, \ldots$

If symbol a has probability $P(a)$, what's the expected gap between consecutive occurences of a?

Hint: In a block of size n, how many times do you expect to see a? What's the average spacing between the occurences?

Answer: (using law of large numbers and the iid-ness)
You expect to see a around $nP(a)$ times, and the average spacing is $\frac{n}{nP(a)} = \frac{1}{P(a)}$.

LZ77 universality proof idea

This generalizes to stationary ergodic processes.

> **Kac's Lemma**
> Let $\ldots, X_{-2}, X_{-1}, X_0, X_1, X_2, \ldots$ be a stationary ergodic process and let $R_n(X_0, \ldots, X_{n-1})$ be the recurrence time (last time $X_0, \ldots, X_{n-1}$ occurred before index 0). Given that $(X_0, \ldots, X_{n-1}) = x_0^{n-1}$, we have
>
> $$E[R_n(X_0, \ldots, X_{n-1})] = \frac{1}{p(x_0^{n-1})}$$

In simple words, if $(X_0, \ldots, X_{n-1}) = x_0^{n-1}$, the same sequence x_0^{n-1} also occured roughly $\frac{1}{p(x_0^{n-1})}$ positions ago. Thus the match offset in LZ77 is $\frac{1}{p(x_0^{n-1})}$.

LZ77 universality proof idea

$n \rightarrow \log_2 n$ bits

- Can encode the match offset $\frac{1}{p(x_0^{n-1})}$ using close to $\log_2 \frac{1}{p(x_0^{n-1})}$ bits using an appropriate integer coder (e.g., check out the Elias Delta code in SCL).

LZ77 universality proof idea

- Can encode the match offset $\frac{1}{p(x_0^{n-1})}$ using close to $\log_2 \frac{1}{p(x_0^{n-1})}$ bits using an appropriate integer coder (e.g., check out the Elias Delta code in SCL).
- Match length and literal contribution is negligible!

LZ77 universality proof idea

- Can encode the match offset $\frac{1}{p(x_0^{n-1})}$ using close to $\log_2 \frac{1}{p(x_0^{n-1})}$ bits using an appropriate integer coder (e.g., check out the Elias Delta code in SCL).
- Match length and literal contribution is negligible!
- Expending $\log_2 \frac{1}{p(x_0^{n-1})}$ bits means we are following the thumb rule, and we use on average $E[l(X^n)] \approx E[\log_2 \frac{1}{p(x_0^{n-1})}] = H(X^n)$

LZ77 universality proof idea

- Can encode the match offset $\frac{1}{p(x_0^{n-1})}$ using close to $\log_2 \frac{1}{p(x_0^{n-1})}$ bits using an appropriate integer coder (e.g., check out the Elias Delta code in SCL).
- Match length and literal contribution is negligible!
- Expending $\log_2 \frac{1}{p(x_0^{n-1})}$ bits means we are following the thumb rule, and we use on average $E[l(X^n)] \approx E[\log_2 \frac{1}{p(x_0^{n-1})}] = H(X^n)$
- Taking this to the limit, we can see that LZ77 achieves the entropy rate!

LZ77 universality proof idea

For a more detailed and rigorous proof, check out Cover and Thomas chapter 13 or A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," in Proceedings of the IEEE, vol. 82, no. 6, pp. 872-877, June 1994, doi: 10.1109/5.286191.

LZ77 universality proof idea

Asymptotic theory doesn't fully explain the excellent performance in practice:

- for a kth order Markov process, you expect LZ77 to do well in the limit, but not amazing for reasonable sized data
- the idea of finding matches is just very well-matched to real-life data and the data is not always modeled easily as a kth order Markov process.

LZ77 parsing on real data - examples

Let's look at how matches look in practice and how the match lengths and offsets are typically distributed.

We use the LZ77 implementation in SCL for this purpose.

