SEA 2024

# Top-*k* Frequent Patterns in Streams and Parameterized-Space LZ Compression

**Patrick Dinklage** Johannes Fischer Nicola Prezza

technische universität
dortmund

Università
Ca'Foscari
Venezia

# Lempel-Ziv (LZ) Compression

`panaman_ananas_banana_pancake`

# Lempel-Ziv (LZ) Compression

panaman_ananas_banana_pancake

↓

panam(4,2)_ananas_banana_pancake

# Lempel-Ziv (LZ) Compression

panaman_ananas_banana_pancake

⇩

panam(4,2)_ananas_banana_pancake

# Lempel-Ziv (LZ) Compression

`panaman_ananas_banana_pancake`

↓

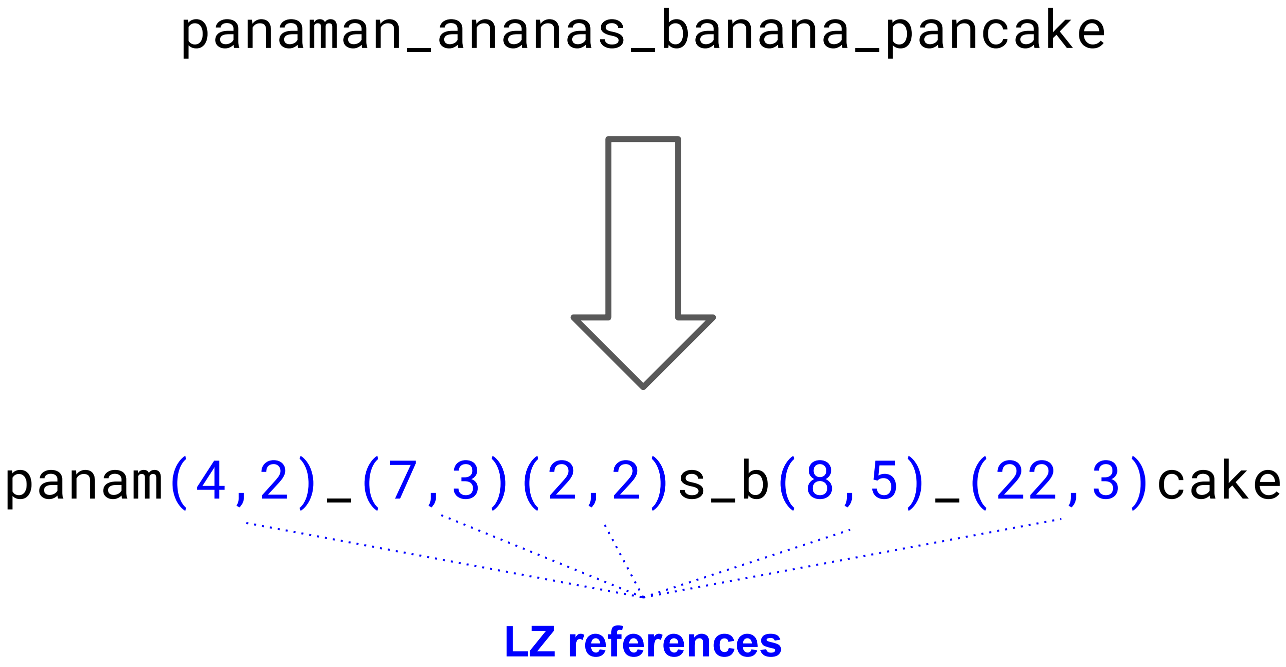
`panam(4,2)_(7,3)nas_banana_pancake`

# Lempel-Ziv (LZ) Compression

```
panaman_ananas_banana_pancake
```

```
panam(4,2)_(7,3)(2,2)s_b(8,5)_(22,3)cake
```

# Lempel-Ziv (LZ) Compression



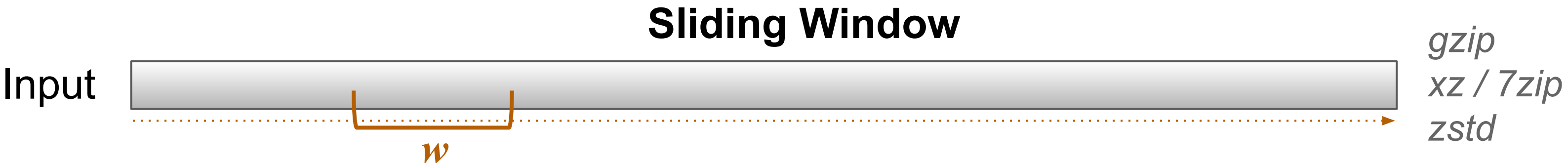
panaman_ananas_banana_pancake

panam(4,2)_(7,3)(2,2)s_b(8,5)_(22,3)cake

**LZ references**

# Compression in parameterized space

# Compression in parameterized space

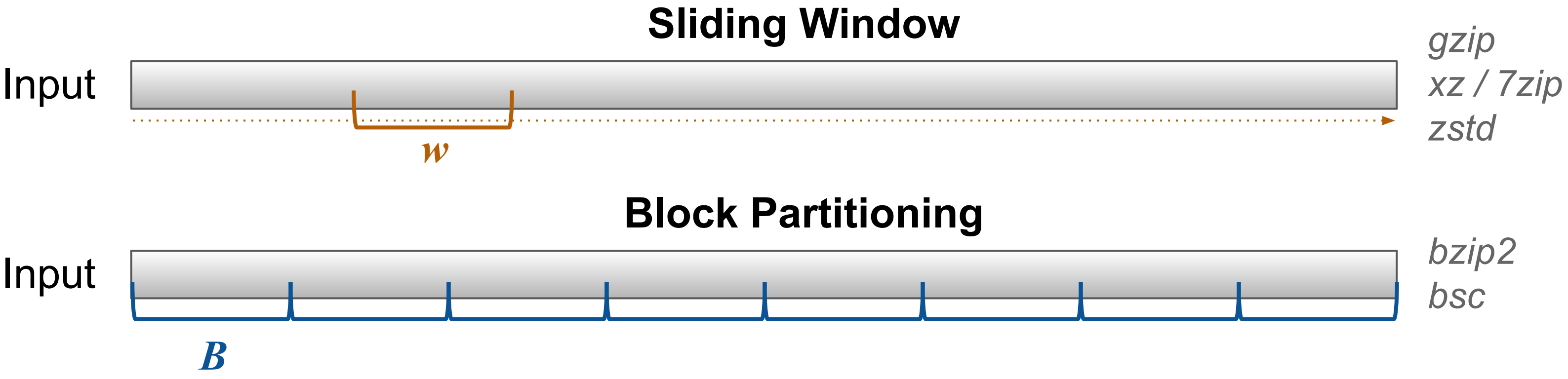


Sliding Window

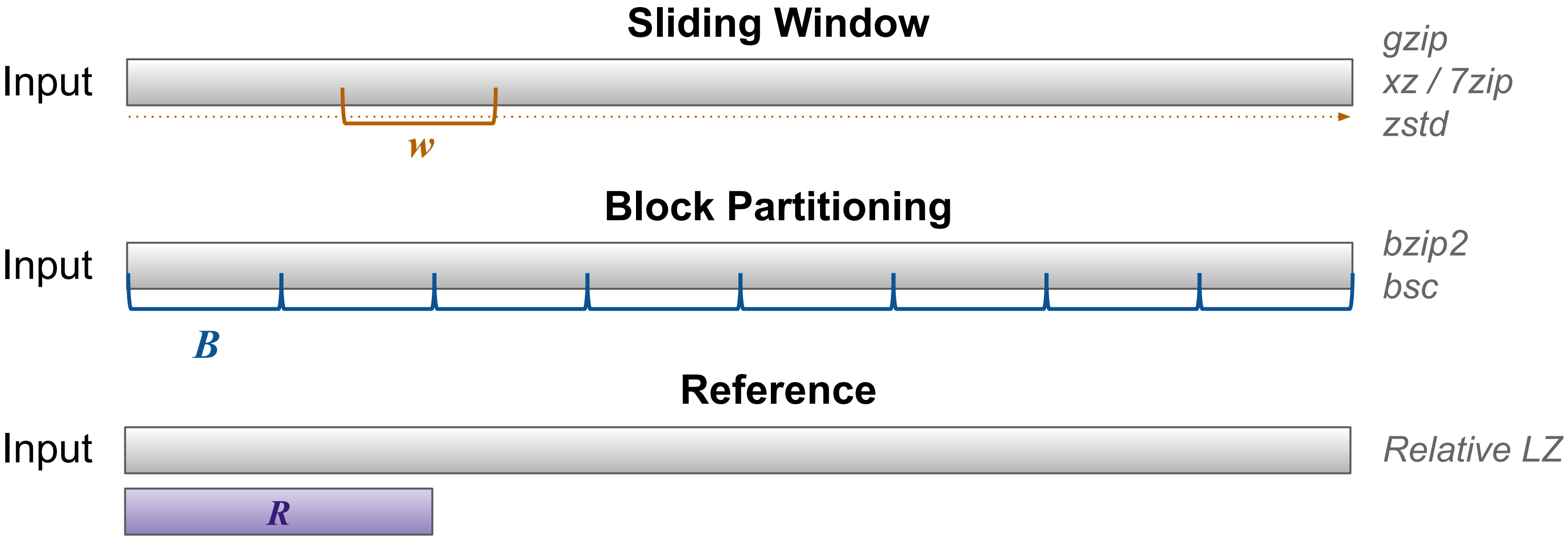
Input

*gzip*
*xz / 7zip*
*zstd*

# Compression in parameterized space



# Compression in parameterized space



### Sliding Window

Input

*gzip*
*xz / 7zip*
*zstd*

### Block Partitioning

Input

*bzip2*
*bsc*

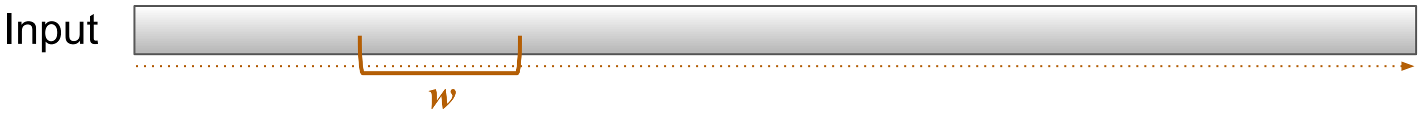
### Reference

Input

*Relative LZ*

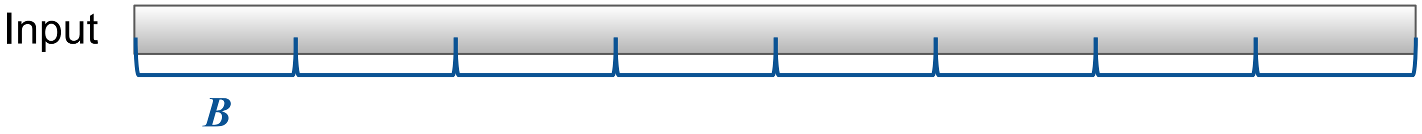
# Compression in parameterized space

### Sliding Window



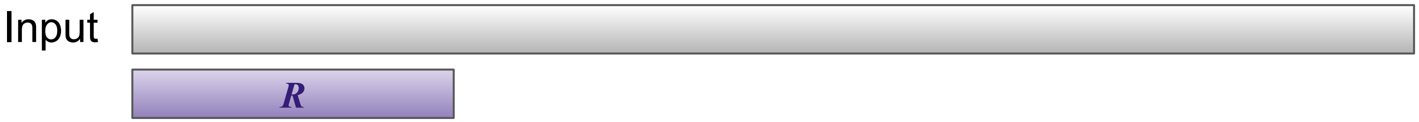
*gzip*
*xz / 7zip*
*zstd*

### Block Partitioning



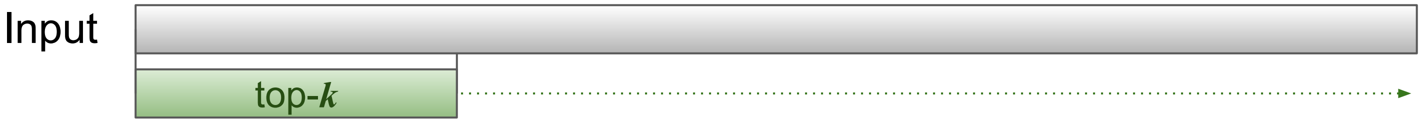
*bzip2*
*bsc*

### Reference



*Relative LZ*

### Frequent Patterns



***this talk!***

# Frequent Patterns

→ We look for frequent **patterns** in a stream $\boldsymbol{S}$ of characters

