Distributed Storage Systems

Compression & Delta Encoding

Agenda

Today’s topics

- Basics of compression
- Compression schemes
- Delta encoding

Class Structure

	Lecture	Lab
Week 1	Course introduction, networking basics, socket programming	Python sockets
Week 2	RPC, NFS, Practical RPC	Flask, JsonRPC, REST API
Week 3	AFS, reliable storage introduction	ZeroMQ, ProtoBuf
Week 4	Hard drives, RAID levels	RPi stack intro, RPi RAID with ZMQ
Week 5	Finite fields, Reed-Solomon Codes	Kodo intro, RS and RLNC with Kodo
Week 6	Repair problem, RS vs Regenerating codes	RPi simple distributed storage with Kodo RS
Week 7	Regenerating codes, XORBAS	RPi Regenerate lost fragments with RS
Week 8	Hadoop	RPi RLNC, recovery with recode
Week 9	Storage Virtualization, Network Attached Storage, Storage Area Networks	RPi basic HDFS (namenode+datanode, read & write pipeline)
Week 10	Object Storage	RPi basic S3 API
Week 11	**Compression, Delta Encoding**	**Mini project consultation**
Week 12	Data Deduplication	RPi Dedup
Week 13	Fog storage	Mini project consultation
Week 14	Security for Storage Systems and Recap	Mini project consultation

Basics of Compression

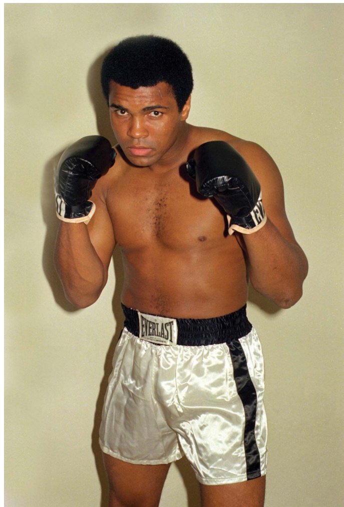
Key question: what is information?

- Example: you fight Muhammad Ali
    - He throws a jab with probability 0.5
    - He throws a cross with probability 0.4
    - He throws an uppercut and hook with probability 0.05 each

What carries more information if you need to block/evade?

How many bits to identify each case efficiently?

AP images



Key question: what is information?

- Any quantification of “information” about an event depends on its probability
  - The greater the probability of an event, the smaller the information associated with knowing that the event occurred
  - Semantics are not important for this definition (but can be for actual algorithms)

Key question: what is information?

Information of an event given that it has a probability p of happening:

$$I = \log_2(1/p) = -\log_2(p)$$

- Base 2 because we use bits :)
- Provides nice properties:
  - Information of independent events A and B is added, but probability should be multiplied
  - $I_A + I_B = -\log_2(p_A) - \log_2(p_B) = -\log_2(p_A p_B)$

(Shannon) Entropy

Expected information in a set of possible outcomes or mutually exclusive events:

$$E[I] = H(p_1, p_2, \ldots p_N) = \sum p_i \log_2(1/p_i)$$

Another way to think of it: expected uncertainty associated with this set of events

- H is non-negative
- $H \leq \log_2(N)$
  (equality when all events are equally likely)

Source Encoding/Compression

Types

Lossless Compression compressed can be reconstituted (uncompressed) without loss of detail or information

Example: Files, Logs, ...

Lossy Compression aim is to obtain the best possible fidelity for a given bit-rate or minimizing the bit-rate to achieve a given fidelity measure

Example: Video, audio

Our Focus: Lossless Compression

When a message is drawn from a set of possible messages, each occurring with some probability

- Entropy:
    - Expected amount of information in a message
    - Lower bound on the amount of information that must be sent

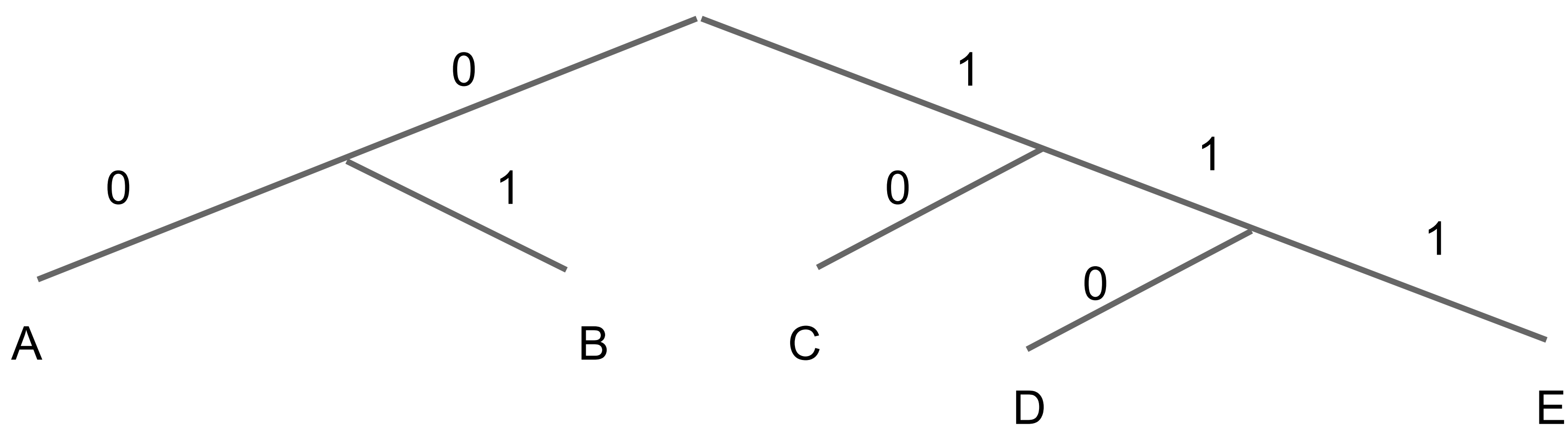
The Shannon-Fano Algorithm

This is a basic information theoretic algorithm

Symbol	A	B	C	D	E
Count	15	7	6	6	5

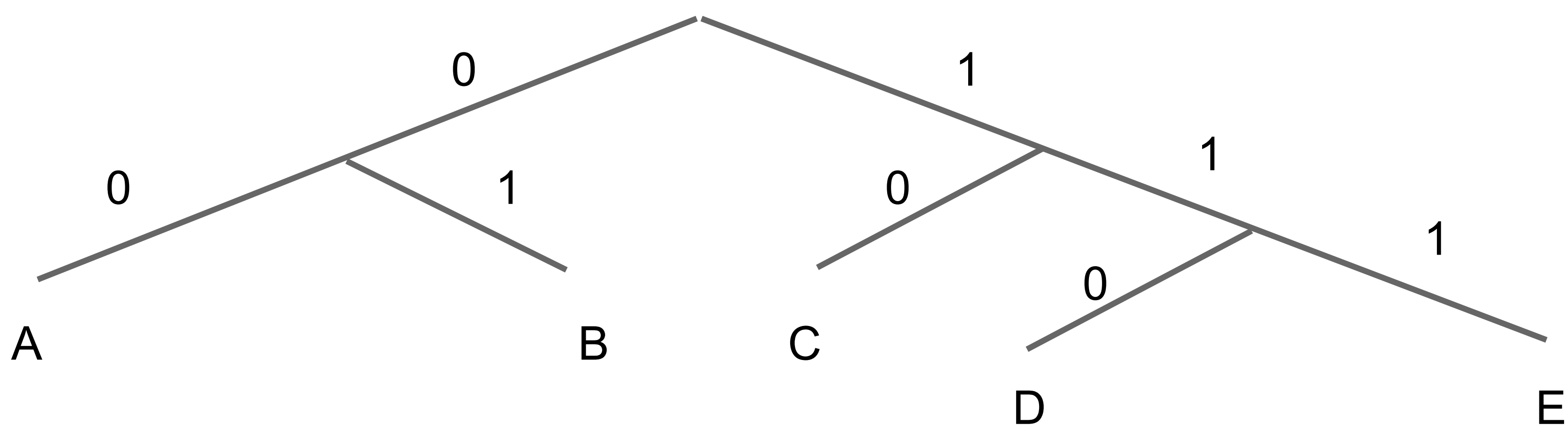
Encoding for the Shannon-Fano Algorithm:

1. Sort symbols according to their frequencies/probabilities: ABCDE
2. Recursively divide into two parts, each with approx. same number of counts



Encoding for the Shannon-Fano Algorithm:

1. Sort symbols according to their frequencies/probabilities: ABCDE
2. Recursively divide into two parts, each with approx. same number of counts



Bits in average: 2 x (15 + 7 + 6)/(39) + 3 x (6 + 5)/39 ~ 2,28 bits/symbols

Huffman Coding

Based on the probability (or frequency of occurrence) of a data item

The principle is to use a lower number of bits to encode the data that occurs more frequently

Codes are stored in a **Code Book**

The code book + encoded data must be transmitted to enable decoding

Fixed # of bits to variable # of bits

Huffman Coding

- **Unique Prefix Property**:
  - No code is a prefix to any other code
  - Great for decoder, unambiguous
- If prior statistics are available and accurate, then Huffman coding is very good

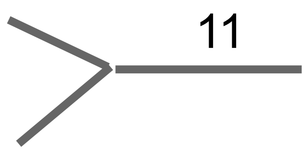
Huffman Coding

1. **Init:** Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE)

2. Repeat until the OPEN list has only one node left:
   a. From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them
   b. Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN
   c. Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN

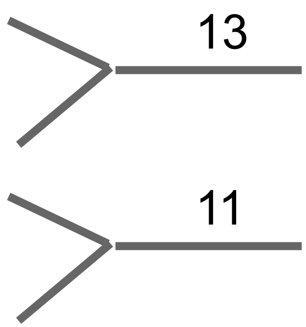
Huffman Coding

Count	Symbol
15	A
7	B
6	C
6	D
5	E



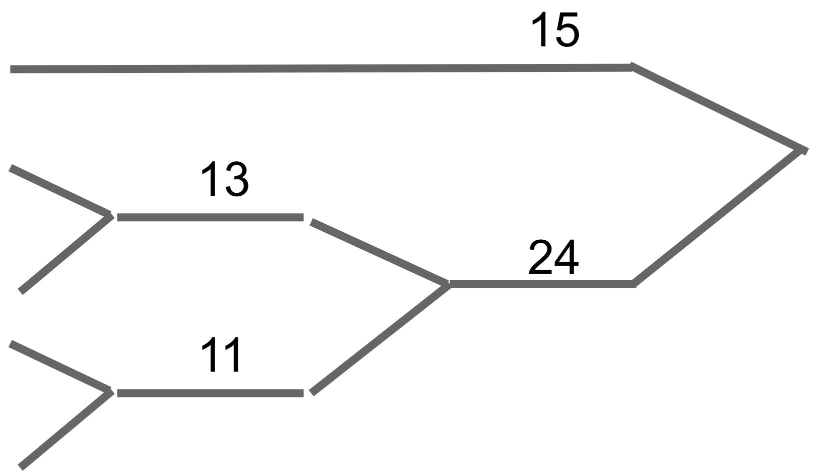
Huffman Coding

Count	**Symbol**
15	A
7	B
6	C
6	D
5	E



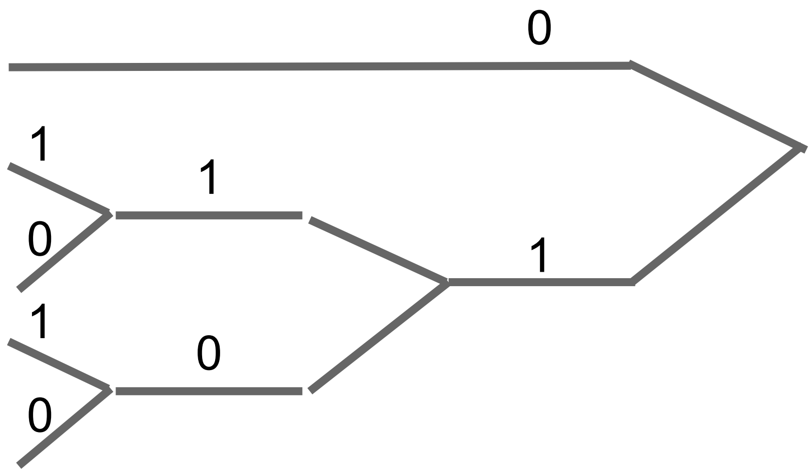
Huffman Coding

Count	Symbol
15	A
7	B
6	C
6	D
5	E



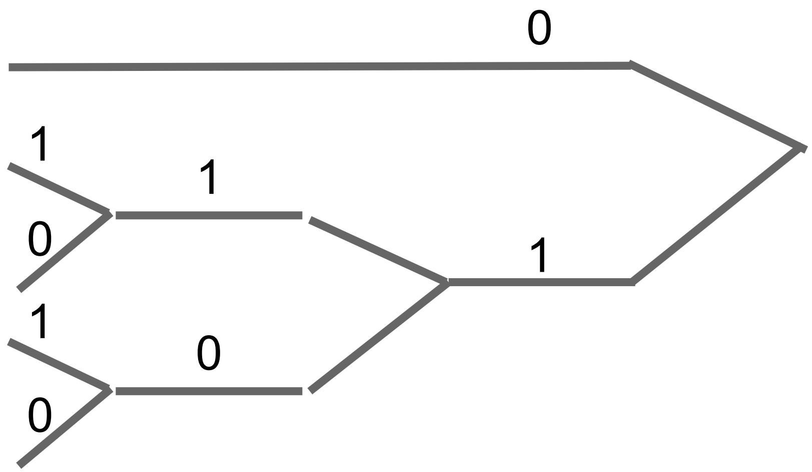
Huffman Coding

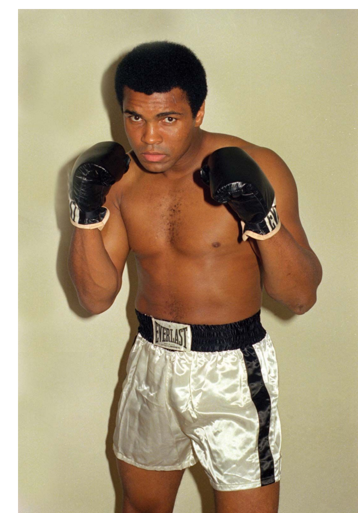
Count	Symbol	Code
15	A	0
7	B	111
6	C	110
6	D	101
5	E	100