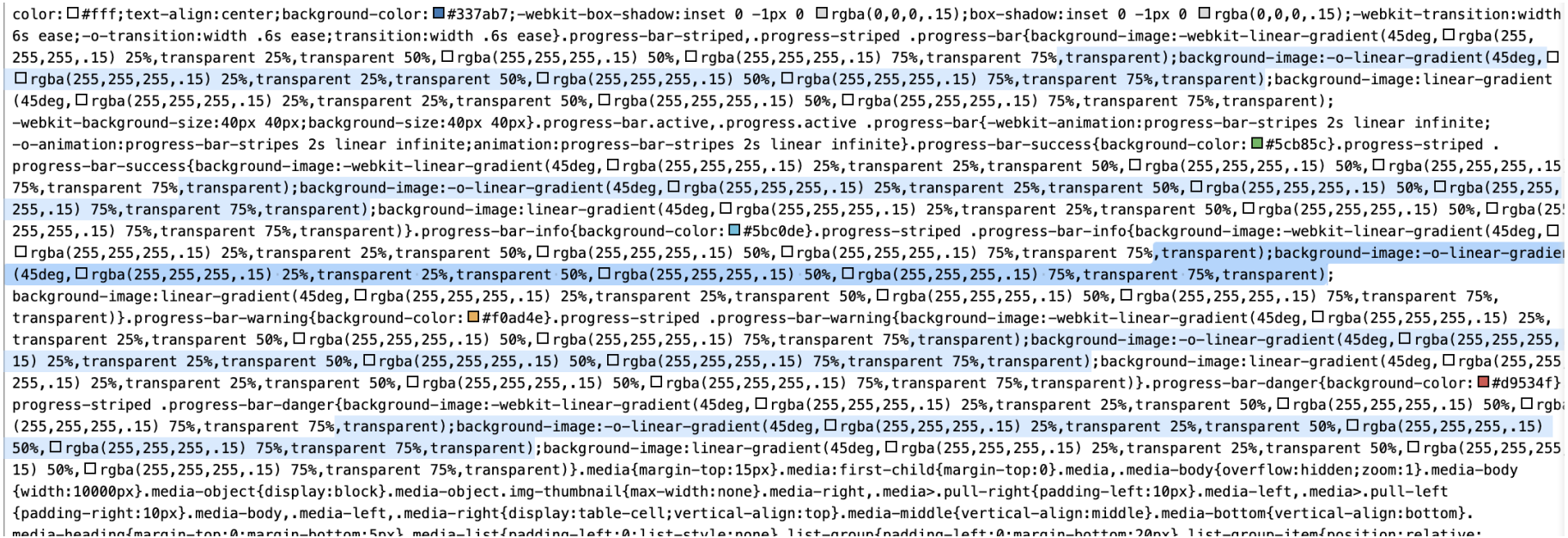
LZ77 parsing on real data - examples

Long matches:

```
`Beautiful Soup, so rich and green,
Waiting in a hot tureen!
Who for such dainties would not stoop?
Soup of the evening, beautiful Soup!
Soup of the evening, beautiful Soup!
    Beau--ootiful Soo--oop!
    Beau--ootiful Soo--oop!
Soo--oop of the e--e--evening,
    Beautiful, beautiful Soup!

`Beautiful Soup!  Who cares for fish,
Game, or any other dish?
Who would not give all else for two p
ennyworth only of beautiful Soup?
Pennyworth only of beautiful Soup?
    Beau--ootiful Soo--oop!
    Beau--ootiful Soo--oop!
Soo--oop of the e--e--evening,
    Beautiful, beauti--FUL SOUP!'
```

LZ77 parsing on real data - examples

Long matches:



LZ77 parsing on real data - examples

3 bits/byte.
pleasure → 24 bits

Far off matches (150 KB apart) [`pleasure`]:

First page:

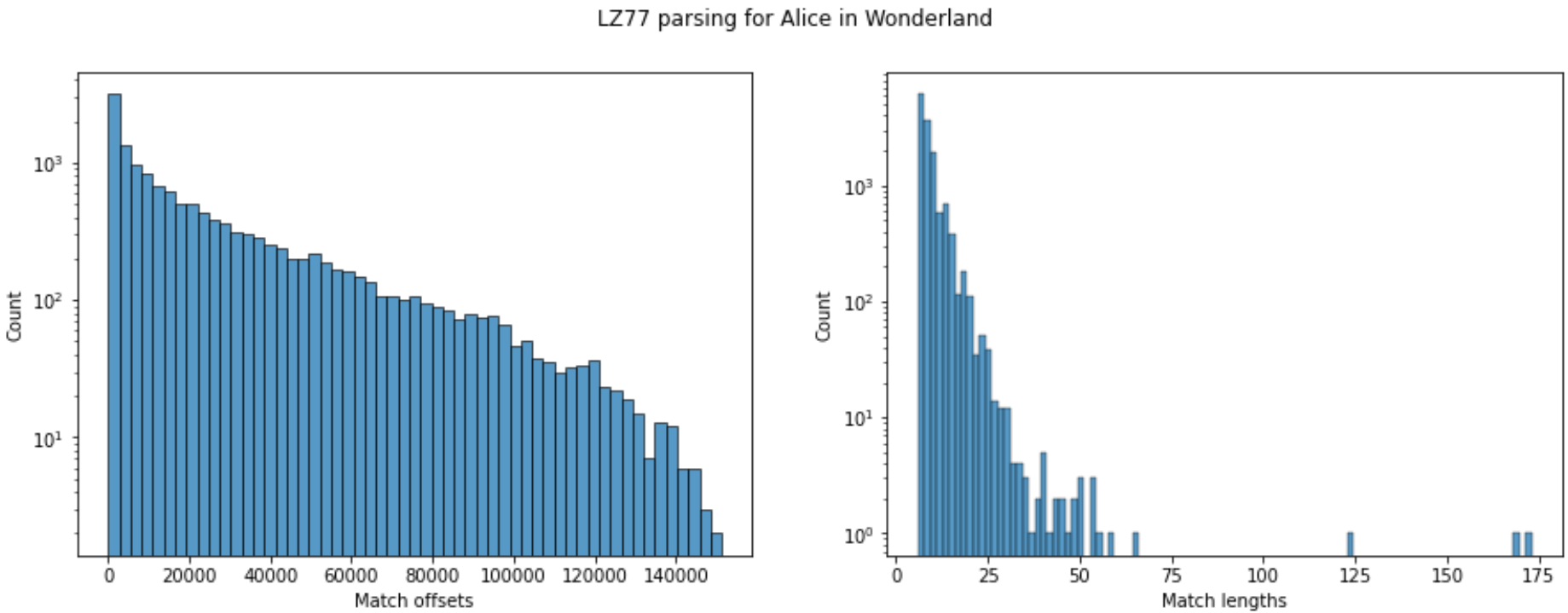
```
So she was considering in her own mind (as well as she could,
for the hot day made her feel very sleepy and stupid), whether
the pleasure of making a daisy-chain would be worth the trouble
of getting up and picking the daisies, when suddenly a White
Rabbit with pink eyes ran close by her.
```

150 KB → 150000
$\log_2(150{,}000) \approx 17.2$ bits

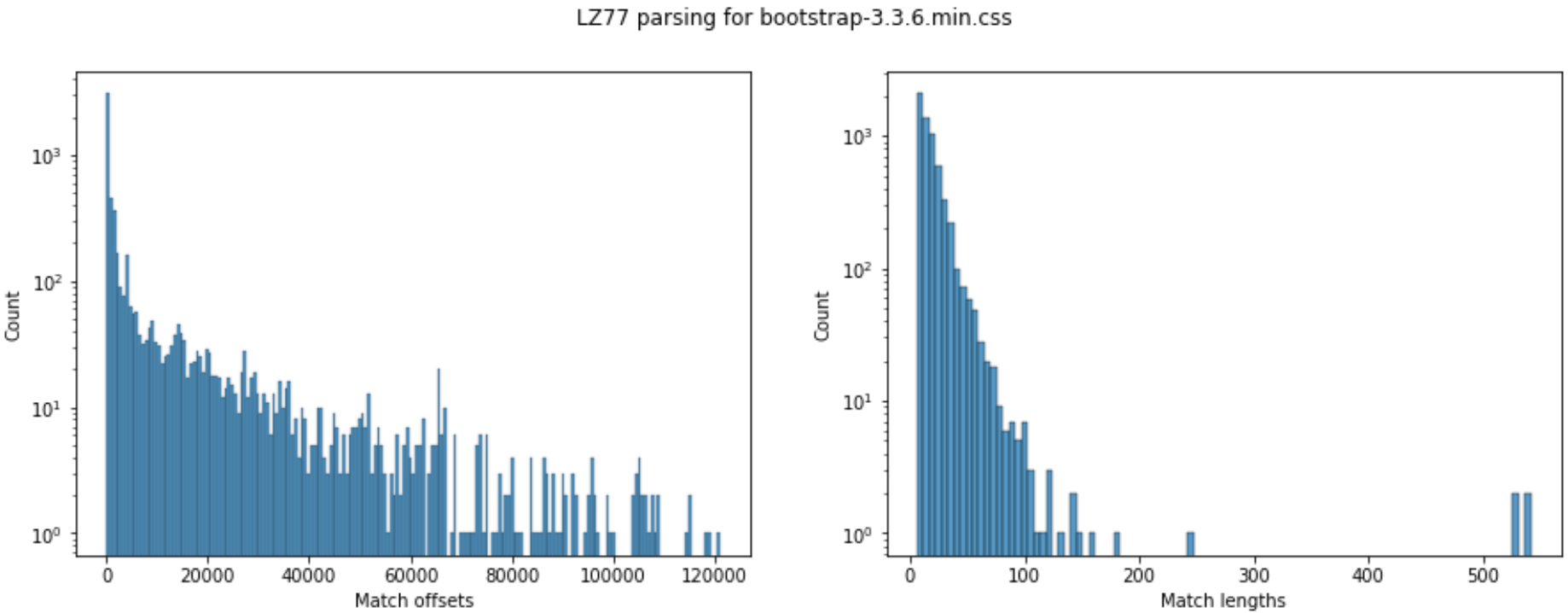
Last page:

```
, and make THEIR eyes bright and eager
with many a strange tale, perhaps even with the dream of
Wonderland of long ago:  and how she would feel with all their
simple sorrows, and find a pleasure in all their simple joys,
remembering her own child-life, and the happy summer days.

                                THE END
```