# Frequent Patterns

→ We look for frequent **patterns** (consecutive substrings) in a stream $\boldsymbol{S}$ of characters

# Frequent Patterns

→ We look for **frequent patterns** (consecutive substrings) in a stream $\boldsymbol{S}$ of characters

```
panaman_ananas_banana_pancake
```

# Frequent Patterns

→ We look for **frequent patterns** (consecutive substrings) in a stream ***S*** of characters

panaman_ananas_banana_pancake

# Frequent Patterns

→ We look for **frequent patterns** (consecutive substrings) in a stream $S$ of characters

`panaman_ananas_banana_pancake`

# Compressing with Frequent Patterns

**Idea:**

- Estimate online which $k$ patterns are the most frequent
- Replace occurrences of frequent patterns by references

# Compressing with Frequent Patterns

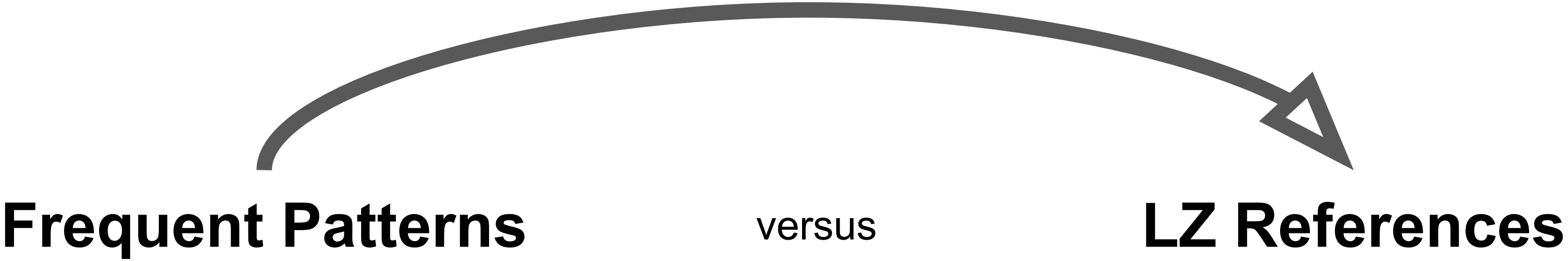
**Idea:**

- Estimate online which $k$ patterns are the most frequent
- Replace occurrences of frequent patterns by references

→ The number of candidate patterns is quadratic in $|S|$

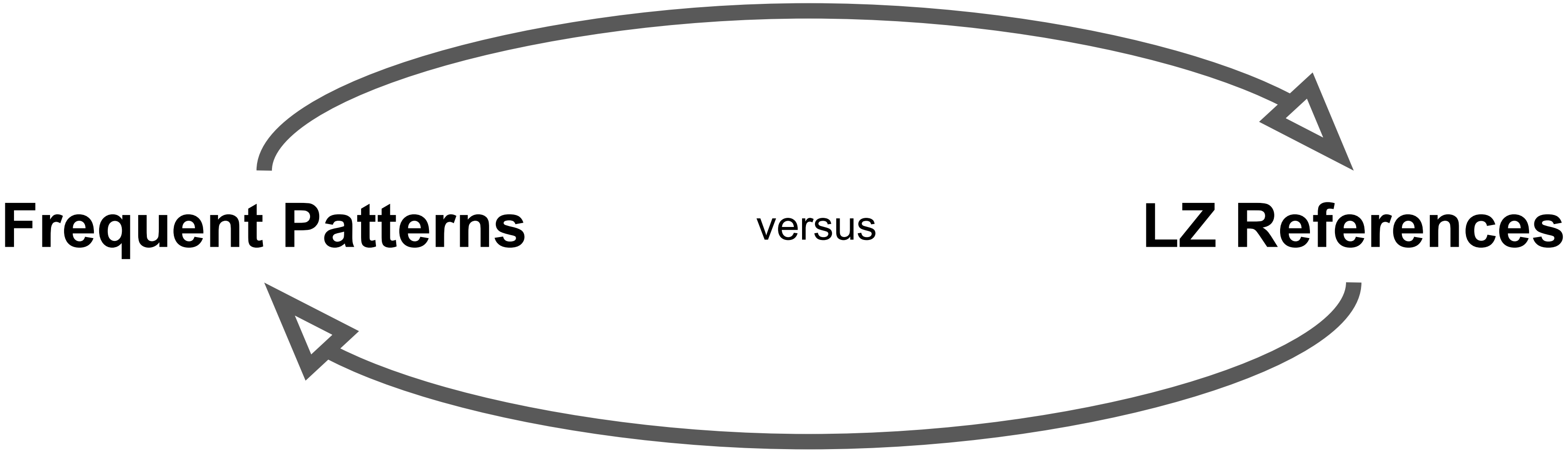
**Frequent Patterns** versus **LZ References**

→ Frequent patterns are captured by LZ references



**Frequent Patterns** versus **LZ References**

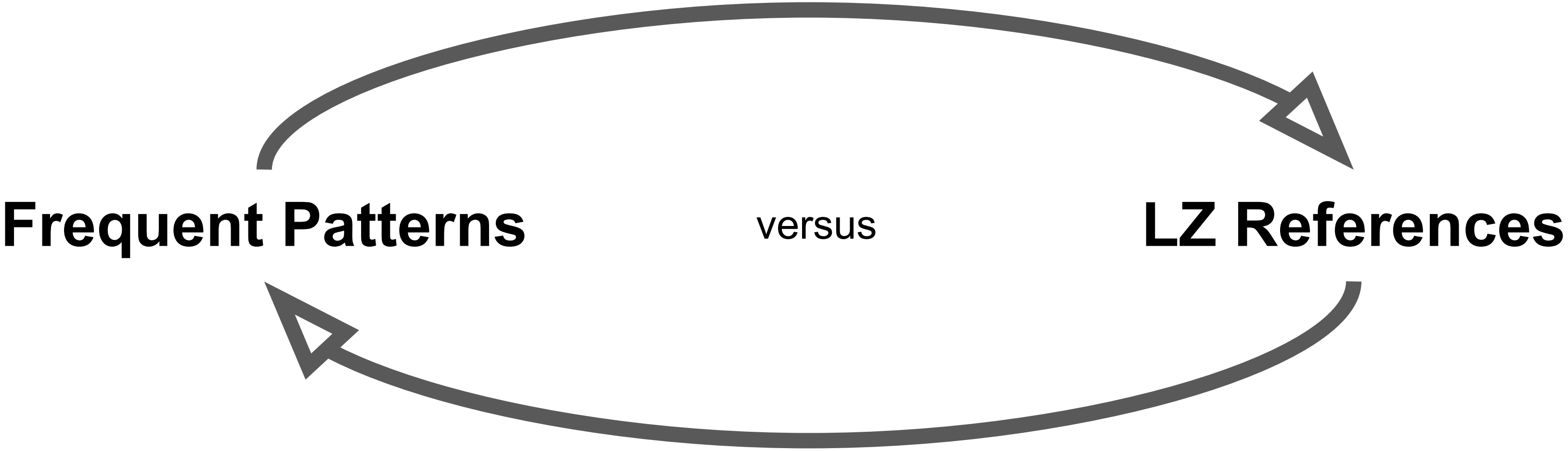
→ Frequent patterns are captured by LZ references



**Frequent Patterns** versus **LZ References**

→ Patterns *not* captured by LZ references *cannot* be frequent (by any measure)

→ Frequent patterns are captured by LZ references

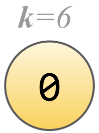


**Frequent Patterns** versus **LZ References**

→ Patterns *not* captured by LZ references *cannot* be frequent (by any measure)

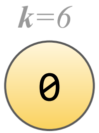
→ The number of LZ references is bounded by $|\boldsymbol{S}|$