Huffman Coding

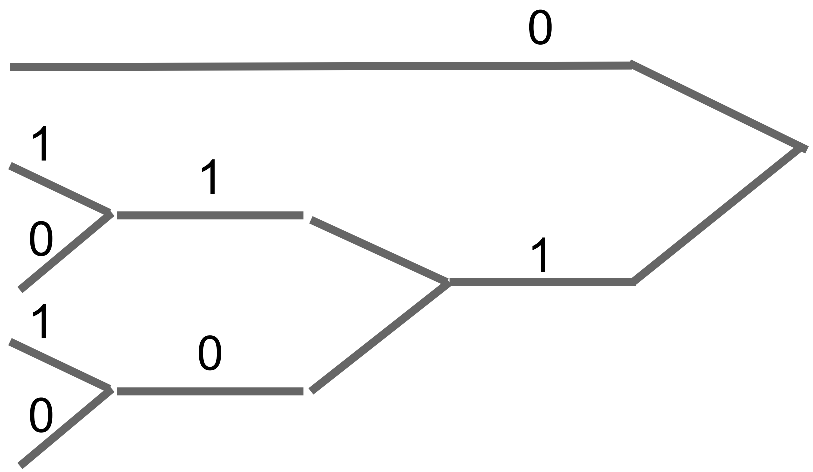
Count	Symbol	Code
15 (Jab)	**A**	**0**
7	B	111
6	C	110
6	D	101
5	E	





Huffman Coding

Count	Symbol	Code
15	A	0
7	B	111
6	C	110
6	D	101
5	E	100

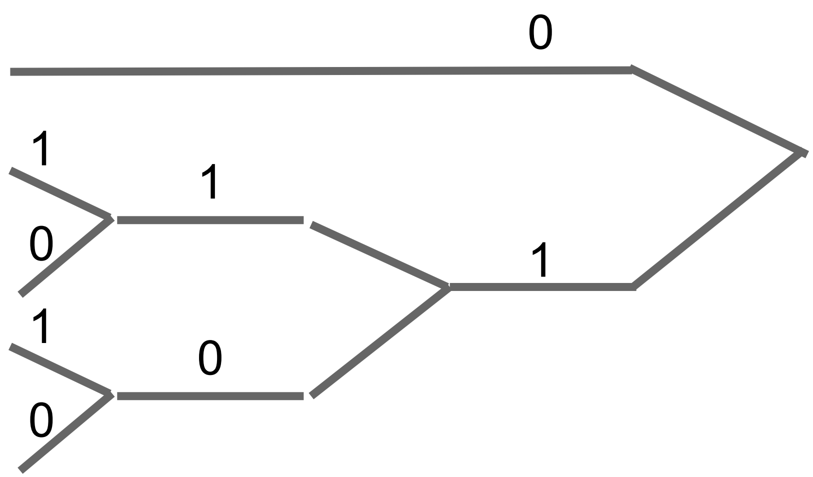


Bits in average: 1 x 15/(39) + 3 x (24)/39 ~ 2,2307 bits/symbol

How about entropy?

Huffman Coding

Count	**Symbol**	**Code**
15	A	0
7	B	111
6	C	110
6	D	101
5	E	100



Bits in average: 1 x 15/(39) + 3 x (24)/39 ~ 2,2307 bits/symbol

Entropy:

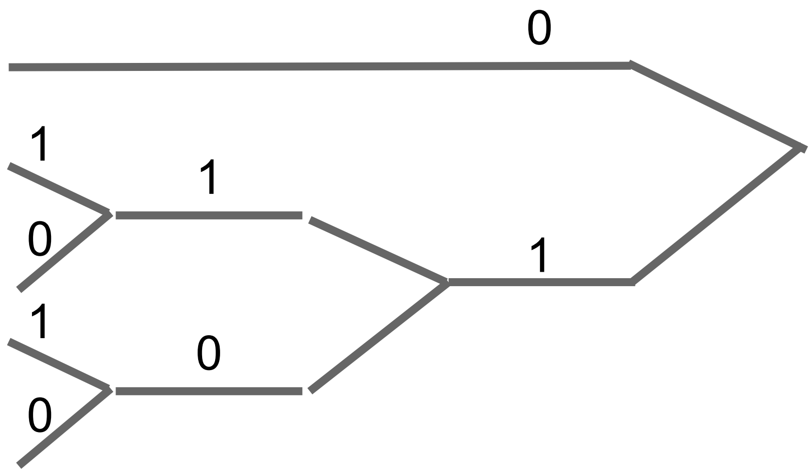
$$- 15/39 \times \log_2(15/39) - 7/39 \times \log_2(7/39) - 2 \times 6/39 \times \log_2(6/39) - 5/39 \times \log_2(5/39)$$

$$= 0,5301967 + 0,44477772 + 0,8309045 + 0,3799325$$

$$\mathbf{= 2,18581142}$$

Huffman Coding

Count	Symbol	Code
15	A	0
7	B	111
6	C	110
6	D	101
5	E	100



Bits in average: 1 x 15/(39) + 3 x (24)/39 ~ 2,2307 bits/symbol

Entropy:

- 15/39 x $\log_2$(15/39) - 7/39 x $\log_2$(7/39) - 2x6/39 x $\log_2$(6/39) - 5/39 x $\log_2$(5/39)

= 0,5301967 + 0,44477772 + 0,8309045 + 0,3799325

= 2,18581142

Shannon-Fano: ~ 2,28 bits/symbols

Adaptive Huffman Coding

The basic Huffman algorithm has been extended

- Statistical knowledge is often not available
- Even when it is available, it could be a heavy overhead especially when many tables have to be sent

The solution is to use adaptive algorithms, e.g. Adaptive Huffman coding (applicable to other adaptive compression algorithms).

Problem: how to determine probabilities?

- Simple idea is to use adaptive model:
  - Start with guess of symbol frequencies
  - Update frequency with each new symbol
- Another idea is to take account of intersymbol probabilities, e.g., prediction by partial matching

Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique

Example: Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries

Why not just transmit each word as an 18 bit number?

Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique

Example: Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries

Why not just transmit each word as an 18 bit number?

- Too many bits
- Everyone needs a dictionary
- For this simple example, it would work for English text only

Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique

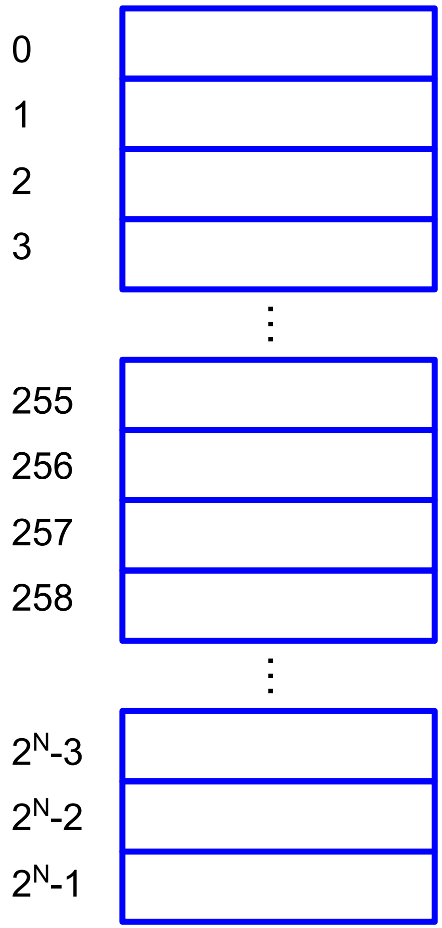
Example: Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries

Why not just transmit each word as an 18 bit number?