LZ77 parsing on real data - examples



LZ77 parsing on real data - examples

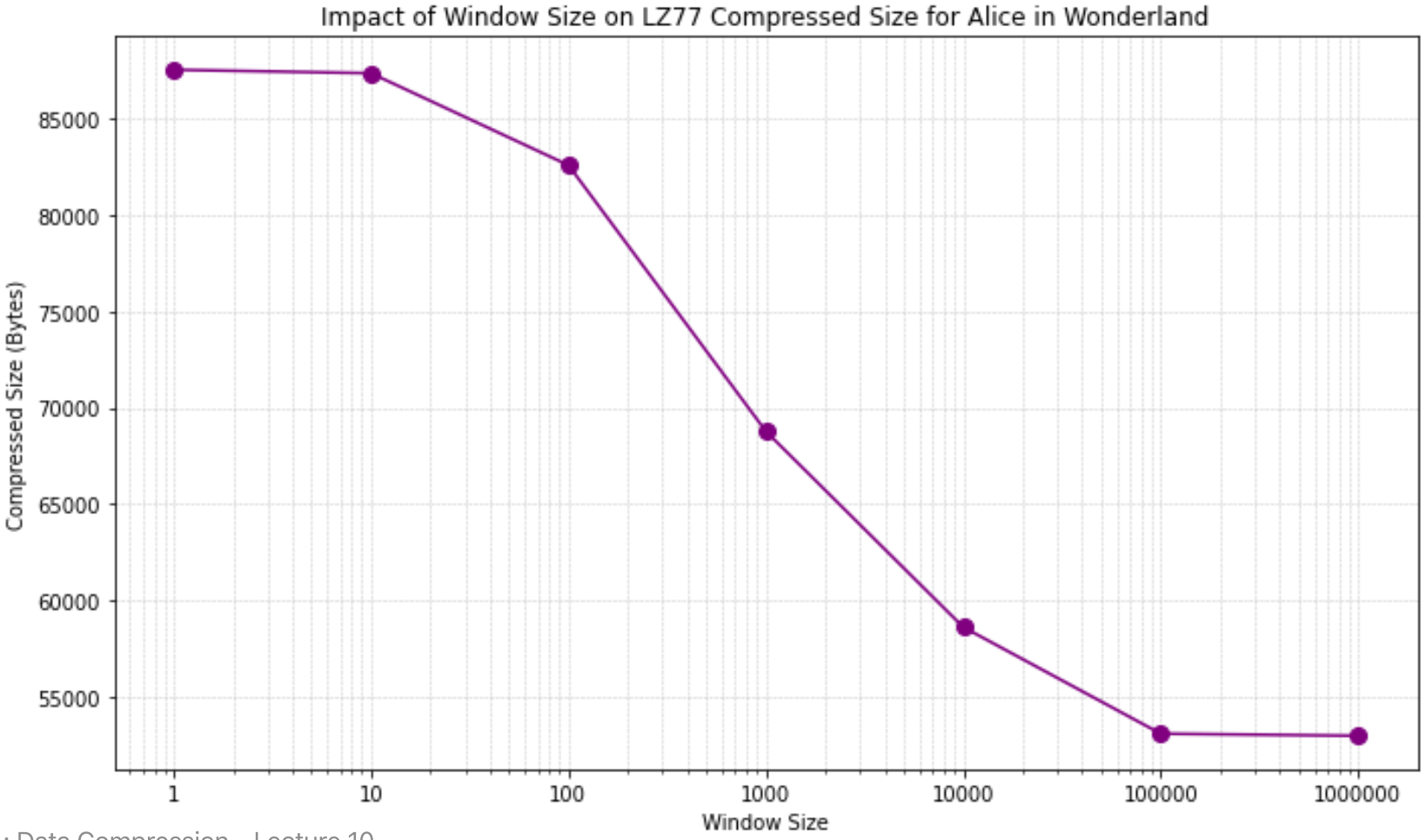


Practical considerations - sliding window

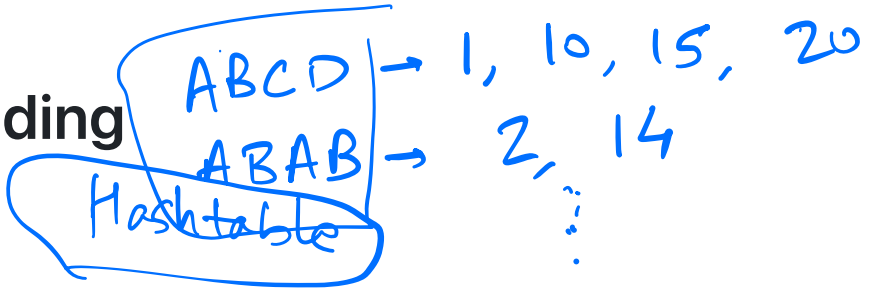
Do I need to keep infinite past memory?