# top-k LZ78



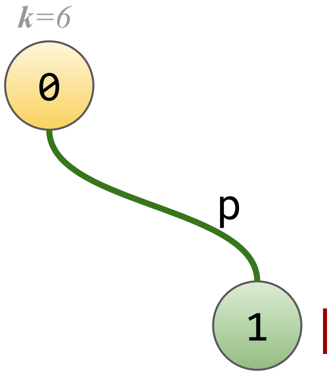
panamanananasbananapancake

# top-k LZ78



panamanananasbananapancake

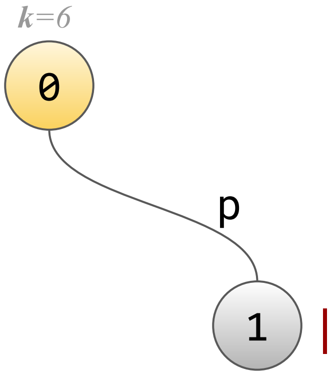
# top-k LZ78



panamananananasbananapancake

(0,p)

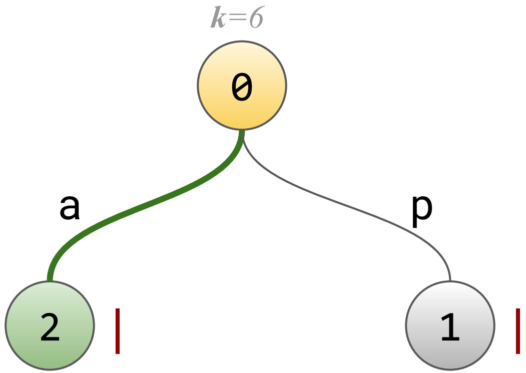
# top-k LZ78



p<u>a</u>namananananasbananapancake

(0,p)

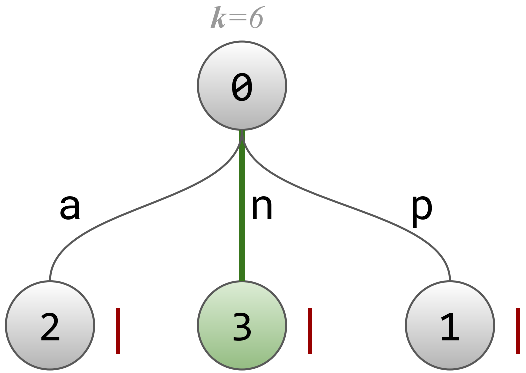
# top-k LZ78



panamanananasbananapancake

(0,p)(0,a)

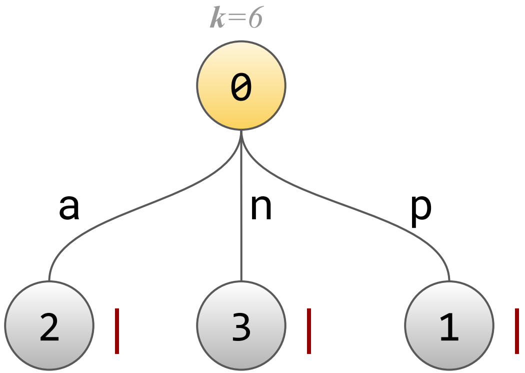
# top-k LZ78



pan<u>n</u>amanananasbananapancake

(0,p)(0,a)(0,n)

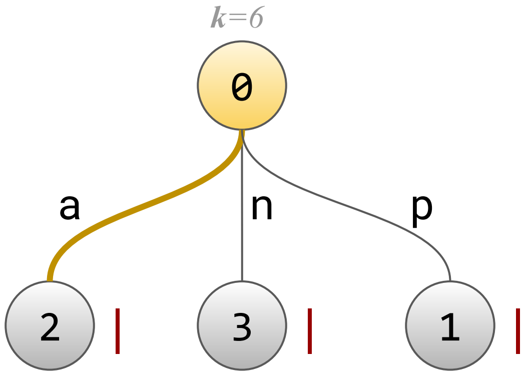
# top-k LZ78



pana<u>a</u>mananananasbananapancake

(0,p)(0,a)(0,n)

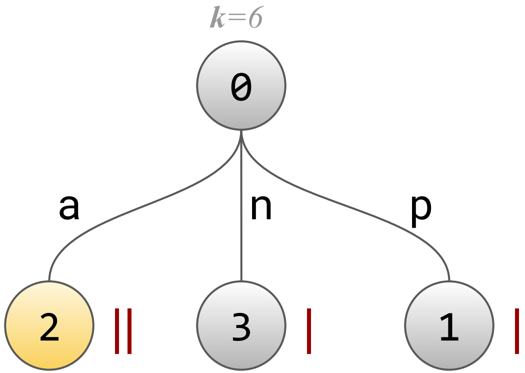
# top-k LZ78



pan<u>a</u>mananananasbananapancake

(0,p)(0,a)(0,n)

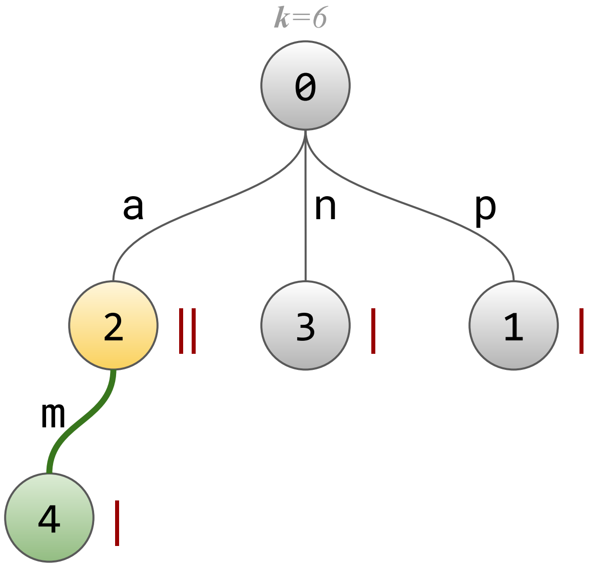
# top-k LZ78



panam̲anananasbananapancake

(0,p)(0,a)(0,n)

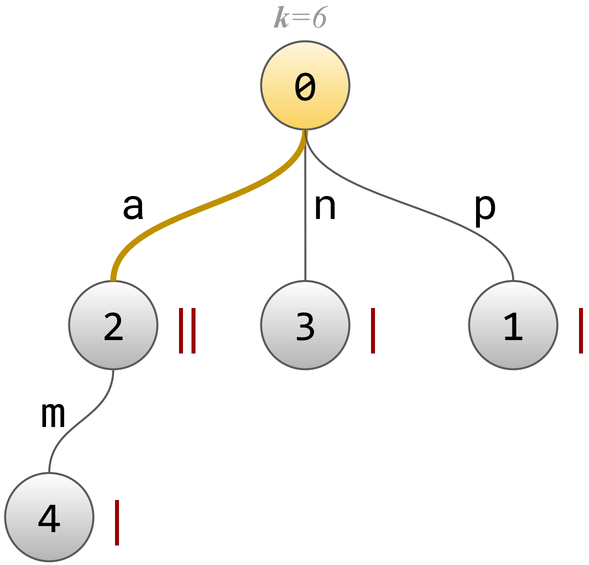
# top-k LZ78



panamananananasbananapancake

(0,p)(0,a)(0,n)(2,m)

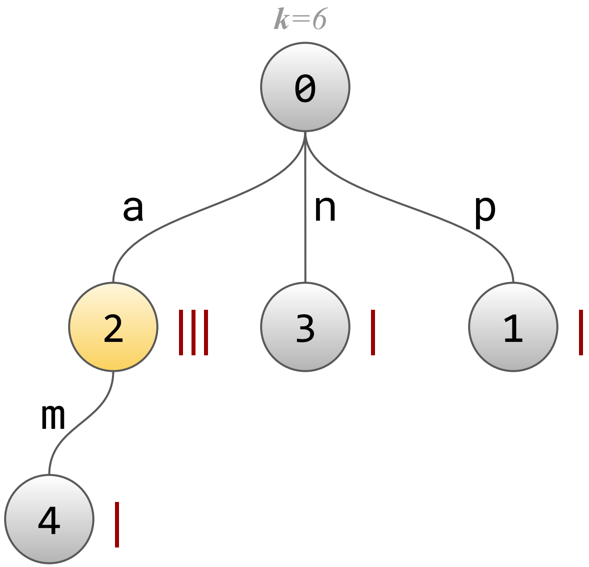
# top-k LZ78



panama<u>a</u>nananasbananapancake

(0,p)(0,a)(0,n)(2,m)

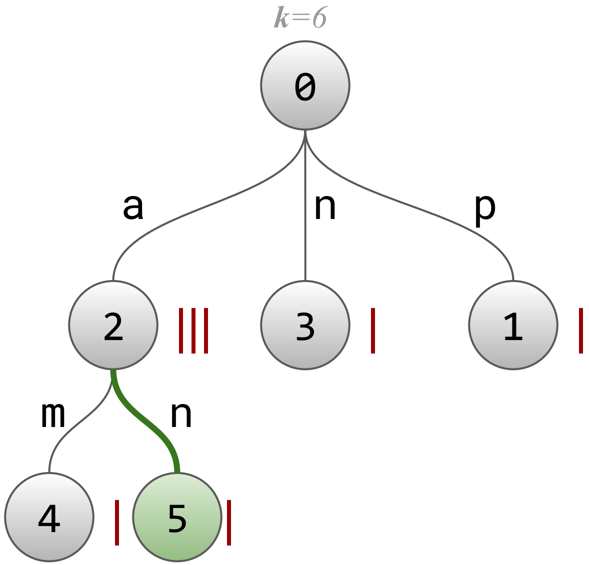
# top-k LZ78



panama<u>n</u>ananasbananapancake

(0,p)(0,a)(0,n)(2,m)