- **Solution**: Find a way to build the dictionary adaptively
  - Original methods due to Ziv and Lempel in 1977 and 1978. Terry Welch improved the scheme in 1984 (called LZW compression).
  - Variable length to fixed length (“reverse” of Huffmann)

LZW Compression

- Table size: 2^N
- N bits to represent each index
- Send **table indexes**
- String table can be reconstructed by the decoder using information from the encoded stream → table is not sent



LZW Compression Algorithm

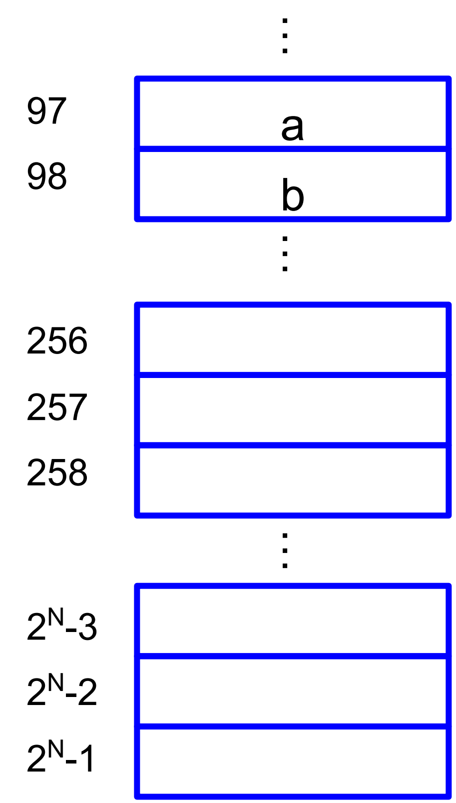
```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w                    (+ = concatenate)
```

LZW Compression Algorithm

__a__bbbabbbab

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = a



Output:

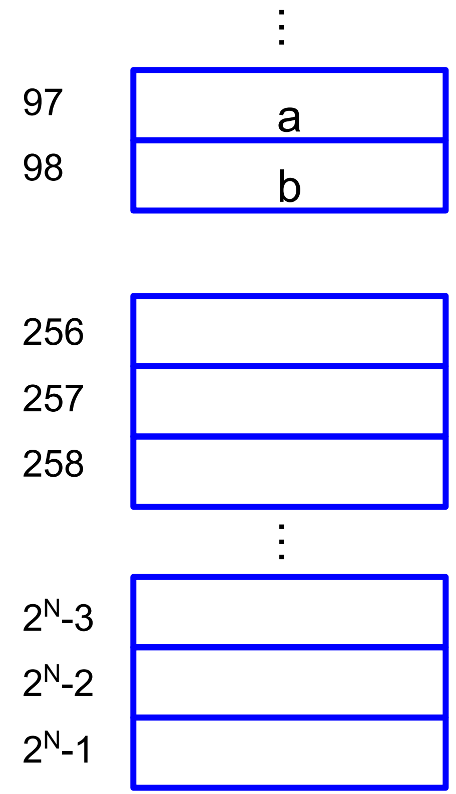
LZW Compression Algorithm

ab*bbabbbab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = a

k = b



Output:

LZW Compression Algorithm

ab*bbabbbab*

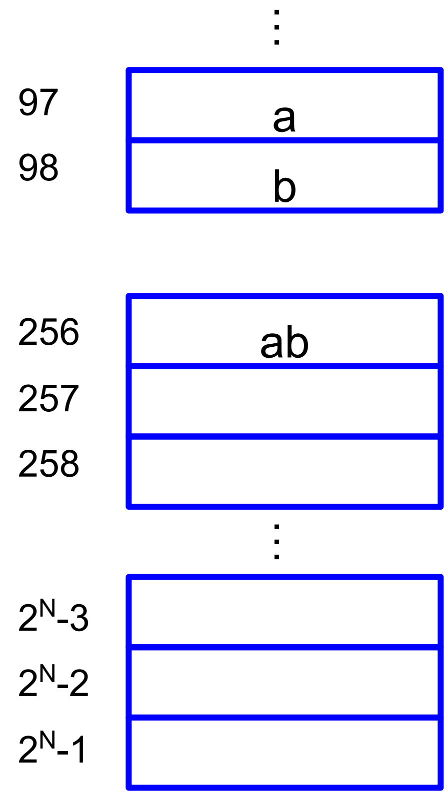
w = get input symbol;
while (there are still input symbols)
{
k = read a character
if w + k exists in the dictionary
w = w + k;
else
add w+k to the dictionary;
output the code for w;
w = k;
}
Output the code for w

w = a

k = b

ab

w=b



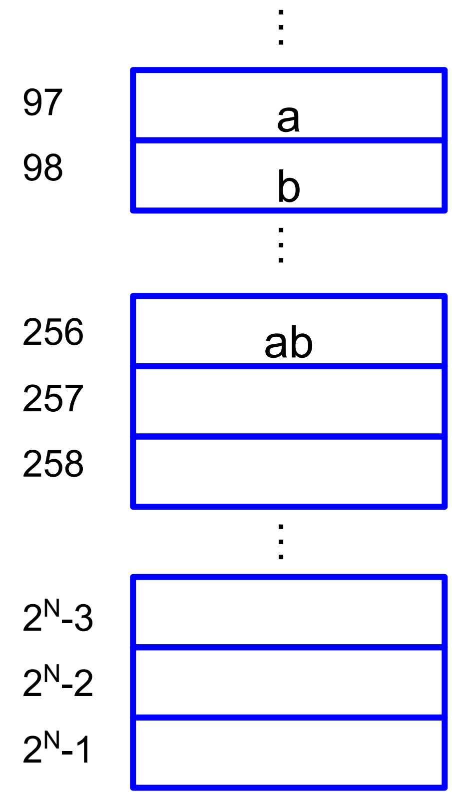
Output: 97,

LZW Compression Algorithm

ab*bbabbbab*

w = get input symbol;
while (there are still input symbols)
{
k = read a character
if w + k exists in the dictionary
w = w + k;
else
add w+k to the dictionary;
output the code for w;
w = k;
}
Output the code for w

w = b



Output: 97,

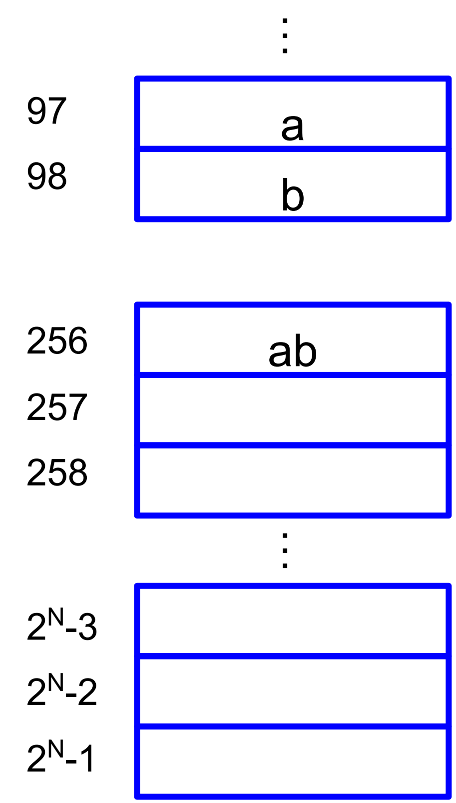
LZW Compression Algorithm

abb*babbbab*

w = get input symbol;
while (there are still input symbols)
{
k = read a character
if w + k exists in the dictionary
w = w + k;
else
add w+k to the dictionary;
output the code for w;
w = k;
}
Output the code for w