- Use sliding window - only find matches in past 10s of KBs (gzip) to multiple MBs (zstd) window.
- Typically use circular buffer to efficiently handle windows without reallocation.
- Bigger window gives better compression but needs more memory for both compression and decompression.
- See `LZ77Window` and `LZ77SlidingWindowEncoder` / `LZ77SlidingWindowDecoder` in SCL to understand this better.

Practical considerations - sliding window

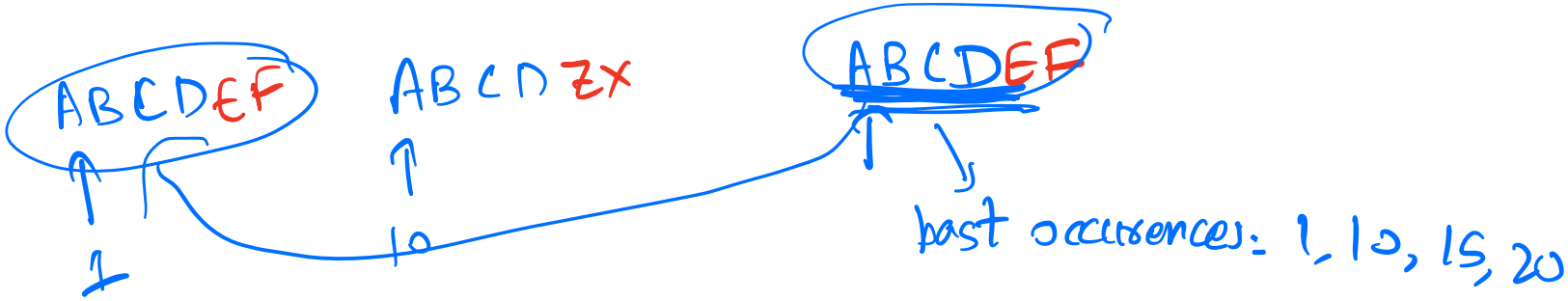


Practical considerations: match finding

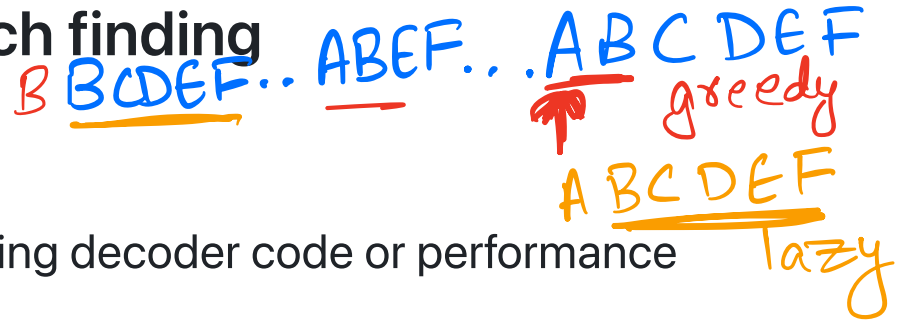


How to find matches?

- Many ways to do this without affecting decoder code or performance
- Basic idea:
  - index past occurences of sequences (e.g., 4-length substrings) in a data structure like hash table/binary tree
  - for the given position, do a lookup to find previous occurences and then extend the candidate match to find longest match



Practical considerations: match finding



How to find matches?

- Many ways to do this without affecting decoder code or performance
- Basic idea:
  - index past occurences of sequences (e.g., 4-length substrings) in a data structure like hash table/binary tree
  - for the given position, do a lookup to find previous occurences and then extend the candidate match to find longest match
- Greedy not always the best
  - can incur a literal to find a longer match at the next position (e.g., lazy strategies don't immediately take a match, instead look ahead to find if there's a longer one)
  - strategies range from fast and greedy to slow and optimal (e.g., using dynamic programming to find the "optimum" parsing)

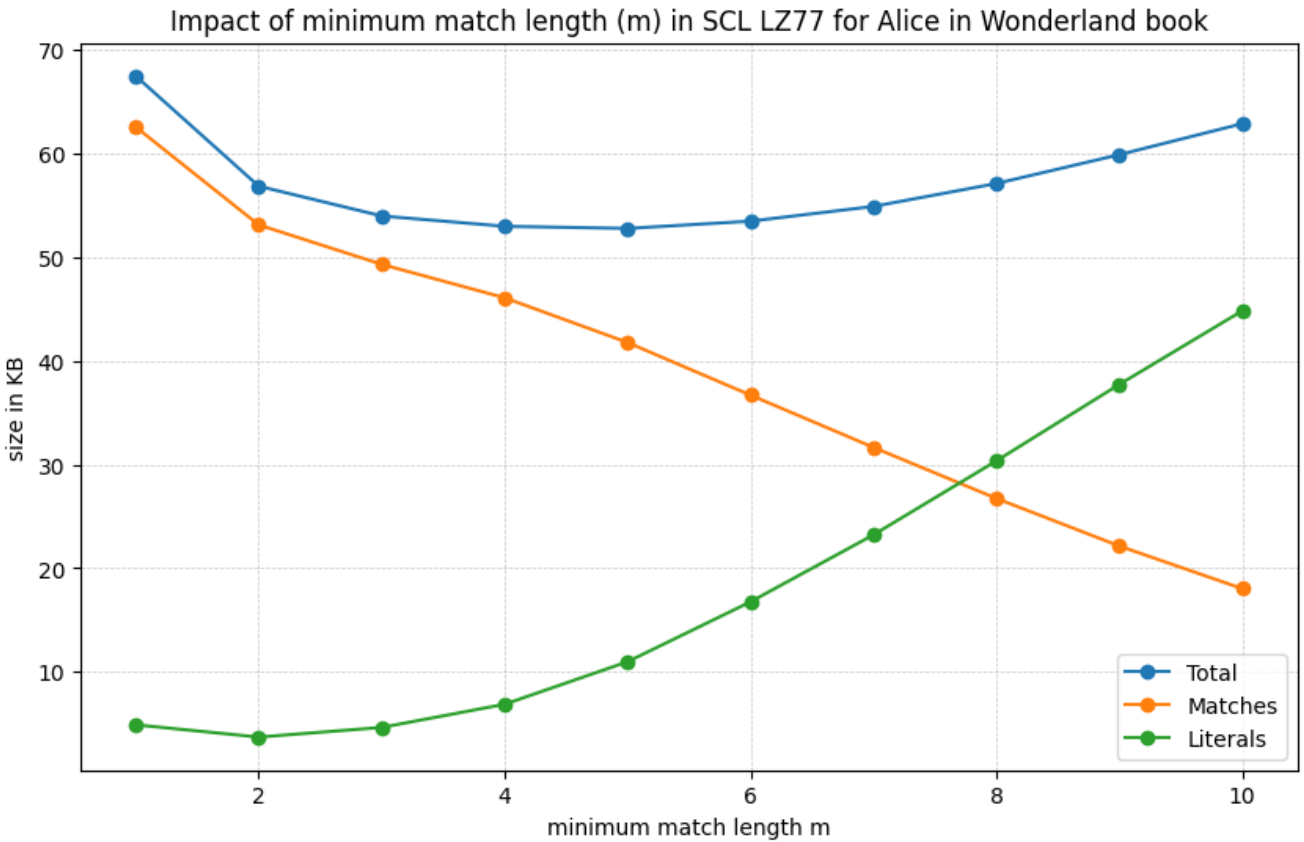
SCL: LZ77SlidingWindowEncoder

```
class LZ77SlidingWindowEncoder:
    - match_finder: Match finder used for encoding (not required for decoding)
    - window_size: size of sliding window (maximum lookback)
```