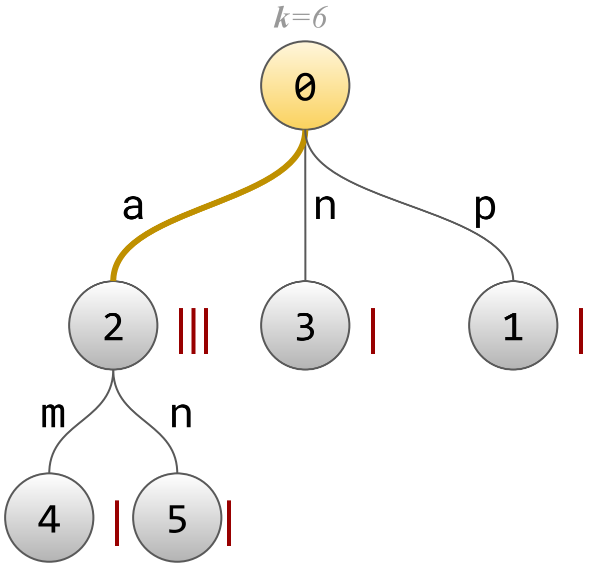
# top-k LZ78



panama<u>n</u>ananasbananapancake

(0,p)(0,a)(0,n)(2,m)<u>(2,n)</u>

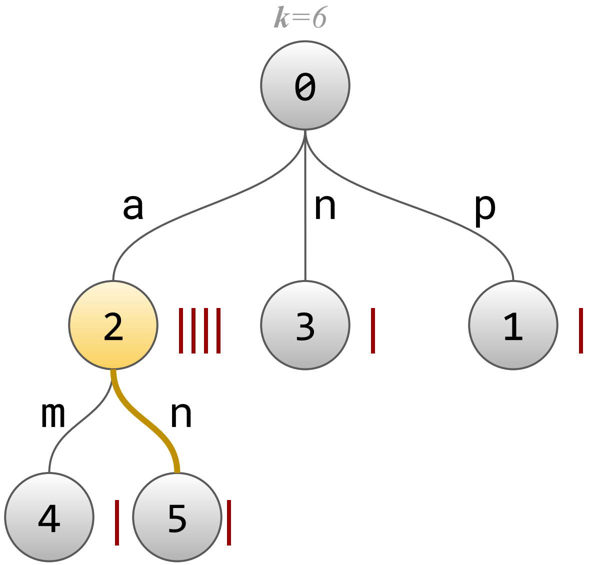
# top-k LZ78



panamana<u>a</u>nanasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)

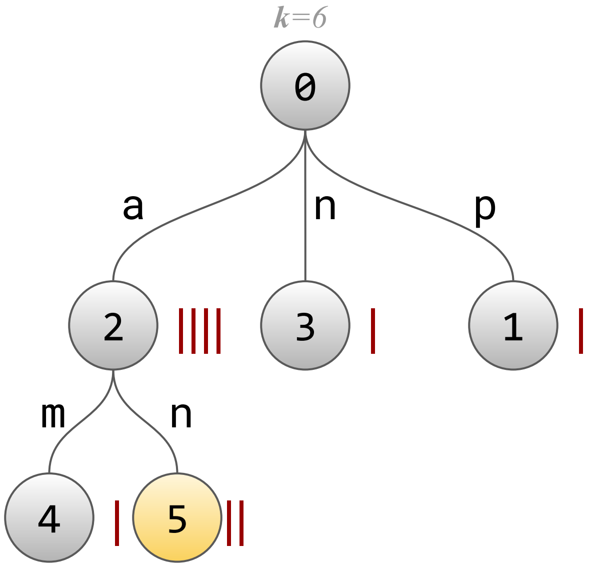
# top-k LZ78



panamana<u>n</u>anasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)

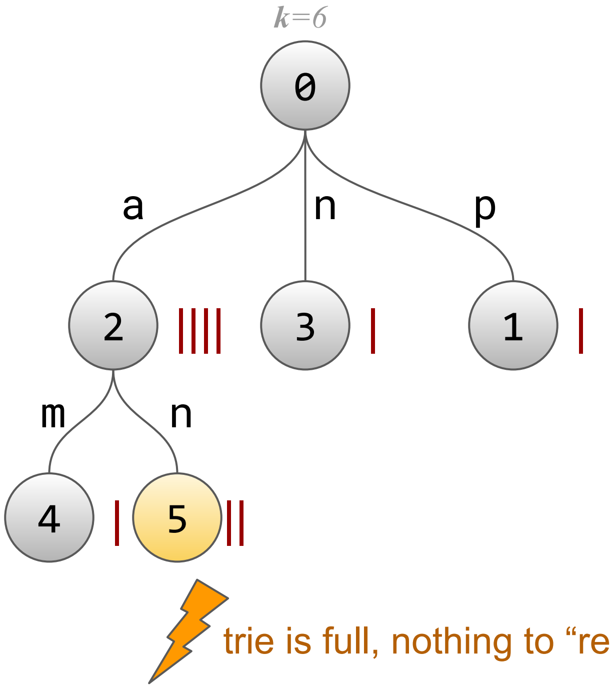
# top-k LZ78



panamanana<u>a</u>nasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)

# top-k LZ78

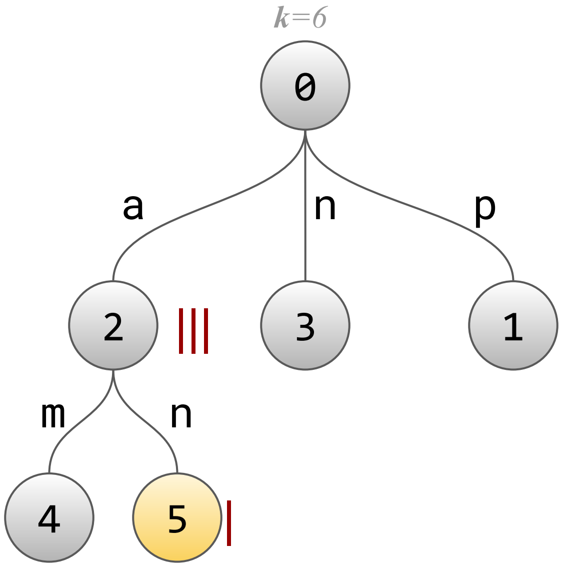


trie is full, nothing to "recycle"

panamanana<u>a</u>nasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)<u>(5,a)</u>

# top-k LZ78

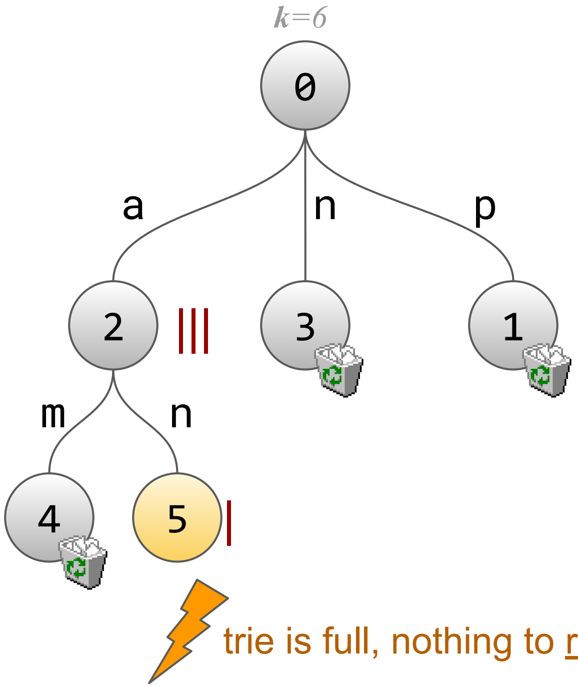


panamananasbananapancake

trie is full, nothing to “recycle” → decrement **all** counters

`(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)`

# top-k LZ78

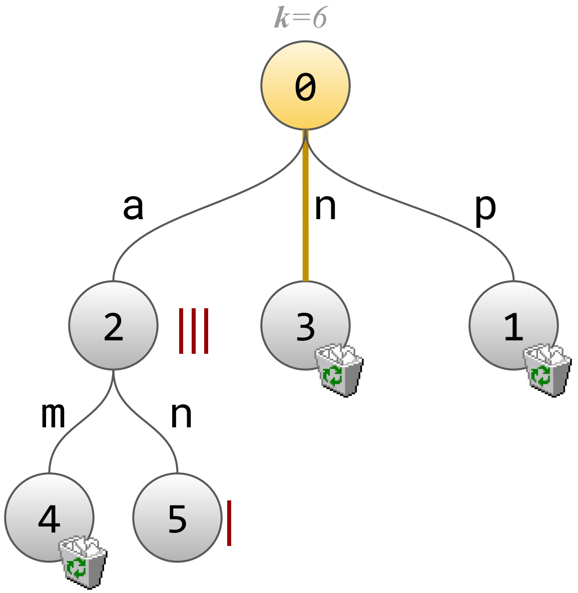


trie is full, nothing to recycle → decrement **all** counters

panamanananasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)

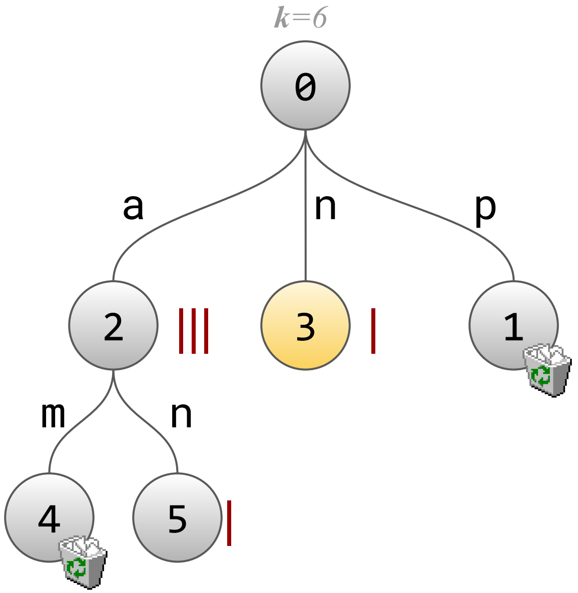
# top-k LZ78



panamananan̲asbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)

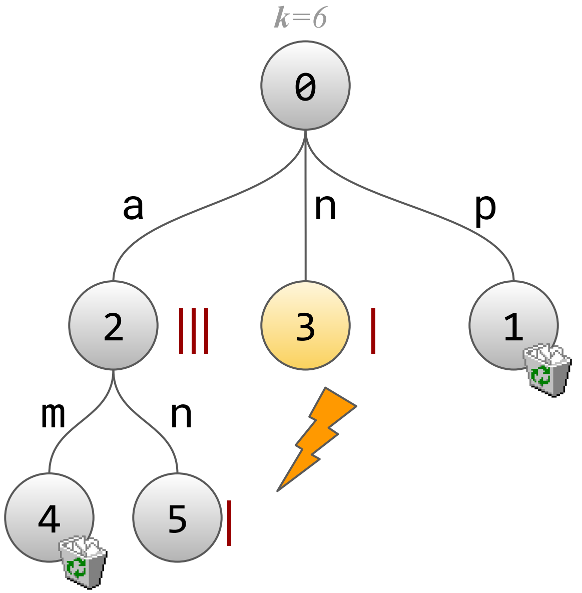
# top-k LZ78



panamananan<u>a</u>sbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)