w = b

k = b



Output: 97,

LZW Compression Algorithm

***abb**babbbab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = b

k = b

bb

w=b

⋮	
97	a
98	b
256	ab
257	bb
258	
⋮	
2^N-3	
2^N-2	
2^N-1	

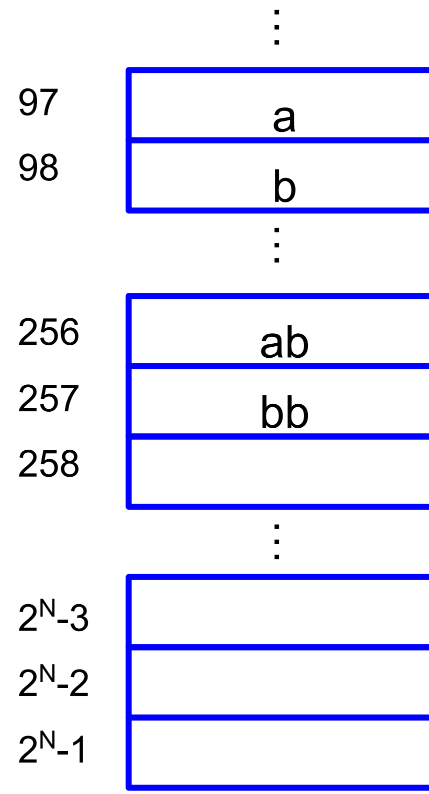
Output: 97, 98,

LZW Compression Algorithm

abb*babbbab*

```
w = get input symbol;
while ( there are still input symbols )
      {
      k = read a character
            if w + k exists in the dictionary
                  w = w + k;
            else
                  add w+k to the dictionary;
                  output the code for w;
                  w = k;
      }
Output the code for w
```

w = b



Output: 97, 98,

LZW Compression Algorithm

abbb*abbbab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = b

k = b

	⋮
97	a
98	b
256	ab
257	bb
258	
	⋮
2^N-3	
2^N-2	
2^N-1	

Output: 97, 98,

LZW Compression Algorithm

abbb*abbbab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = b

k = b

w=bb

	⋮
97	a
98	b
256	ab
257	bb
258	
	⋮
2^N-3	
2^N-2	
2^N-1	

Output: 97, 98,

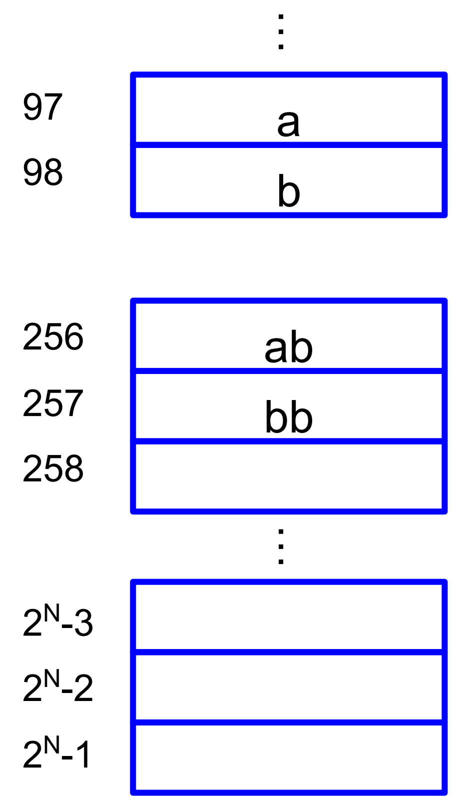
LZW Compression Algorithm

abbba*bbbab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = bb

k = a



Output: 97, 98,

LZW Compression Algorithm

***abbba**bbbab*

```
w = get input symbol;
while ( there are still input symbols )
      {
      k = read a character
            if w + k exists in the dictionary
                  w = w + k;
            else
                  add w+k to the dictionary;
                  output the code for w;
                  w = k;
      }
Output the code for w
```

w = bb

k = a

bba

w=a

⋮	
97	a
98	b
256	ab
257	bb
258	bba
⋮	
2^N-3	
2^N-2	
2^N-1	

Output: 97, 98, 257,

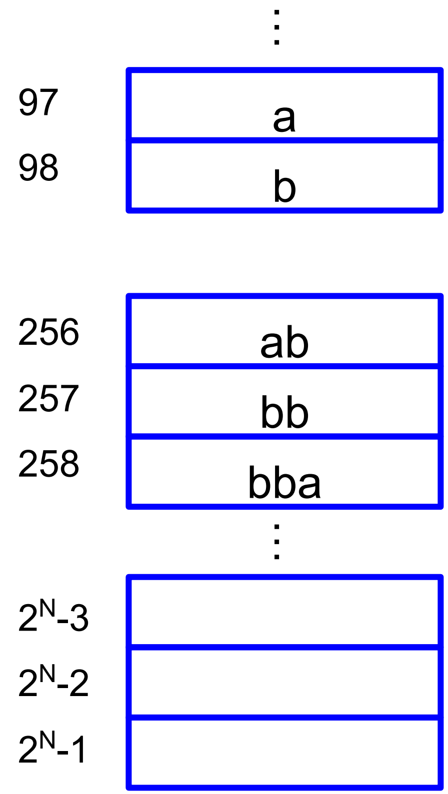
LZW Compression Algorithm

abbbab*bbab*

w = a

k = b

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```



Output: 97, 98, 257

LZW Compression Algorithm

abbbab*bbab*

```
w = get input symbol;
while ( there are still input symbols )
      {
      k = read a character
            if w + k exists in the dictionary
                  w = w + k;
            else
                  add w+k to the dictionary;
                  output the code for w;
                  w = k;
      }
Output the code for w
```

w = a

k = b

w=ab

	⋮
97	a
98	b
256	ab
257	bb
258	bba
	⋮
2^N-3	
2^N-2	
2^N-1	

Output: 97, 98, 257,

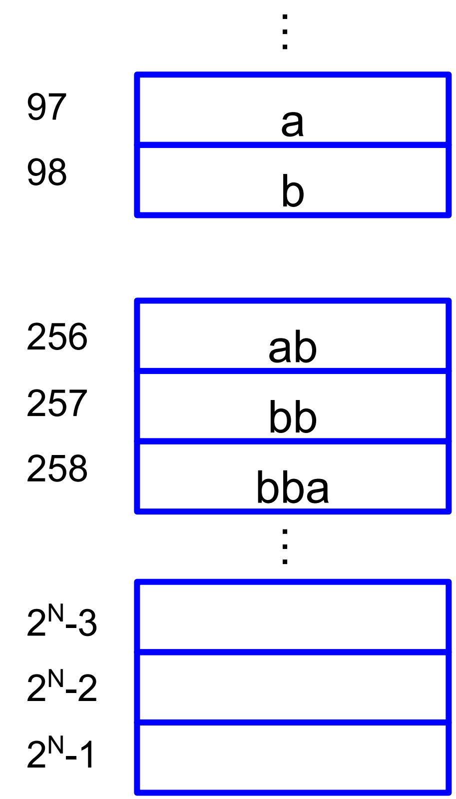
LZW Compression Algorithm

abbbabb*bab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = ab

k = b



Output: 97, 98, 257

LZW Compression Algorithm

abbbabb_bab_

```
w = get input symbol;
while ( there are still input symbols )
      {
      k = read a character
            if w + k exists in the dictionary
                  w = w + k;
            else
                  add w+k to the dictionary;
                  output the code for w;
                  w = k;
      }
Output the code for w
```