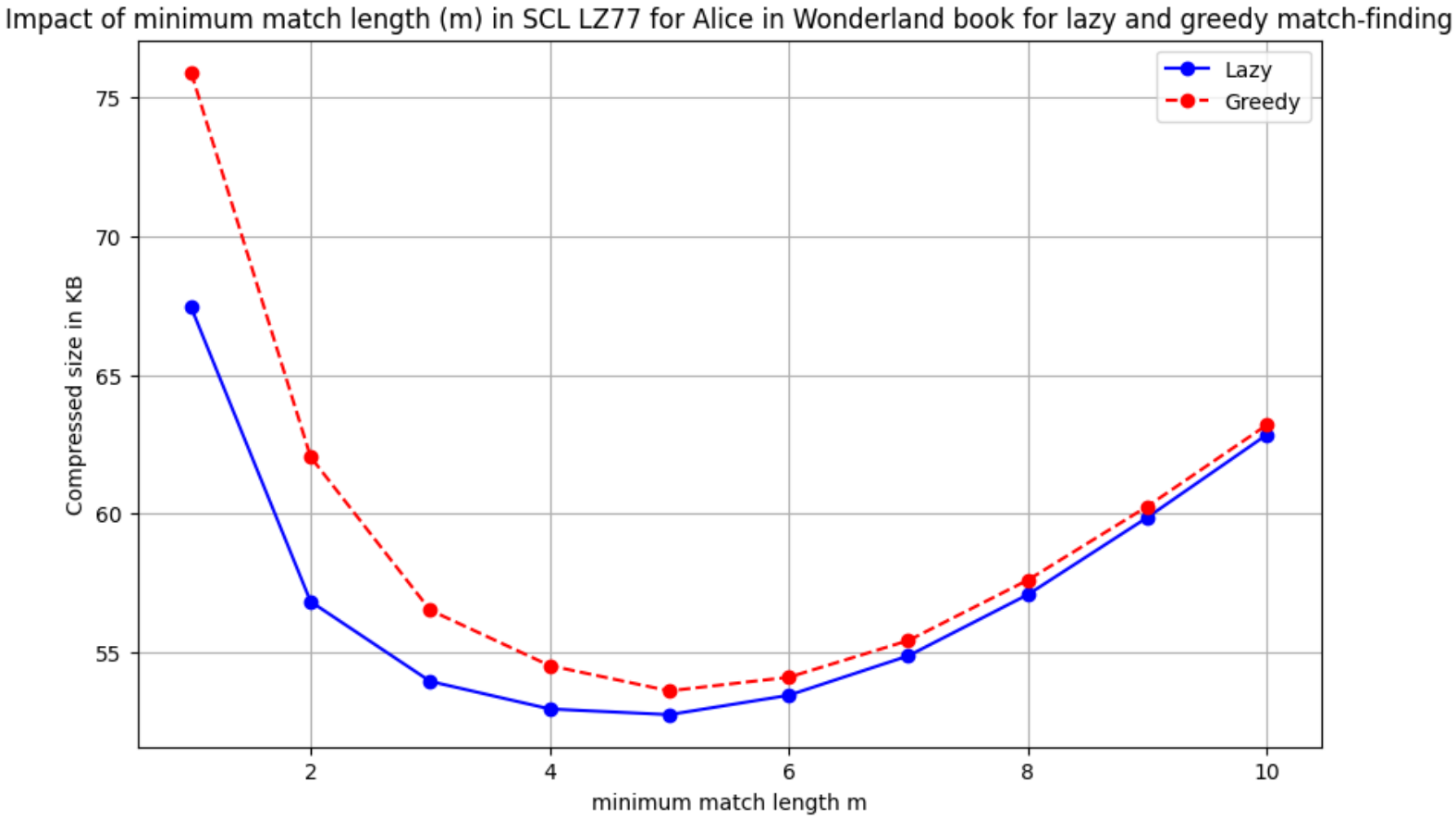
```
class MatchFinderBase:
      def extend_match(
        self, match_start_in_lookahead_buffer, match_pos, lookahead_buffer, left_extension=True)

      def find_best_match(self, lookahead_buffer)
```

```
class HashBasedMatchFinder(MatchFinderBase):
    - hash_length (int): The length of byte sequences to hash.
    - hash_table_size (int): Size of the hash table.
    - max_chain_length (int): Maximum length of the chains in the hash table.
    - lazy (bool): Whether to use lazy matching where LZ77 considers one step ahead and skips a literal if it finds a longer match.
    - minimum_match_length (int): Minimum length of a match to be considered.
```

Practical considerations: match finding



Practical considerations: match finding



Entropy coding

Unmatched literals	Match length	Match offset
AABBB	4	1
-	5	9
CDCD	2	2

encoded as

→ AABBBCDCD

Entropy coding

```
literals = AABBBCDCD
```

and

Literal counts	Match length	Match offset
5	4	1
0	5	9
4	2	2

Entropy coding

Each of streams encoded using various entropy coding approaches.