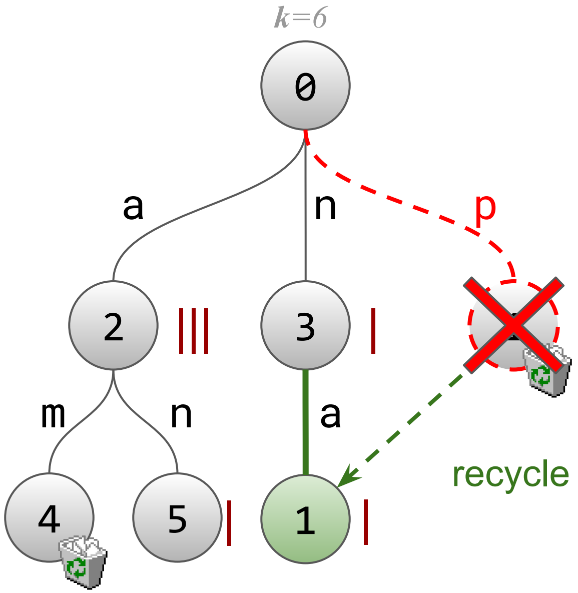
# top-k LZ78



panamananan<ins>a</ins>sbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)<ins>(3,a)</ins>

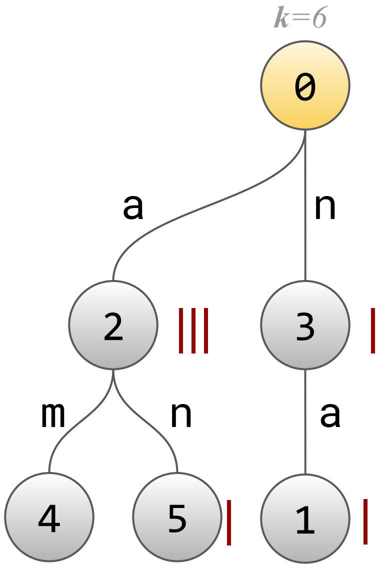
# top-k LZ78



panamananana sbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)(3,a)

# top-k LZ78

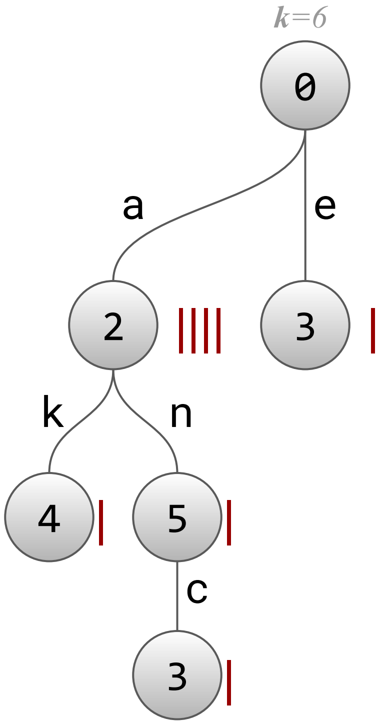


panamananana<u>s</u>bananapancake

**… and so forth …**

`(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)(3,a)`

# top-k LZ78



panamananananasbananapancake

(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)(3,a)(0,s)(0,b)(5,a)(1,p)(5,c)(2,k)(0,e)

# top-k LZ78

`(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)(3,a)(0,s)(0,b)(5,a)(1,p)(5,c)(2,k)(0,e)`

→ **time-dependent** top-k trie references

# top-k LZ78

`(0,p)(0,a)(0,n)(2,m)(2,n)(5,a)(3,a)(0,s)(0,b)(5,a)(1,p)(5,c)(2,k)(0,e)`

→ **time-dependent** top-k trie references

→ hybrid of **LZ78** compression and **Misra-Gries** sketch

→ simulate decrements & manage garbage queue using [Metwally et al., 2005]

→ constant amortized time per input character

# LZ78 vs. LZ77

Input: $\mathrm{a}^{n}$ = aaaaaaaaaaaaaaaaa …

LZ78:

LZ77:

# LZ78 vs. LZ77

Input: a$^n$ = aaaaaaaaaaaaaaaaa …

LZ78: (0,a)(1,a)(2,a)(3,a) … → $\Theta(\sqrt{n})$ phrases

LZ77:

# LZ78 vs. LZ77

Input: $a^n$ = aaaaaaaaaaaaaaaa …

LZ78: (0,a)(1,a)(2,a)(3,a) … → $\Theta(\sqrt{n})$ phrases

LZ77: a(1,$n$-$1$) → 2 phrases

# LZ78 vs. LZ77

Input: $a^n$ = aaaaaaaaaaaaaaaa …

LZ78: (0,a)(1,a)(2,a)(3,a) … → $\Theta(\sqrt{n})$ phrases

⊕ simple algorithm (compressed space by default)

⊕ straightforward efficient encoding

LZ77: a(1,$n$-$1$) → 2 phrases

⊖ much harder to compute

⊖ harder to encode efficiently

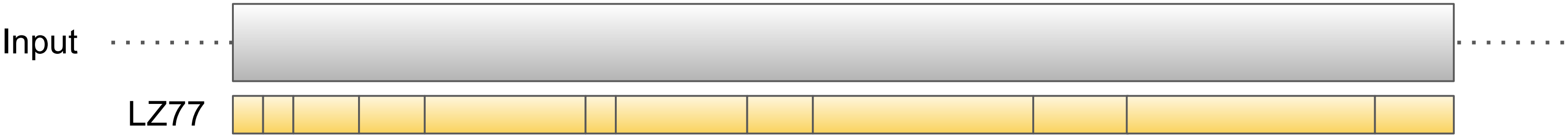
→ best of both worlds?

# top-k LZ77

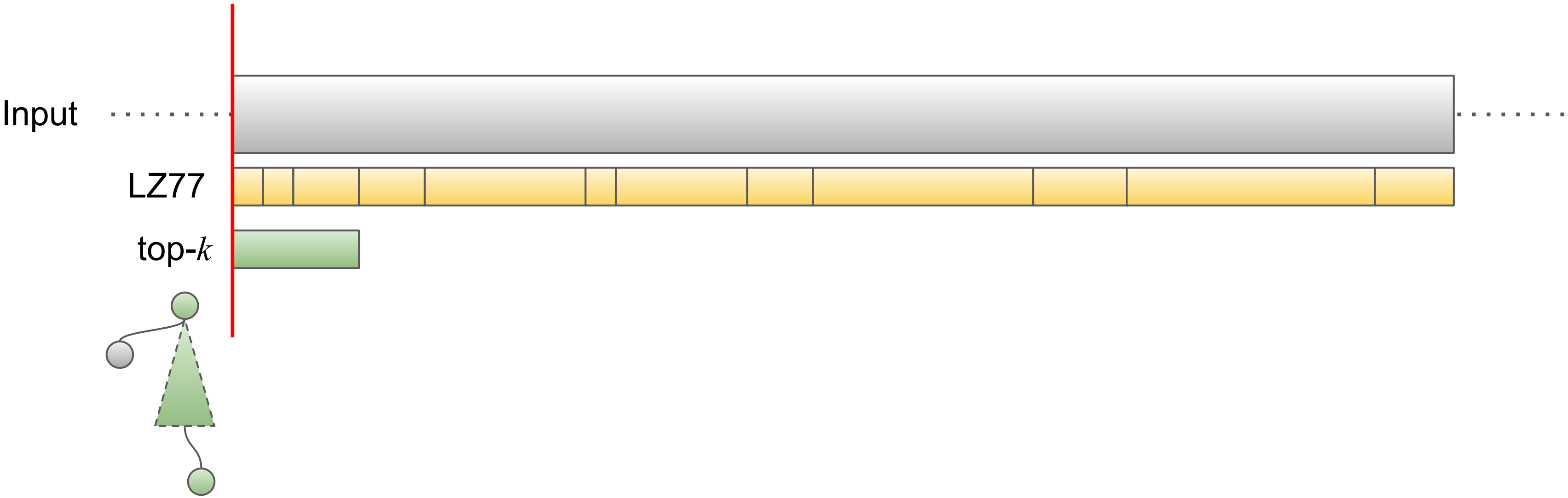
**Algorithm Sketch:**

1. Maintain a top-k trie of phrases (like in top-k LZ78)
2. Partition the input into blocks of size $\boldsymbol{B} = O(\boldsymbol{k})$
3. Compute the block's LZ77 parsing in time and space $O(\boldsymbol{B}) = O(\boldsymbol{k})$
4. Greedily pick the next LZ77 phrase or matching pattern from the trie