w = ab

k = b

abb

w=b

	⋮
97	a
98	b
256	ab
257	bb
258	bba
259	abb
	⋮
2^N-2	
2^N-1	

Output: 97, 98, 257, 256

LZW Compression Algorithm

abbbabbb*ab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = b

k = b

	⋮
97	a
98	b
256	ab
257	bb
258	bba
259	abb
	⋮
2^N-2	
2^N-1	

Output: 97, 98, 257, 256

LZW Compression Algorithm

abbbabbb*ab*

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = b

k = b

w=bb

⋮	⋮
97	a
98	b
256	ab
257	bb
258	bba
259	abb
⋮	⋮
2^N-2	
2^N-1	

Output: 97, 98, 257, 256

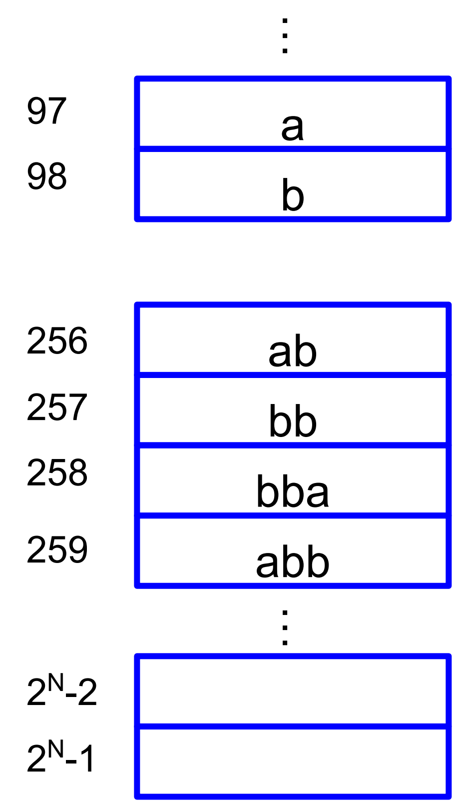
LZW Compression Algorithm

abbbabbbba_b_

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = bb

k = a



Output: 97, 98, 257, 256

LZW Compression Algorithm

abbbabbba*b*

w = get input symbol;
while (there are still input symbols)
{
k = read a character
if w + k exists in the dictionary
w = w + k;
else
add w+k to the dictionary;
output the code for w;
w = k;
}
Output the code for w

w = bb

k = a

w=bba

	⋮
97	a
98	b
256	ab
257	bb
258	bba
259	abb
	⋮
2^N-2	
2^N-1	

Output: 97, 98, 257, 256

LZW Compression Algorithm

abbbabbbab

```
w = get input symbol;
while ( there are still input symbols )
      {
      k = read a character
            if w + k exists in the dictionary
                  w = w + k;
            else
                  add w+k to the dictionary;
                  output the code for w;
                  w = k;
      }
Output the code for w
```

w = bba

k = b

	⋮
97	a
98	b
256	ab
257	bb
258	bba
259	abb
	⋮
2^N-2	
2^N-1	

Output: 97, 98, 257, 256

LZW Compression Algorithm

abbbabbbab

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
Output the code for w
```

w = bba

k = b

bbab

w=b

⋮	
97	a
98	b
256	ab
257	bb
258	bba
259	abb
260	bbab
⋮	
2^N-1	

Output: 97, 98, 257, 256, 258, 98

LZW Compression Algorithm

abbbabbbab **10 symbols of 8 bits each → 80 bit sequence**

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
```

Output: 97, 98, 257, 256, 258, 98 **6 symbols of N bits each → 6N bit sequence**

LZW Compression Algorithm

abbbabbbab **10 symbols of 8 bits each → 80 bit sequence**

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
```

As long as $6N < 80$ → compression

$N < 40/3$ ~13,333 bits

Ex: $N = 12$ bits → Rate = 6 x 12 / 80 = 0,9

$N_{min} = 9$ bits → Rate = 6 x 9 / 80 = 0,675

Output: 97, 98, 257, 256, 258, 98 **6 symbols of N bits each → 6N bit sequence**

LZW Compression Algorithm

abbbabbbab