- Huffman - dynamic/static
  - gzip exclusively relies on this
  - zstd - Huffman only for literals
- tANS: zstd uses tANS for ~~literals and~~ match lengths & offsets
- Context-based arithmetic coding (slower to encode and decode): LZMA
- For very high speeds, skip entropy coding and use fixed length codes! (LZ4, Snappy)

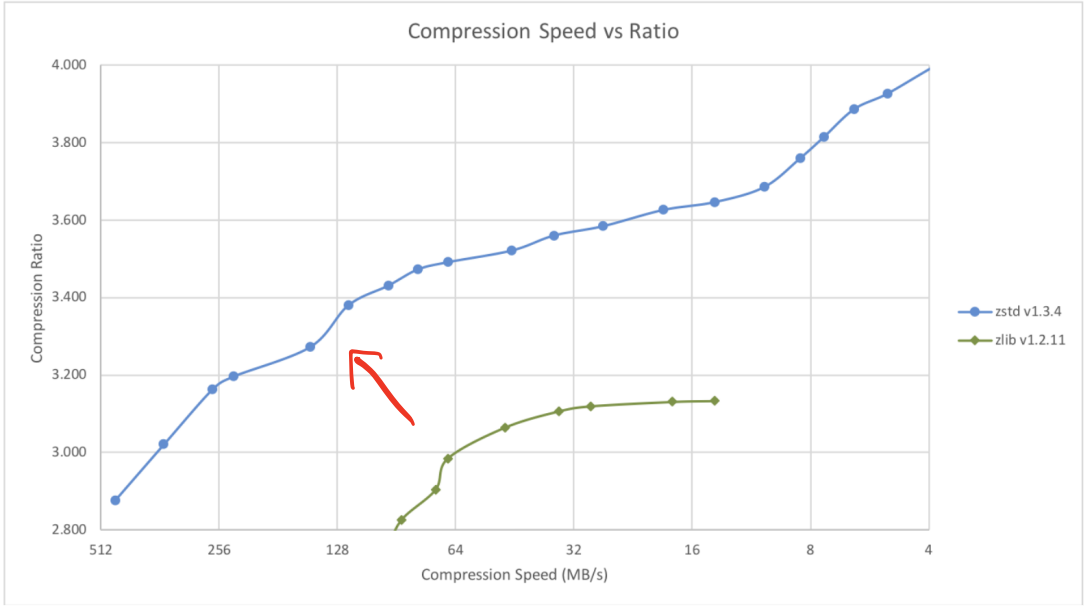
Entropy coding of integers

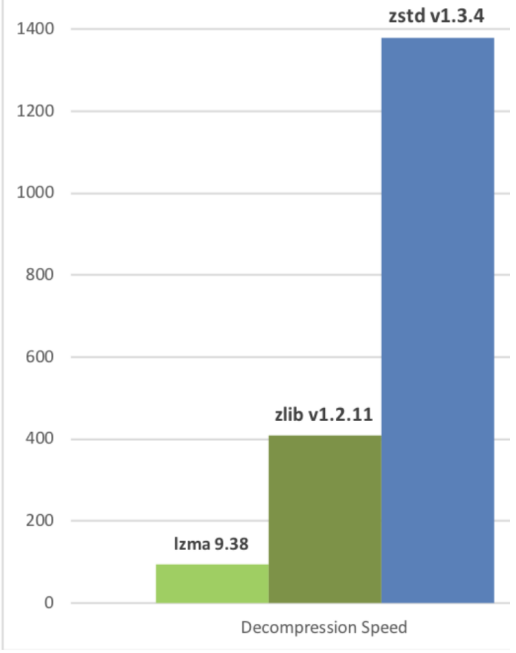
- Match lengths/offsets have a large range (can be up to millions).
- Naively applying Huffman codes on such a large alphabet inefficient due to lack of counts (e.g., a specific match offset like 14546 might not occur many times).
- Typical approach: divide the integers into bins: entropy code bin index and encode the position in bin using plain old fixed length code.
- E.g., bins 2-3, 4-7, 8-15 and if in bin 8-15 use 3 bits to encode within bin position.
- Can imagine this as a Huffman tree where you first have the bins as leaves and then attach a complete binary tree to each of these.
- Check out `LogScaleBinnedIntegerEncoder` in SCL LZ77 implementation.

Practical considerations

- Parsing/match finding strategy, window size, entropy coding, implementation details (SIMD and more) matters a lot in determining speed and memory usage.

github.com/facebook/zstd





LZ77 implementation benchmark (https://facebook.github.io/zstd/)

Compressor name	Ratio	Compression	Decompress.
zstd 1.4.5 -1	2.884	500 MB/s	1660 MB/s
zlib 1.2.11 -1	2.743	90 MB/s	400 MB/s
brotli 1.0.7 -0	2.703	400 MB/s	450 MB/s
zstd 1.4.5 --fast=1	2.434	570 MB/s	2200 MB/s
zstd 1.4.5 --fast=3	2.312	640 MB/s	2300 MB/s
quicklz 1.5.0 -1	2.238	560 MB/s	710 MB/s
zstd 1.4.5 --fast=5	2.178	700 MB/s	2420 MB/s
lzo1x 2.10 -1	2.106	690 MB/s	820 MB/s
lz4 1.9.2	2.101	740 MB/s	4530 MB/s
lzf 3.6 -1	2.077	410 MB/s	860 MB/s
snappy 1.1.8	2.073	560 MB/s	1790 MB/s

Universal doesn't mean perfect

- LZ77 is universal in an asymptotic sense, but need not be the best choice for a given dataset.
- There are things to consider beyond the compression rate - speed, memory usage and so on.
- Choices (discussed above) made in LZ77 implementations play a big role in its performance.

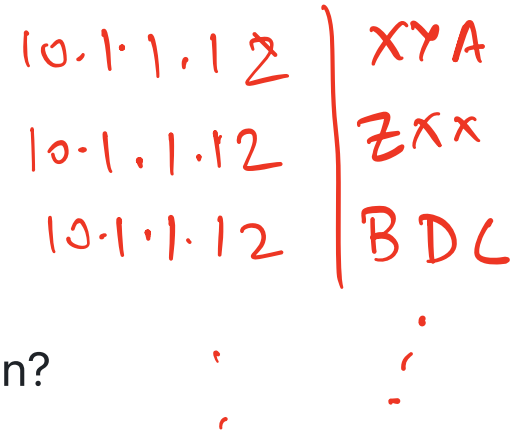
That's it on LZ!

