CS213/293 Data Structure and Algorithms 2025

Lecture 12: Data compression

Instructor: Ashutosh Gupta

IITB India

Compile date: October 3, 2025

Data compression

You must have used Zip, which reduces the space a file uses.

How does Zip work?

Fixed-length vs. Variable-length encoding

- Fixed-length encoding. Example: An 8-bit ASCII code encodes each character in a text file.
- Variable-length encoding: each character is given a different bit length encoding.
- We may save space by assigning fewer bits to the characters that occur more often.
- We may have to assign some characters more than 8-bit representation.

Example: Variable-length encoding

Example 12.1

Consider text: “agra”

- In a text file, the text will take 32 bits of space.
  - 01100001011001110111001001100001
- There are only three characters. Let us use encoding, $a =$ “0”, $g =$ “10”, and $r =$ “11”. The text needs six bits.
  - 010110

Exercise 12.1

Are the six bits sufficient?

Commentary: If the encoding depends on the text content, we also need to record the encoding along with the text.

Example: decoding variable-length encoding

Example 12.2

Consider encoding $a =$ "0", $g =$ "10", and $r =$ "11" and the following encoding of a text.

$$101100001110$$

The text is "*graaaarg*".

We scan the encoding from the left. As soon as a match is found, we start matching the next symbol.

Example: decoding bad variable-length encoding

Example 12.3

Consider encoding $a =$ "0", $g =$ "01", and $r =$ "11" and the following encoding of a text.

$$0111000011001$$

We cannot tell if the text starts with a "g" or an "a".

Prefix condition: Encoding of a character cannot be a prefix of encoding of another character.

Topic 12.1

An example of variable-length encoding: Unicode and UTF-8

How can we support all languages?

ASCII is designed for only english like languages.

How can we support all symbols ever used by humanity, including Emojis?

Answer: Unicode

Example 12.4

Character A is U+41
Character क is U+915
Emoji 😉 is U+1F609

Variable length encoding

The maximum length of unicode is 21 bits.

Storing unicode in a file

Unicode itself does not satisfy prefix condition.

If we do fixed-length encoding, we will be wasting space and ASCII files will be incompatible.

How to remain backward compatible and not waste space?

Answer: UTF-8

Unicode U+uvvvvwwwwxxxxyyyyzzzz will be encoded as follows.

Start code	Last code	Byte1	Byte2	Byte3	Byte4
U+00	U+7F	0yyyzzzz			
U+80	U+7FF	110xxxyy	10yyzzzz		
U+800	U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+10000	U+10FFFF	11110uvv	10vvwwww	10xxxxyy	10yyzzzz

UTF-8 ensures prefix condition.

Example: UTF-8 encoding

Example 12.5

Consider U+915, which is the code for character क.

wwww = 0000, xxxx = 1001, yyyy = 0001, zzzz = 0101

The code will be stored using the following three bytes.

$$11100000\ 10100100\ 10010101$$

Topic 12.2

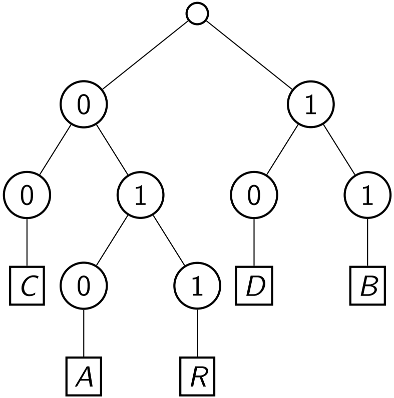
Principles of variable-length encoding

Encoding trie

Definition 12.1

An encoding trie is a binary trie that has the following properties.

- Each terminating leaf is labeled with an encoded character.
- The left child of a node is labeled 0 and the right child of a node is labeled 1

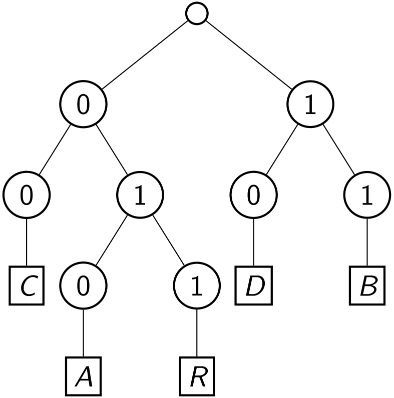


Character encoding/codewords:
$C = 00$, $A = 010$, $R = 011$, $D = 10$, and $B = 11$.

Exercise 12.2

Show: An encoding trie ensures that the prefix condition is not violated.