```
w = get input symbol;
while ( there are still input symbols )
    {
    k = read a character
        if w + k exists in the dictionary
            w = w + k;
        else
            add w+k to the dictionary;
            output the code for w;
            w = k;
    }
```

Clearly, for this particular sequence there might have been an advantage to use a smaller starting dictionary (only containing a and b)

Best case (if you look back and try to optimize) would require 3 bits per symbol in the output: 18 bits

For such small sequence, how much of we set 1 bit for representing a or b?

Output: 97, 98, 257, 256, 258, 98

Lossless Compression Algorithms (Pattern Substitution)

- We substitute frequently repeating patterns with a code
- The code is shorter than the pattern

A simple Pattern Substitution scheme could employ predefined code

Example: replace all occurrences of `The' with the code '&'

Lossless Compression Algorithms (Pattern Substitution)

More typically tokens are assigned to according to frequency of occurrence of patterns:

- Count occurrence of tokens
- Sort in Descending order
- Assign some symbols to highest count tokens

A predefined symbol table may used i.e. assign code *i* to token *i* (good if you know the application)

It is more usual to dynamically assign codes to tokens

(We will see a different example for Delta Encoding later on)

Other Compression Techniques

Simple Repetition Suppression

- If *n* successive tokens appear, we can replace them with a token and a count number of occurrences
- We usually need to have a special **flag** to denote when the repeated token appears

Example: 89400000000000000000000000000000000

can be replaced with

894f32

where **f** is the flag for zero

- Compression savings depend on the content of the data

- **Applications:** Zero Length Suppression for silence in audio data, pauses in conversation, blanks in text or program source files, backgrounds in images

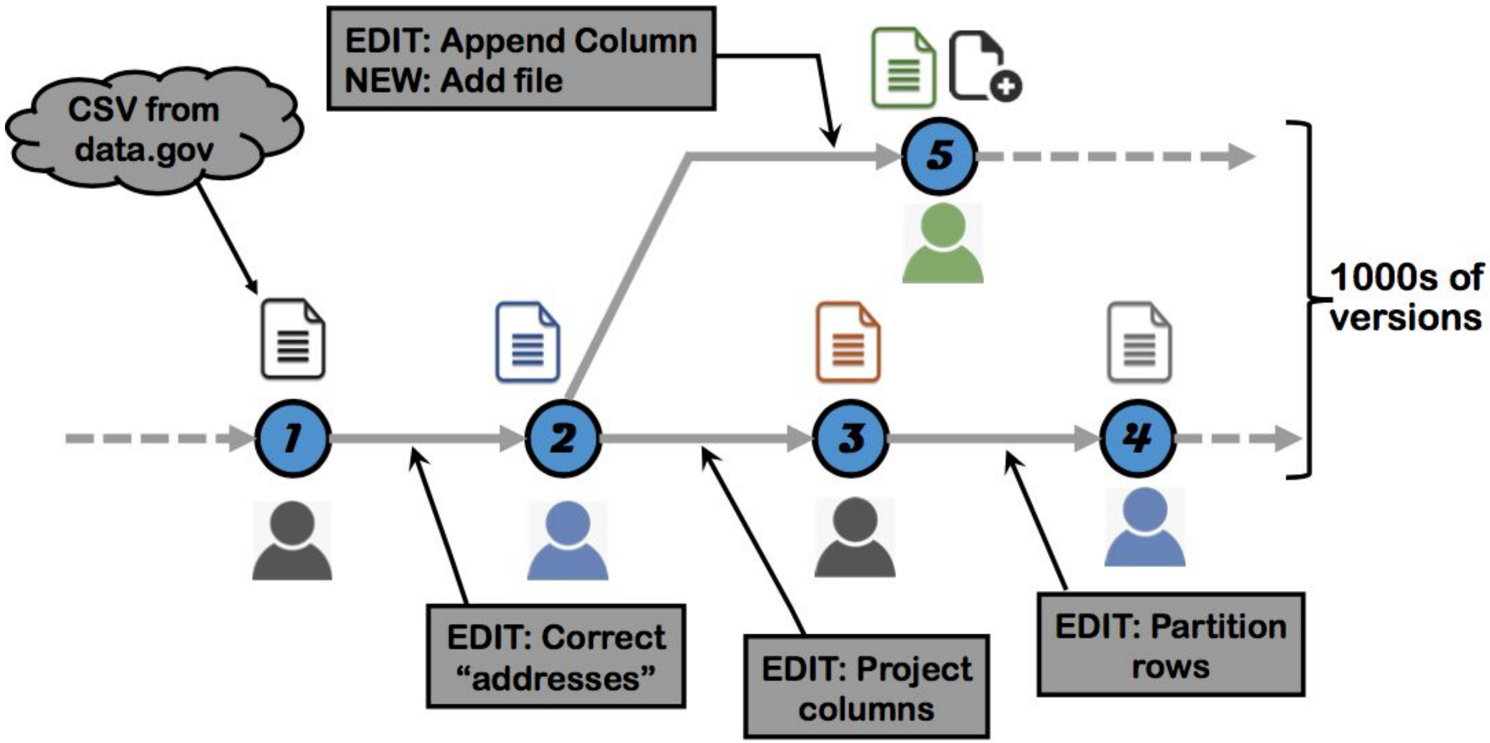
Run-length Encoding

- Frequently applied to images (or pixels in a scan line).
- Used in JPEG (small part of the process)
- Sequences of image elements $X_1, X_2, \ldots, X_n$ are mapped to pairs $(c_1, l_1), (c_1, L_2), \ldots, (c_n, l_n)$ where c_i represent image intensity or colour and l_i the length of the *i*-th run of pixels
- **Example:**
  Original Sequence: 111122233333311112222

  can be encoded as: (1,4),(2,3),(3,6),(1,4),(2,4)
- Worst case: encoding is more heavy than original file 2*integer rather 1* integer if data is represented as integers

Data Versioning

A typical data analysis workflow



The dataset versioning problem

- Many private copies of the datasets lead to **massive redundancy** in storage
- No easy way to keep **track of dependencies**
- No mechanisms to support and record manual **conflict resolution**
- No way to **analyze/compare/query** versions (across users)

Dataset version control desiderata

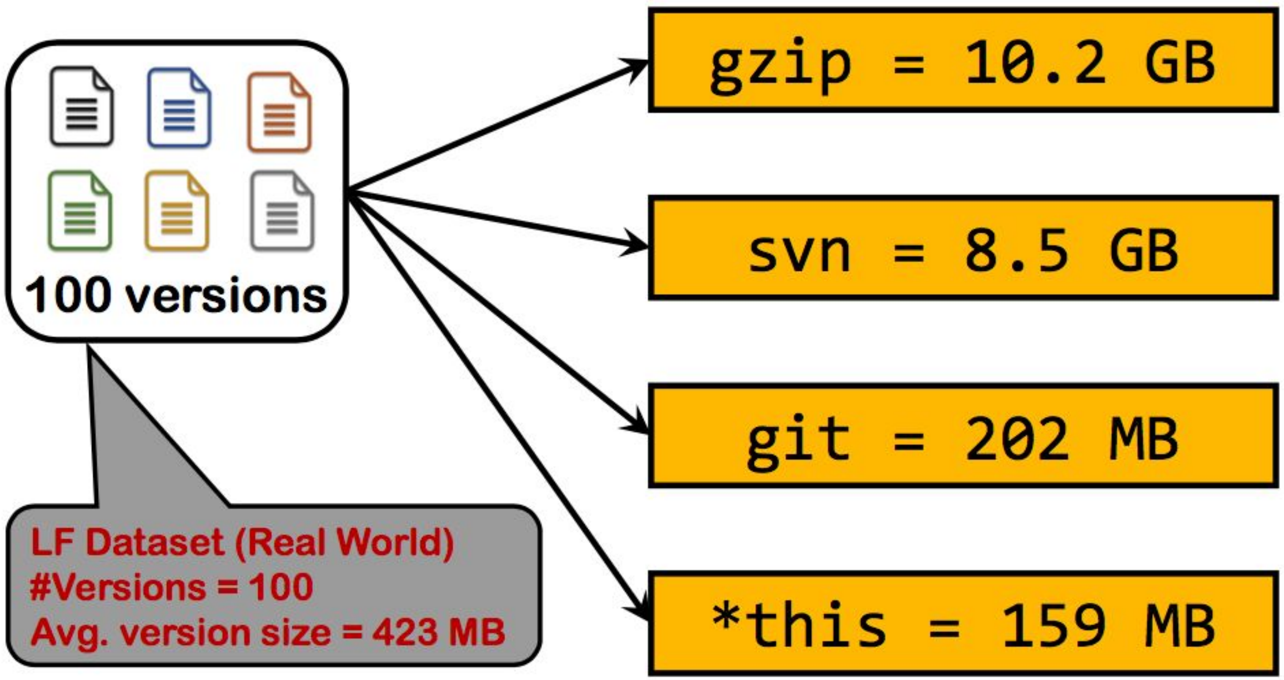
- Branch, update, merge, transform
  - Large unstructured or structured datasets

- Main challenges:
  - How can we store thousands of versions of datasets compactly?
  - How to access any version, on-demand, efficiently?

Version Control Systems

