Theory of Computation: Homework 2

Author

- Name: 윤준혁
- Student ID: 2023-23475
- Date: 2025-05-07

Execution Environment

- Operating System: Ubuntu 22.04.3 LTS
- Compiler: g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
- Compile option: `std=c++17`

Implementation

LZ78 incrementally builds an explicit dictionary (phrase trie). Each output token is a pair (**`index`**, **`next_char`**) where

- `index` = dictionary phrase index of the longest prefix found so far (0 for the empty string),
- `next_char` = the first symbol that extends that prefix in the input.

The implementation has the following features:

- 7-bit fixed symbol codes for an extended 71-symbol alphabet,
- variable-width phrase indices (`k` bits, doubled when `2^k` phrases are created),
- compact bit-packing via custom `BitWriter` / `BitReader`,
- trie-based phrase lookup for $O(L)$ factorisation (where L is input length).

Encoder `encoder.cpp`

Global Constants and Data Structures

Symbol	Meaning	Value / Structure
`ALPHABET_SIZE`	Total supported symbols	`62+9=71`
`CHAR_BITS`	Bits per symbol code	`7` ($\lceil \log_2 71 \rceil$)
`SENTINEL`	Pseudo-symbol signalling EOF	71
`struct Node`	Trie node	`child[72]`, `phrase_idx`
`vector<Node> trie`	Global dynamic trie	root at index 0

- The implementation uses a flat array of children for each node, enabling O(1) branching.
- The trie is constructed using a vector, allowing it to grow in size as needed dynamically.

`code(char c)`

Purpose. Deterministically map an ASCII character in the supported alphabet to a 0-based integer.

Input. `char c`

Output. `int idx ∈ [0,70]` or `−1` (invalid).

Algorithm. Cascaded range checks (`'a'..'z'`, `'A'..'Z'`, `'0'..'9'`) followed by a `switch` on punctuation.

Time. O(1)

`make_new_node()`, `init_trie()`

- `make_new_node()` appends a zero-initialised `Node` to `trie`, returns its index.
- `init_trie()` clears the trie, creates the root, and sets `phrase_idx=0` to represent the empty phrase.

`class BitWriter`

A little-endian bit accumulator that flushes every full byte to an `ostream`.

Member	Role
`uint64_t buf`	64-bit staging buffer
`int bits`	Count of valid bits currently in `buf`
`put(value,n)`	OR-in *n* LSBs of `value`; emit bytes while possible
`flush()`	Zero-pad and output residual bits

Time. Amortised O(1) per call; every bit is written exactly once.

`encode(std::istream&, std::ostream&)`

High-level Algorithm

```
root ← init_trie(); k ← 1; next_idx ← 1; v ← root
while (true):
    ch ← next input character or EOF
    cidx ← code(ch) or SENTINEL
    if trie[v].child[cidx] exists:
        v ← that child
        continue        // extend match
    output ⟨trie[v].phrase_idx, cidx⟩
    insert new_node as child(v, cidx) with phrase_idx = next_idx++
    if next_idx == 2^k: k++    // widen index field
    v ← root
    if ch == EOF: break
flush bit buffer
```

Detailed Function Roles

Step	Code Fragment	Functionality
Longest-match scan	`nxt = trie[v].child[cidx]; if(nxt)`	Single-step trie traversal— `v` always holds the deepest match so far.

Token emission	`bw.put(..., k); bw.put(..., 7);`	Packs phrase index (`k` bits) then symbol code (7 bits).
Dictionary growth	`make_new_node()`	Adds **phrase = (matching string + next_char)**.
Dynamic index width	`if (next_phrase_idx == 1<<k) ++k;`	Doubling range when necessary.
EOF handling	When `fin.eof()` is reached, one final token with `next_char = SENTINEL` is written; decoder recognises this sentinel to terminate.	

Complexity

- time = O(L)
- space = O(P), where P ≤ L+1 is the number of phrases.

Decoder `decoder.cpp`

Global Constants and Data Structures

Identical to the encoder with an augmented `Node`:

- `parent` (index of predecessor node)
- `ch` (the character leading from the parent to this node)

`code(char)` / `decode_char(int)`

Purpose. Deterministically map an ASCII character in the supported alphabet to a 0-based integer.

`decode_char` is the inverse; it returns the printable ASCII representation for a given 0-70 code.

Time. O(1)

`make_new_node()` , `init_trie()`

Same semantics as in the encoder, plus clear initialisation of `parent` and `ch` to `-1`.

`class BitReader`

Mirrors `BitWriter`:

Member	Purpose
`get(n)`	Returns the next *n* bits (little-endian) as an unsigned integer. Internally fills the buffer by reading bytes until enough bits are available.

`write_phrase(int, ostream&)`

Performs a reverse walk from node `idx` to the root, collecting `ch` along the way. The resulting string is reversed and streamed to `out`.