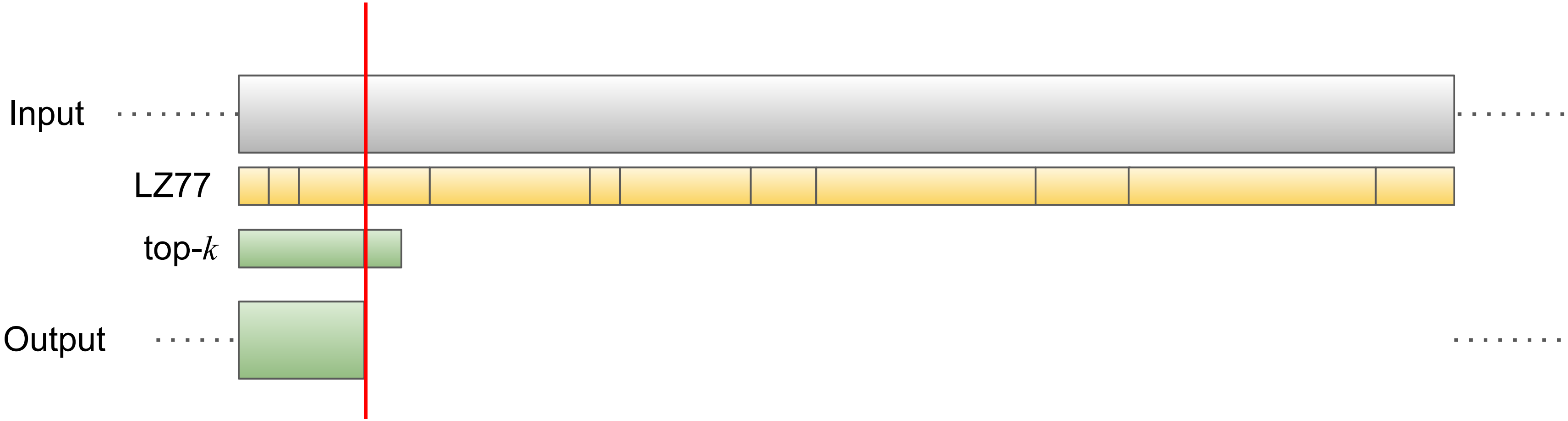
# top-k LZ77



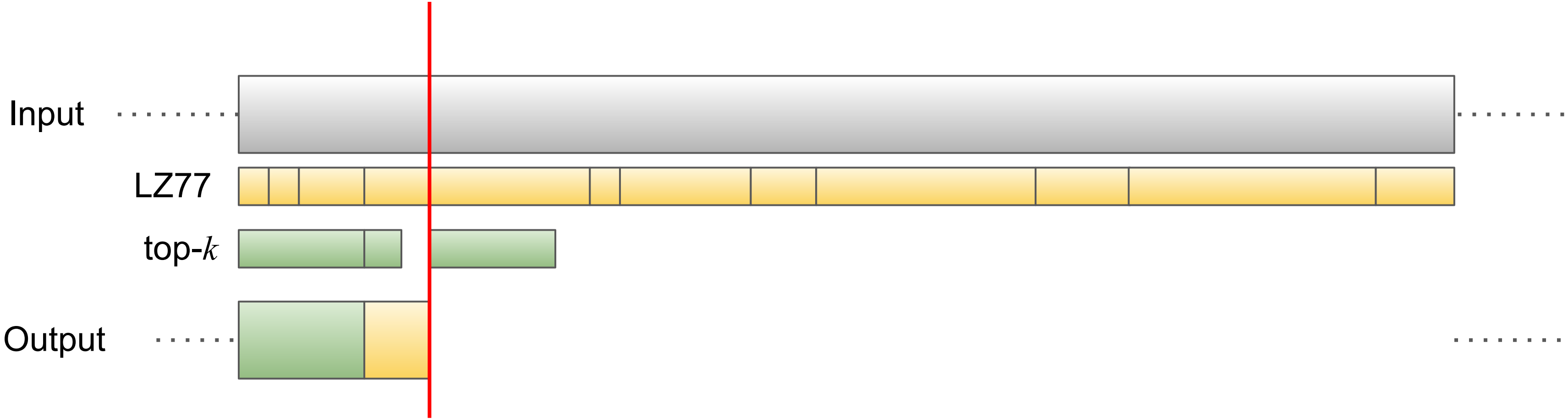
# top-k LZ77



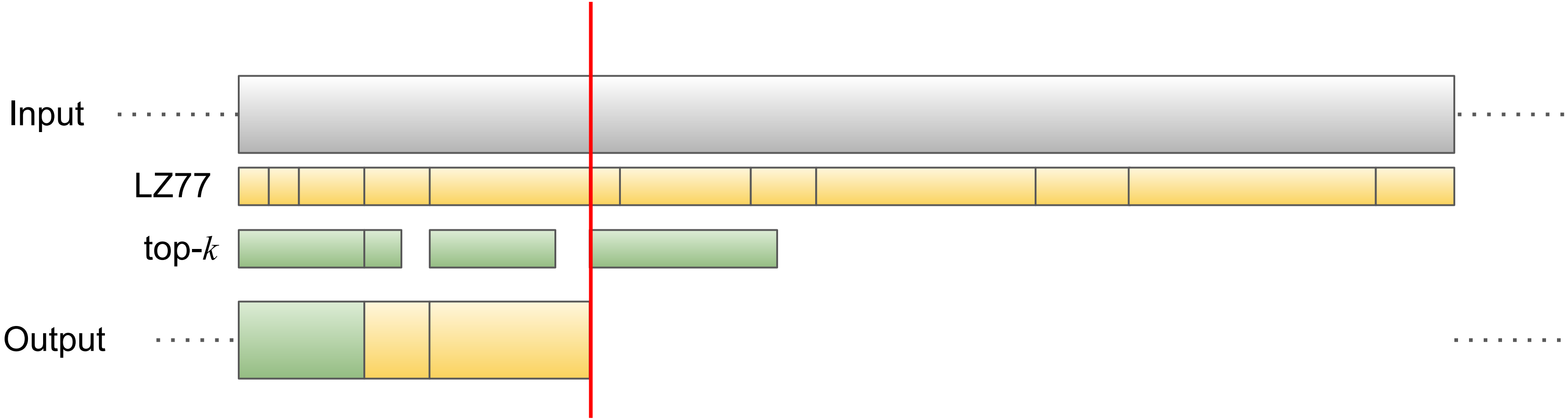
# top-k LZ77



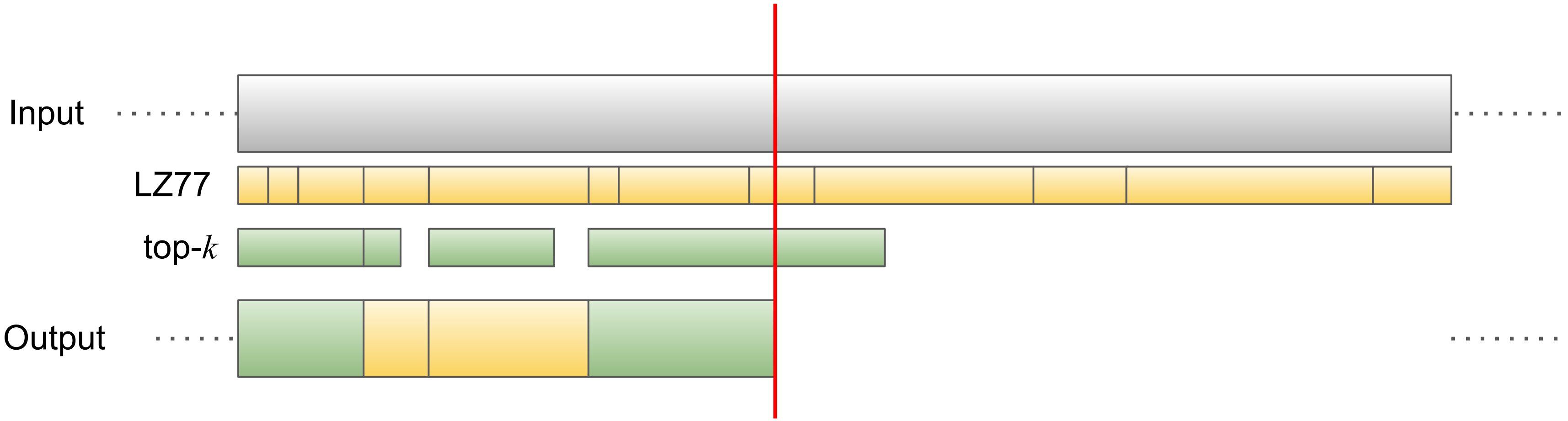
# top-k LZ77



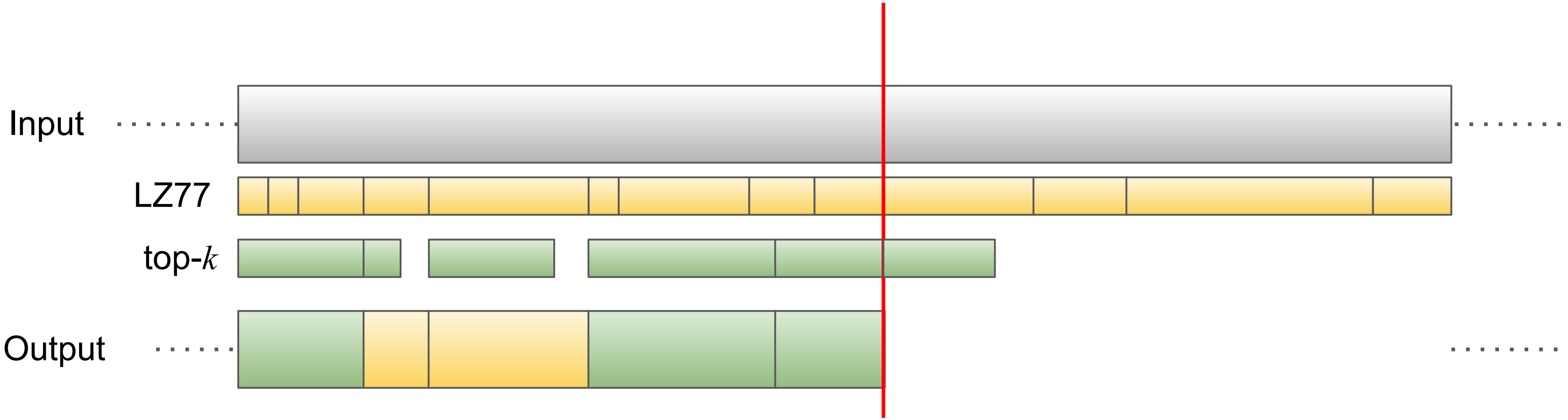
# top-k LZ77



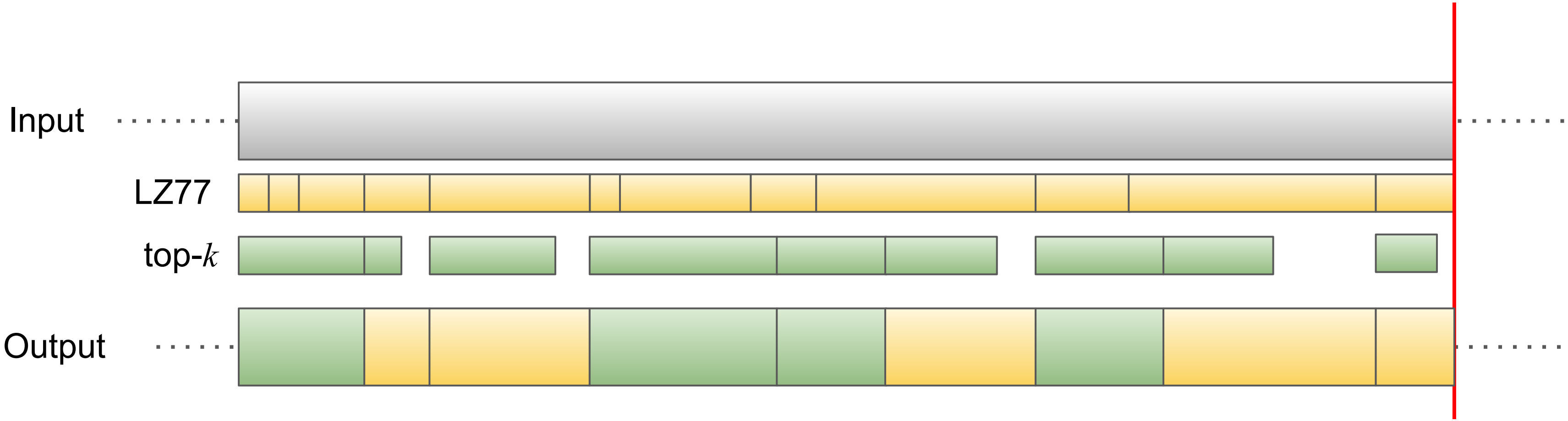
# top-k LZ77



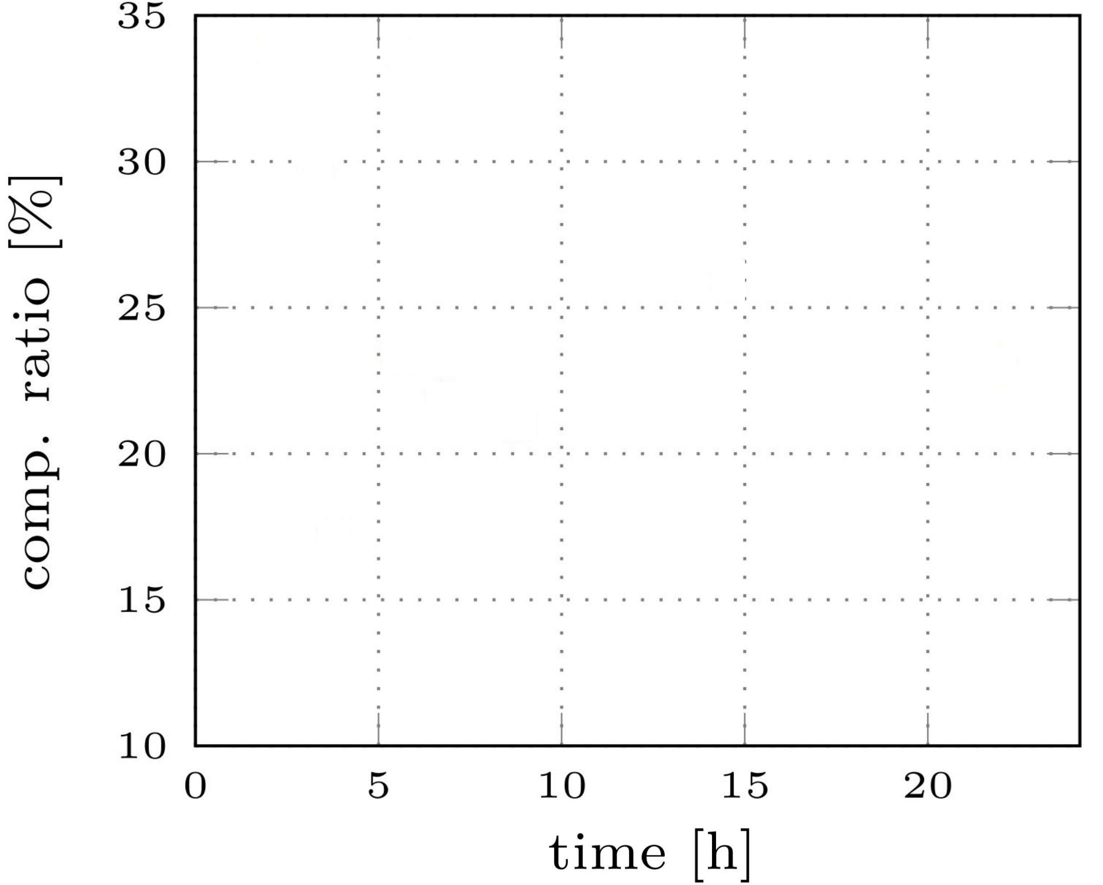