- We already have Git/SVN and many more
- Versioning algorithms optimized to work with code-like data
  - Sparse
  - Local changes (focused in specific parts of the file)
- Scenario: What if we reformat a date that appears in all tuples of a structured dataset?

Version Control Systems in Practice

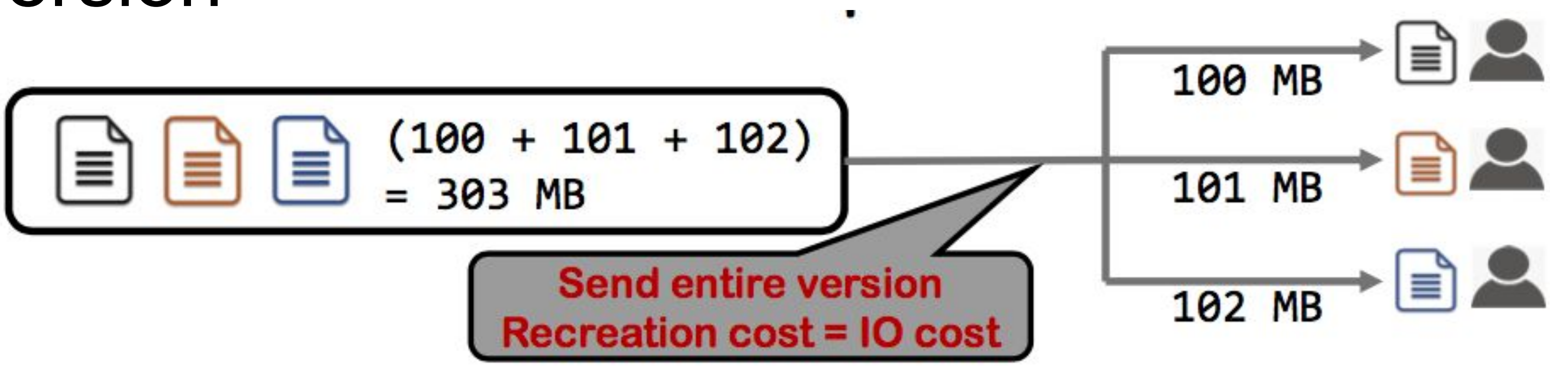


Even git uses large amounts of RAM for large files!

Costs

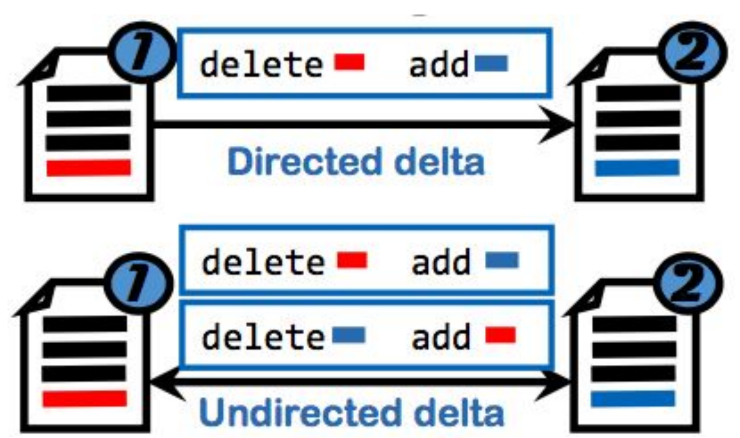
Storage Cost: Space to store all versions

Recreation Cost: time required to access a version



How to recreate a version?

- Use **deltas:** A delta between versions is a file which allows constructing one version given the other
- A delta has its own storage cost and recreation cost
- Examples: Unix diff, xdelta, VCDIFF



VCDIFF (RFC 3284)

Uses a source (can be empty as an input) to encode an incoming file and perform delta compression

Represents a file using basic operations (including):

COPY(t,p) defines the *t* number of bytes to use starting from position *p* in the dictionary/source

ADD(t,s) indicates the *t* number of bytes to add to the dictionary (and the new file) with a sequence *s* (s has t bytes)

RUN (x,s): A byte b will be repeated x times

https://tools.ietf.org/html/rfc3284

VCDIFF (RFC 3284)

File V_1: 0,1, 2, 3, 4, 5, 6, 7, 8, 9

can be a **source** in VCDIFF terms (a prior dictionary)

File V_2: 0, 1, 2, 7, 8, 9, 3, 4, 5, 6, F, D

arrives, the delta representation of this file is

{COPY (3,0), COPY(3,7), COPY(4,3), ADD(2, {F,D}) }

→ Recognized two sequences of 3 bytes & one of 4 bytes that could be used in the delta representation

VCDIFF (RFC 3284)

File V_1: 0,1, 2, 3, 4, 5, 6, 7, 8, 9
can be a **source** in VCDIFF terms (a prior dictionary)

File V_2: 0, 1, 2, 7, 8, 9, 3, 4, 5, 6, F, D
arrives, the delta representation of this file is

{COPY (3,0), COPY(3,7), COPY(4,3), ADD(2, {F,D}) }

→ Performance is usually close to GZIP

→ Can use multiple previous files as source

VCDIFF - Example

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d w x y z e f g h e f g h e f g h e f g h z z z z

VCDIFF - Example

Position 0

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d w x y z e f g h e f g h e f g h e f g h z z z z

COPY 4, 0

VCDIFF - Example

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d **w x y z** e f g h e f g h e f g h e f g h z z z z

COPY 4, 0

ADD 4, w x y z

VCDIFF - Example

Position 4

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d w x y z **e f g h** e f g h e f g h e f g h z z z z

COPY 4, 0

ADD   4, w x y z

COPY 4, 4

VCDIFF - Example

Position 0

Position 15

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d w x y z e f g h **e f g h e f g h e f g h** z z z z

Position 16

Position 24

12 in length

COPY 4, 0

ADD 4, w x y z

COPY 4, 4

COPY **12**, 24

VCDIFF - Example

Source:

a b c d e f g h i j k l m n o p

Target:

a b c d w x y z e f g h e f g h e f g h e f g h **z z z z**

COPY 4, 0

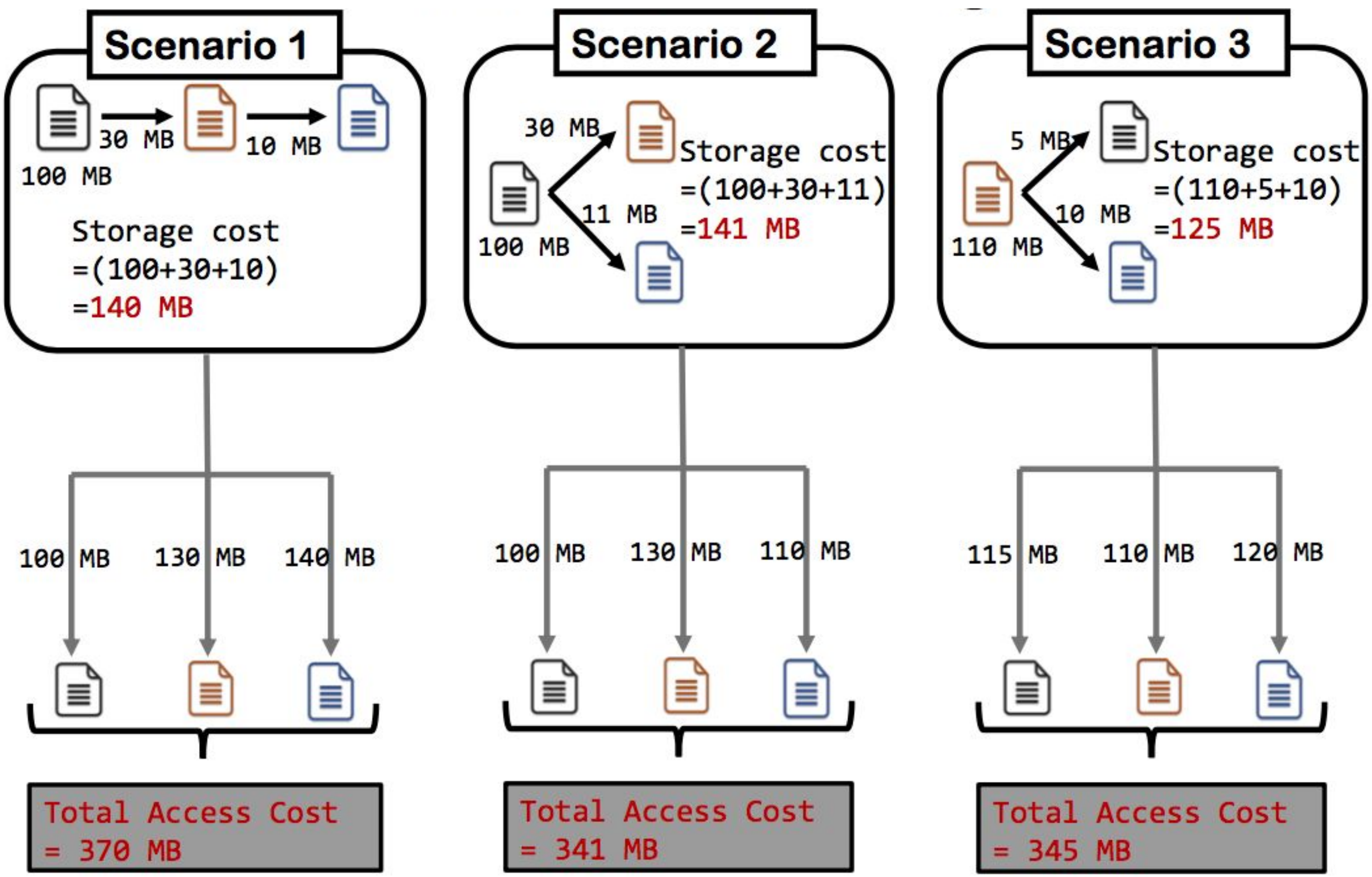
ADD 4, w x y z

COPY 4, 4

COPY 12, 24

RUN 4, z

Storage/Recreation Tradeoff (with delta encoding)



Problem

Find a storage solution that:

- Minimizes (total) recreation cost within storage budget
- Minimizes maximum recreation cost within a storage budget