`decode(std::istream&, std::ostream&)`

High-level Algorithm

```
root ← init_trie(); k ← 1; next_idx ← 1
loop:
```

```
idx  ← br.get(k)
cidx ← br.get(7)
write_phrase(idx, fout)
if cidx == SENTINEL: break
ch ← decode_char(cidx); fout.put(ch)
new_node ← make_new_node()
trie[idx].child[cidx] = new_node
trie[new_node] = {parent=idx, ch=ch, phrase_idx=next_idx++}
if next_idx == 2^k: k++
```

1. **Token extraction.** Reads exactly `k+7` bits per iteration.
2. **Prefix reproduction.** `write_phrase` emits the entire phrase referenced by `idx`.
3. **Symbol append.** Writes `ch` (unless EOF).
4. **Dictionary update.** Inserts the new phrase = *prefix + ch*.
5. **Bit-width adaptation.** Keeps decoder's `k` in sync with encoder.

Complexity Analysis

Stage	Time	Space (extra)
Encoding	Θ(L)	Θ(P) nodes (≤ L+1)
Decoding	Θ(L)	Θ(P) nodes

- `L` = *length of input*
- `P` = *number of phrases*

Example running 1

```
(base) LAPTOP-PROFLWR:lz78:% ./run_encoder.sh infile.txt encoding.bin
Output written to encoding.bin
[+] encoding time (msec): 286.578 msec
[+] input size (byte): 1555051
[+] output size (byte): 770500
[+] compression ratio (%): 50.45%
(base) LAPTOP-PROFLWR:lz78:% ./run_decoder.sh encoding.bin outfile.txt
Decoded output written to outfile.txt
[+] decoding time (msec): 520.065 msec
(base) LAPTOP-PROFLWR:lz78:% diff infile.txt outfile.txt
```

input of encoding

- infile.txt

encoding time and file size (compression ratio)

[+] encoding time (msec): 286.578 msec
[+] input size (byte): 1555051
[+] output size (byte): 770500
[+] compression ratio (%): 50.45%

decoding time

[+] decoding time (msec): 520.065 msec

output of decoding

- same as infile.txt

command `diff`

- all same (see above screendump)

comparison of the efficiency and compression ratio with zip, gzip, and xz

```
(base) LAPTOP-PROFLWR:lz78:% zip encoding.zip infile.txt

updating: infile.txt (deflated 58%)
(base) LAPTOP-PROFLWR:lz78:% gzip -c infile.txt > encoding.gz
(base) LAPTOP-PROFLWR:lz78:% xz -c infile.txt > encoding.xz
(base) LAPTOP-PROFLWR:lz78:% stat -c %s infile.txt

1555051
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding.bin

770500
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding.zip

650833
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding.gz

650692
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding.xz

537248
```

	no compression	LZ78	zip	gzip	xz
encoding (byte)	1555051	770500	650833	650692	537248
compression rate (%)	0.00	50.45	58.15	58.16	65.45

- compression ratio = (1- compressed data size/uncompressed data size) x 100
  - The lower the compression ratio, the better the compression.

Example running 2

```
(base) LAPTOP-PROFLWR:lz78:% ./run_encoder.sh input_ex1.txt encoding_ex1.bin
Output written to encoding_ex1.bin
[+] encoding time (msec): 1.11508 msec
[+] input size (byte): 17
[+] output size (byte): 11
[+] compression ratio (%): 35.29%
(base) LAPTOP-PROFLWR:lz78:% ./run_decoder.sh encoding_ex1.bin outfile_ex1.txt
Decoded output written to outfile_ex1.txt
[+] decoding time (msec): 0.270869 msec
(base) LAPTOP-PROFLWR:lz78:% diff input_ex1.txt outfile_ex1.txt
```

input of encoding

```
aaabbabaabaaabab
```

encoding time and file size (compression ratio)

[+] encoding time (msec): 1.11508 msec
[+] input size (byte): 17
[+] output size (byte): 11
[+] compression ratio (%): 35.29%

decoding time

[+] decoding time (msec): 0.270869 msec

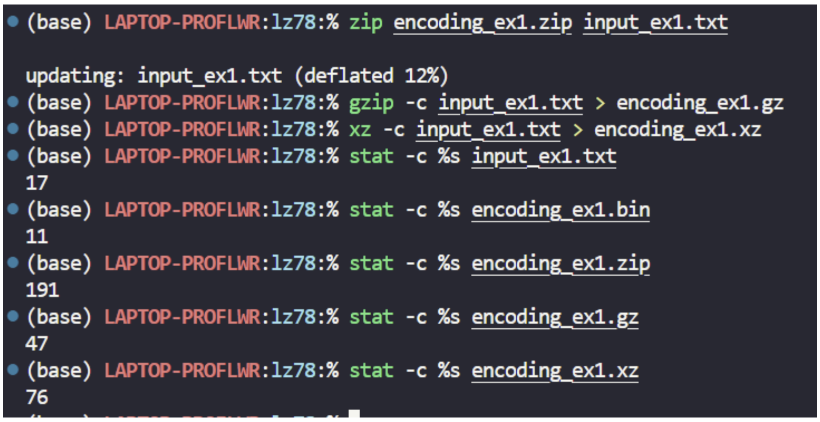
output of decoding

```
aaabbabaabaaabab
```