# top-k LZ77



# top-k LZ77

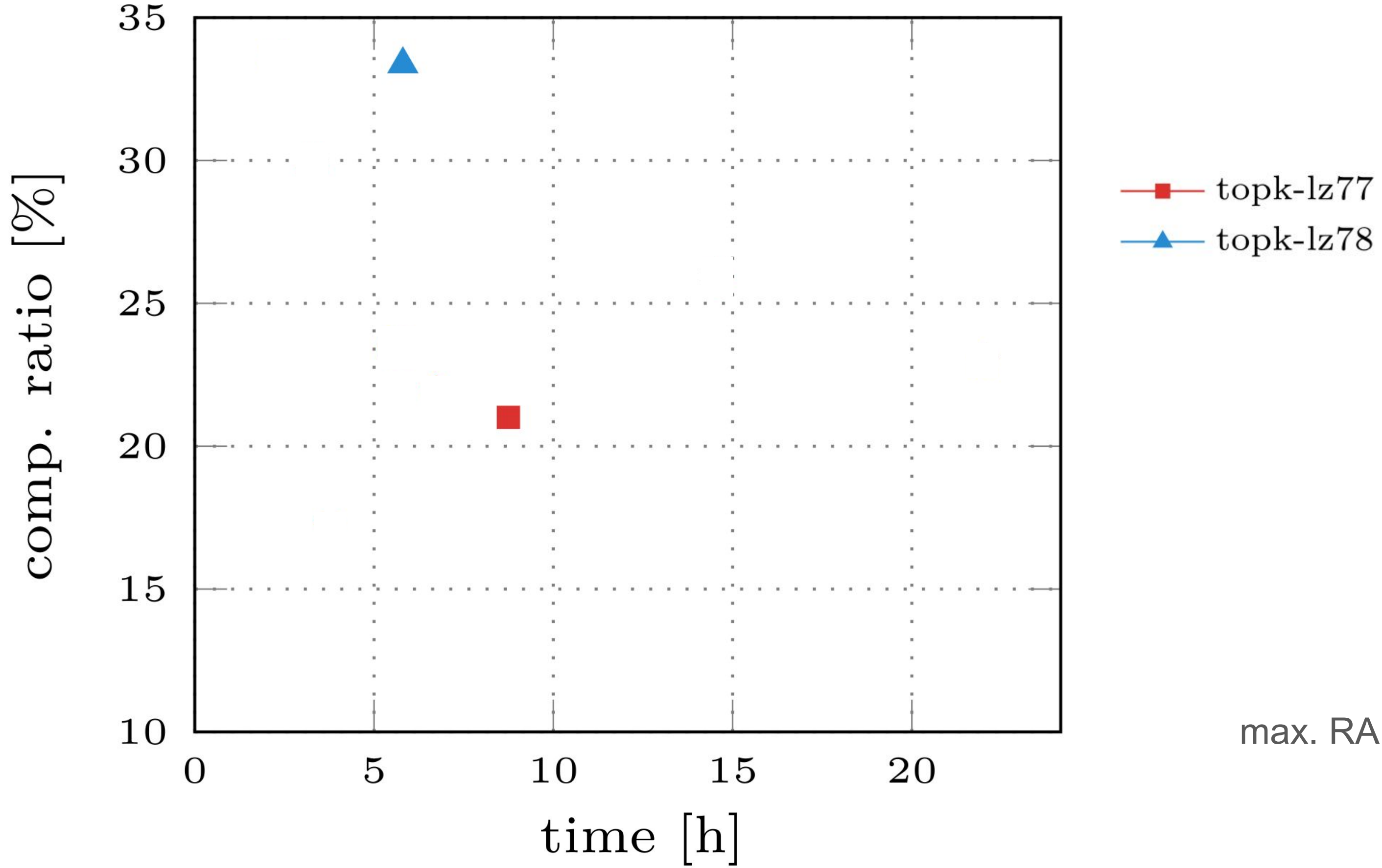


## CommonCrawl (100 GiB web text crawl)



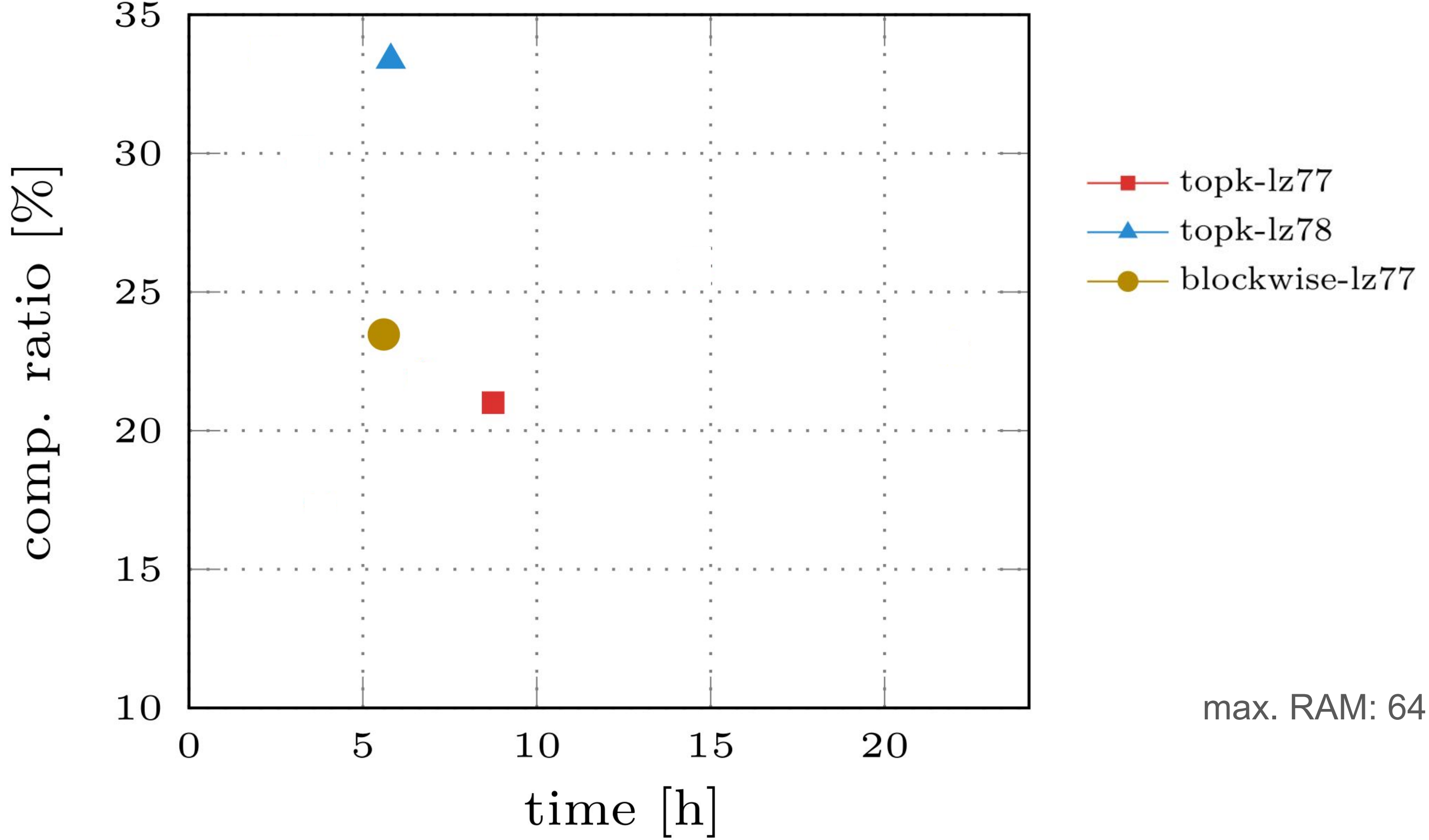
max. RAM: 64 GB

# CommonCrawl (100 GiB web text crawl)



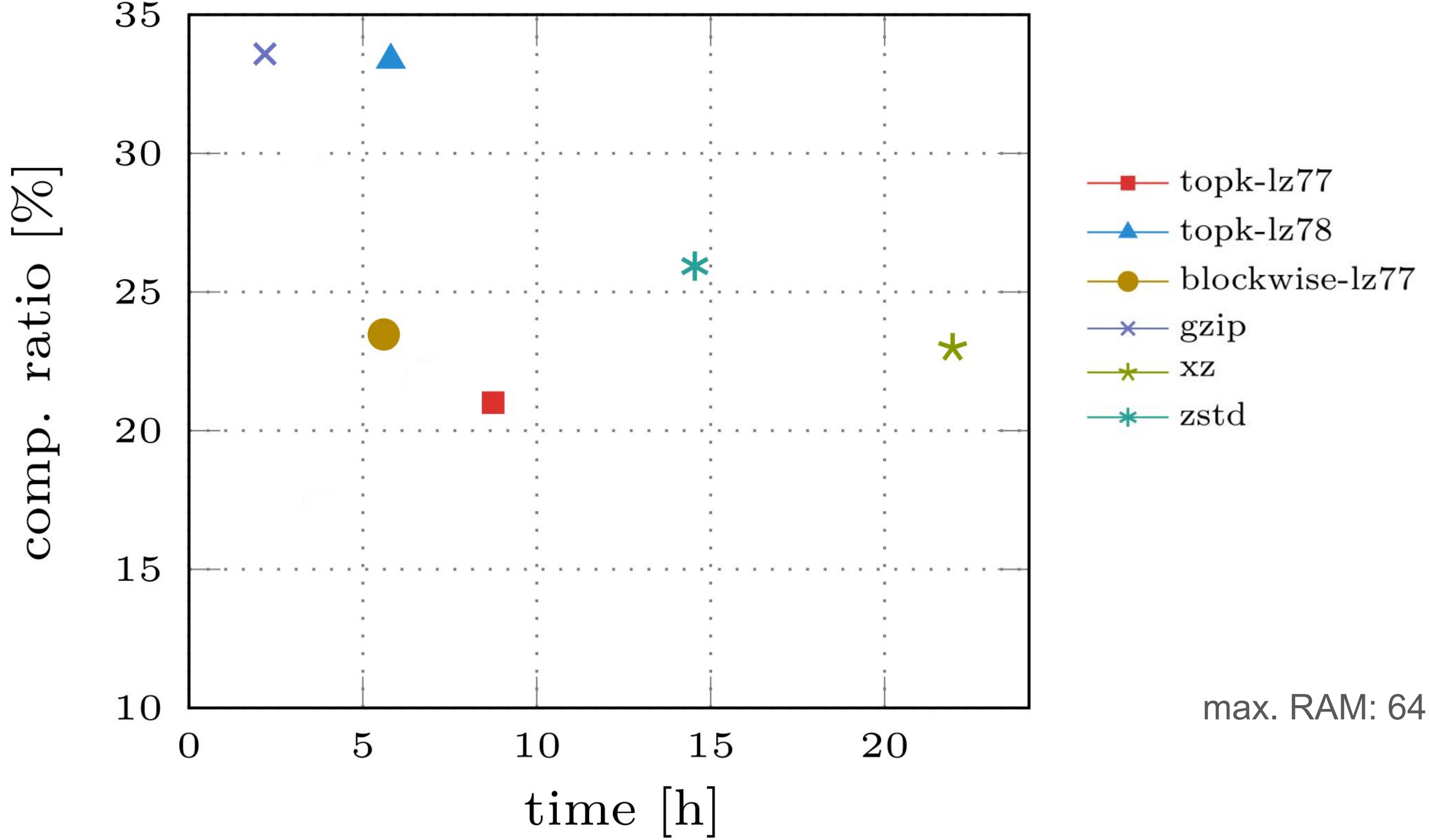
max. RAM: 64 GB

## CommonCrawl (100 GiB web text crawl)



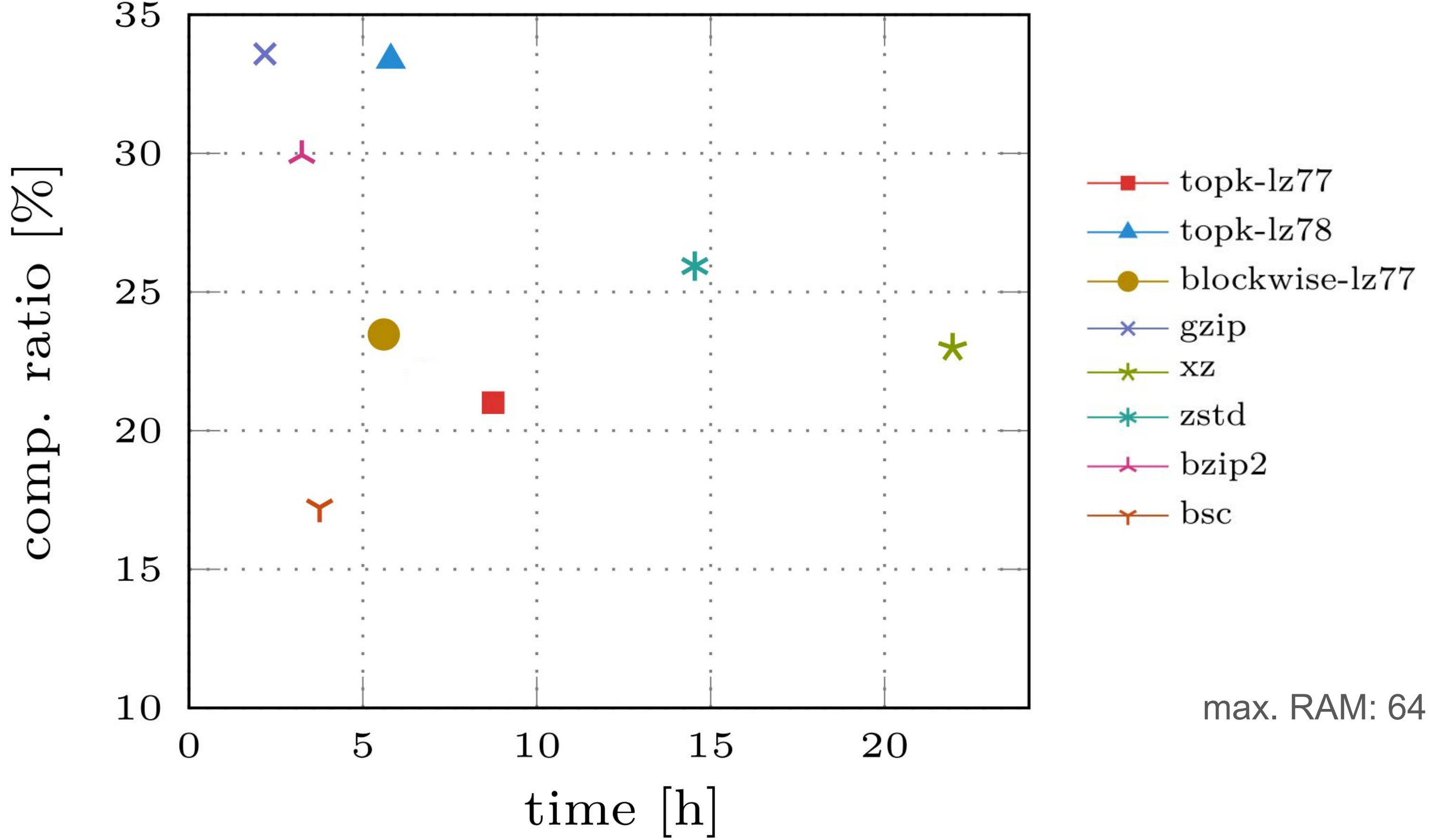
max. RAM: 64 GB

# CommonCrawl (100 GiB web text crawl)



max. RAM: 64 GB

# CommonCrawl (100 GiB web text crawl)



max. RAM: 64 GB

# Future Work

- Parallel computation
- Statistical encoding of trie references
- Dynamic string attractors? (as opp. to tries)
- **Precompression** of long repetitions
- Random Access (e.g., like [Arz & Fischer, 2018])
- Information Retrieval
- Your idea here!