- We didn't talk about LZ78 and LZW - similar core ideas but slightly different tree-based parsing method
  - For LZ78, possible to prove very powerful universality results, including non-asymptotic ones!
  - In particular can show that LZ78 gets compression rate within $O(\frac{k}{\log n} + \frac{\log \log n}{\log n})$ of the optimal kth order model for any sequence.
  - Learn more in EE 376C.
- Much more on LZ77 we didn't cover (e.g., repcodes, optimal parsing)

Great project ideas!

Practical tips on lossless compression

Is it possible you don't need all this data?



- Identify parts of data that are costing the most:
  - Can you get rid of it or change the representation?
  - Does the identifier need to be so random?
- Do you need to store the data for so long?
  - Sometimes legal requirements
- What is your data access pattern?
  - Backup/archive (cold storage) cheaper than hot storage

Also: lossy compression - starting next lecture!

Compression is not only about storage

- memory reduction
- bandwidth reduction
- faster querying

You definitely need to store this data losslessly! Now what?

Things not to do

- Use your recently acquired EE274 knowledge and straightaway start spending a bunch of time developing your own compressor
- Implement LZ77 and all the entropy coders on your own
- Search online and purchase the license to a fancy looking compression solution
- Decide to use gzip with default settings since that's the popular thing

All of these can make sense, but not as the first step!