command `diff`

- all same (see above screendump)

comparison of the efficiency and compression ratio with zip, gzip, and xz



```
(base) LAPTOP-PROFLWR:lz78:% zip encoding_ex1.zip input_ex1.txt

updating: input_ex1.txt (deflated 12%)
(base) LAPTOP-PROFLWR:lz78:% gzip -c input_ex1.txt > encoding_ex1.gz
(base) LAPTOP-PROFLWR:lz78:% xz -c input_ex1.txt > encoding_ex1.xz
(base) LAPTOP-PROFLWR:lz78:% stat -c %s input_ex1.txt
17
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex1.bin
11
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex1.zip
191
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex1.gz
47
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex1.xz
76
```

	no compression	LZ78	zip	gzip	xz
encoding (byte)	17	11	191	47	76
compression rate (%)	0.00	35.29	-1023.53	-176.47	-347.06

- compression ratio = (1- compressed data size/uncompressed data size) x 100
  - The lower the compression ratio, the better the compression.

- **All other compression methods resulted in compressed files that were larger than the input file.**

Example running 3

```
● (base) LAPTOP-PROFLWR:lz78:% ./run_encoder.sh input_ex2.txt encoding_ex2.bin
  Output written to encoding_ex2.bin
  [+] encoding time (msec): 1.30227 msec
  [+] input size (byte): 1228
  [+] output size (byte): 886
  [+] compression ratio (%): 27.85%
● (base) LAPTOP-PROFLWR:lz78:% ./run_decoder.sh encoding_ex2.bin outfile_ex2.txt
  Decoded output written to outfile_ex2.txt
  [+] decoding time (msec): 1.9488 msec
● (base) LAPTOP-PROFLWR:lz78:% diff input_ex2.txt outfile_ex2.txt
○ (base) LAPTOP-PROFLWR:lz78:% █
```

input of encoding

```
What is Lorem Ipsum?::
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has b
een the industrys standard dummy text ever since the 1500s, when an unknown printer took a
galley of type and scrambled it to make a type specimen book. It has survived not only five cen
turies, but also the leap into electronic typesetting, remaining essentially unchanged. It was po
pularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, an
d more recently with desktop publishing software like Aldus PageMaker including versions of L
orem Ipsum.:::

Why do we use it?;;
It is a long established fact that a reader will be distracted by the readable content of a page w
hen looking at its layout. The point of using Lorem Ipsum is that it has a moreorless normal distr
ibution of letters, as opposed to using Content here, content here, making it look like readable E
nglish. Many desktop publishing packages and web page editors now use Lorem Ipsum as their
default model text, and a search for lorem ipsum will uncover many web sites still in their infanc
y. Various versions have evolved over the years, sometimes by accident, sometimes on purpos
e injected humour and the like.;
```

encoding time and file size (compression ratio)

[+] encoding time (msec): 1.30227 msec
[+] input size (byte): 1228
[+] output size (byte): 886
[+] compression ratio (%): 27.85%

decoding time

[+] decoding time (msec): 1.9488 msec

output of decoding

```
What is Lorem Ipsum?::
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has b
een the industrys standard dummy text ever since the 1500s, when an unknown printer took a
galley of type and scrambled it to make a type specimen book. It has survived not only five cen
turies, but also the leap into electronic typesetting, remaining essentially unchanged. It was po
pularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, an
d more recently with desktop publishing software like Aldus PageMaker including versions of L
orem Ipsum.:::

Why do we use it?;;
It is a long established fact that a reader will be distracted by the readable content of a page w
hen looking at its layout. The point of using Lorem Ipsum is that it has a moreorless normal distr
ibution of letters, as opposed to using Content here, content here, making it look like readable E
nglish. Many desktop publishing packages and web page editors now use Lorem Ipsum as their
default model text, and a search for lorem ipsum will uncover many web sites still in their infanc
y. Various versions have evolved over the years, sometimes by accident, sometimes on purpos
e injected humour and the like.;
```

command `diff`

- all same (see above screendump)

comparison of the efficiency and compression ratio with zip, gzip, and xz

```
(base) LAPTOP-PROFLWR:lz78:% zip encoding_ex2.zip input_ex2.txt

updating: input_ex2.txt (deflated 47%)
(base) LAPTOP-PROFLWR:lz78:% gzip -c input_ex2.txt > encoding_ex2.gz

(base) LAPTOP-PROFLWR:lz78:% xz -c input_ex2.txt > encoding_ex2.xz

(base) LAPTOP-PROFLWR:lz78:% stat -c %s input_ex2.txt
1228
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex2.bin
886
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex2.zip
822
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex2.gz
678
(base) LAPTOP-PROFLWR:lz78:% stat -c %s encoding_ex2.xz
772
(base) LAPTOP-PROFLWR:lz78:%
```

	no compression	LZ78	zip	gzip	xz
encoding (byte)	1228	886	822	678	772
compression rate (%)	0.00	27.85	33.06	44.79	37.13

- compression ratio = (1- compressed data size/uncompressed data size) x 100
    - The lower the compression ratio, the better the compression.