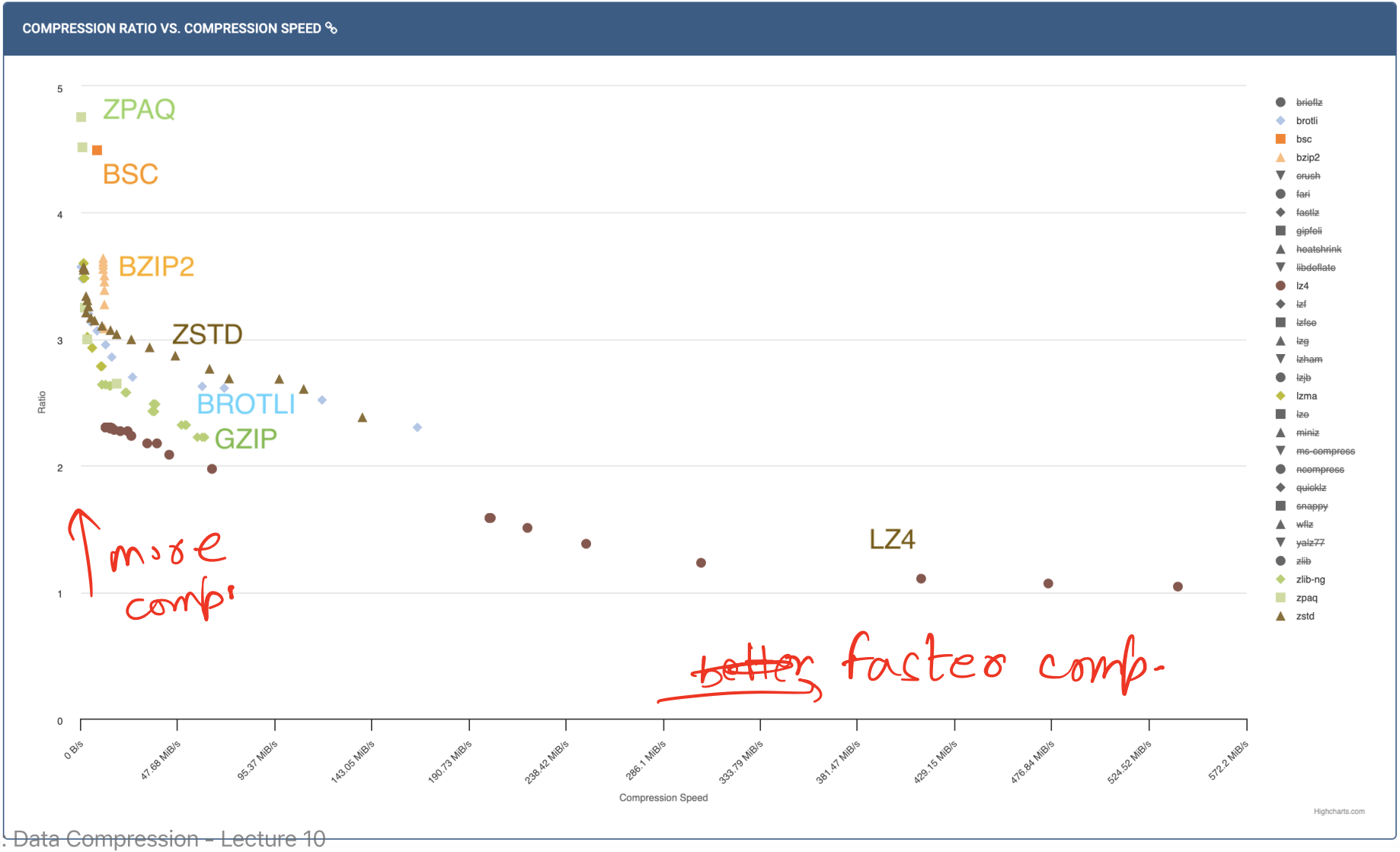
Understand your application

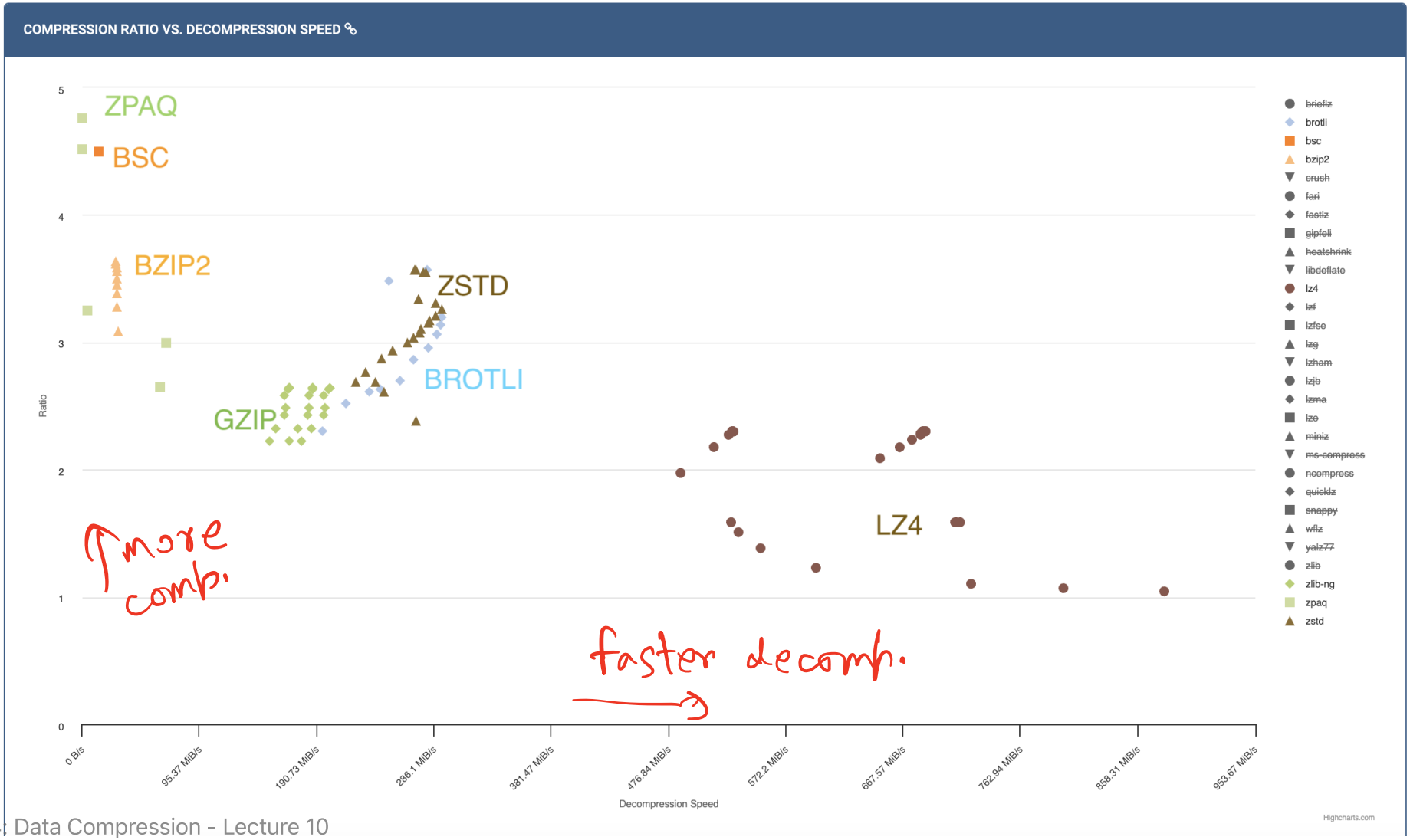
- Resource requirements - compression/decompression speed/memory
- Where will compression and decompression happen?
  - Closed system owned by you? Deployed by customers? Open source library?
- How will compression and decompression happen?
  - CLI? Library in some programming language?
- Is the data homogenous? Is there a lot of the same kind of data?

All these would affect your choice of compressor (or your decision to build your own), as well as how to deploy it.

Using compression benchmarks

- http://quixdb.github.io/squash-benchmark/unstable/
- Measure compression ratio as well as compression and decompression speed on your specific dataset or on a general corpus
- Find the most prefered operating point


COMPRESSION RATIO VS. COMPRESSION SPEED
ZPAQ
BSC
BZIP2
ZSTD
BROTLI
GZIP
LZ4
Ratio
Compression Speed
0 B/s
47.68 MiB/s
95.37 MiB/s
143.05 MiB/s
190.73 MiB/s
238.42 MiB/s
286.1 MiB/s
333.79 MiB/s
381.47 MiB/s
429.15 MiB/s
476.84 MiB/s
524.52 MiB/s
572.2 MiB/s
brieflz
brotli
bsc
bzip2
crush
fari
fastlz
gipfeli
heatshrink
libdeflate
lz4
lzf
lzfse
lzg
lzham
lzjb
lzma
lzo
miniz
ms-compress
ncompress
quicklz
snappy
wflz
yalz77
zlib
zlib-ng
zpaq
zstd
more comp.
better
faster comp.
Highcharts.com


COMPRESSION RATIO VS. DECOMPRESSION SPEED
ZPAQ
BSC
BZIP2
ZSTD
BROTLI
GZIP
LZ4
more comp.
faster decomp.
Ratio
Decompression Speed
0 B/s
95.37 MiB/s
190.73 MiB/s
286.1 MiB/s
381.47 MiB/s
476.84 MiB/s
572.2 MiB/s
667.57 MiB/s
762.94 MiB/s
858.31 MiB/s
953.67 MiB/s
brieflz
brotli
bsc
bzip2
crush
fari
fastlz
gipfeli
heatshrink
libdeflate
lz4
lzf
lzfse
lzg
lzham
lzjb
lzma
lzo
miniz
ms-compress
ncompress
quicklz
snappy
wflz
yalz77
zlib
zlib-ng
zpaq
zstd
Highcharts.com

General rule of thumb

- zstd has very fast decompression, and a variety of compression levels - should be the first thing to try. Don't use gzip unless you have a very good reason.
- To go even faster, try LZ4.

General rule of thumb

- zstd has very fast decompression, and a variety of compression levels - should be the first thing to try. Don't use gzip unless you have a very good reason.
- To go even faster, try LZ4.
- For slower but better compression, try out:
  - LZMA based compressors (7-zip, xz)
  - BWT based compressors (bzip2, bsc) - faster compression (than LZMA), slower decompression [more on BWT in HW 3!]
- For even more resource intensive - try context based arithmetic coding with kth order, PPMd, etc. (these don't always beat LZ/BWT)
- CMIX/NNCP/LLM - if you want to understand the limits!

Using zstd and using it well

- Use the latest version
- Choose the right level (1-19, also negative and ultra levels)
  - sometimes makes sense to use custom advanced parameters (e.g., window size, hash length)
- Use CLI/library in language of your choice - zstd also supports multithreaded compression
- Reuse zstd context objects if processing multiple files
- For small files, consider using dictionary compression (HW3 problem!)

When does a domain specific compressor make sense?

- noticeable gap between existing compressors and estimated entropy (theory super useful here!)
- do well while being faster than existing (using the structure of the data)
- easier if closed ecosystem
- lots of data of the same type making investment worth it
    - design
    - implement
    - test
    - format and versioning
    - monitoring and maintenance
    - portability and language support

That's all on lossless compression!

Thank you :)