Example: Decoding from a Trie



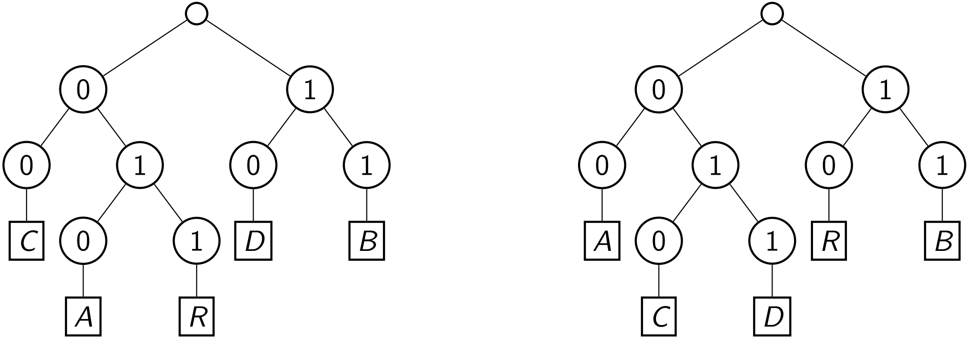
Encoding: 0101101101000010100101011011010

Text: ABRACADABRA

Encoding length

Example 12.6

Let us encode ABRACADABRA using the following two tries.

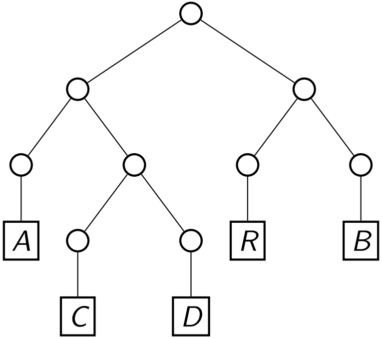


Encoding:(29 bits)
01011011010 0001010 01011011010

Encoding:(24 bits)
00111000 01000011 00111000

Drawing with tries without labels

Since we know the label of an internal node by observing that a node is a left or right child, we will not write the labels.



Commentary: We can assign any bit to a node as long as the sibling will use a different bit.

Topic 12.3

Optimal compression

Optimal compression

Different tries will result in different compression levels.

Design principle: We encode a character that occurs more often with fewer bits.

frequency

Definition 12.2

The frequency f_c of a character c in a text T is the number of times c occurs in T.

Example 12.7

The frequencies of the characters in ABRACADABRA are as follows.

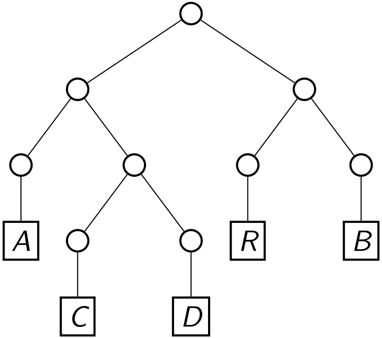
- $f_A = 5$
- $f_B = 2$
- $f_R = 2$
- $f_C = 1$
- $f_D = 1$

Characters encoding length

Definition 12.3

The encoding length l_c of a character c in a trie is the number of bits needed to encode c.

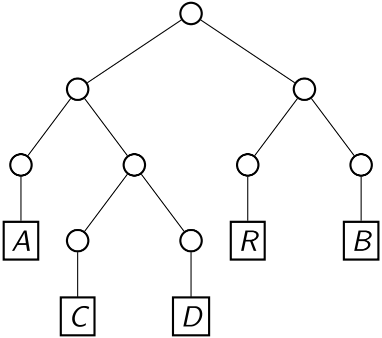
Example 12.8



In the left trie, the encoding length of the characters are as follows.

- $l_A = 2$
- $l_B = 2$
- $l_R = 2$
- $l_C = 3$
- $l_D = 3$

Weighted path length == number of encoded bits



The total number of bits needed to store a text is

$$\sum_{c \in Leaves} f_c l_c.$$

Example 12.9

The number of bits needed for ABRACADABRA using the left trie is the following sum.

$$f_A * l_A + f_C * l_C + f_D * l_D + f_R * l_R + f_B * l_B$$

$$= 5 * 2 + 1 * 3 + 1 * 3 + 2 * 2 + 2 * 2 = 24$$

Is this the best trie for compression? How can we find the best trie?

Huffman encoding

Algorithm 12.1: HUFFMAN(Integers $f_{c_1}, ..., f_{c_k}$)

```
for i ∈ [1, k] do
    N := CreateNode(c_i, Null, Null);
    T_i := CreateNode(f_{c_i}, N, Null);
return BuildTree(T_1, ..., T_k)
```

CREATENODE(Value, LeftChild, RightChild) is a constructor of a node.

Algorithm 12.2: BUILDTREE(Nodes $T_1, ..., T_k$)

```
if k == 1 then
    return T_1
Find T_i and T_j such that value(T_i) and value(T_j) are minimum;
T_new := CreateNode(value(T_i) + value(T_j), T_i, T_j);
return BuildTree(T_1, ..., T_{i-1}, T_{i+1}, ..., T_{j-1}, T_{j+1}, ..., T_k, T_new)
```

Exercise 12.3

a. Is BuildTree tail recursive?

b. How should we resolve non-determinism if there is a tie in finding the minimum?

Running time analysis of Huffman encoding

We need to find minimums repeatedly. We use a heap to store the values of the roots.

Running time analysis

- BuildTree will be recursively called k times.
- In each recursive call, we need to call
  - two deleteMins for removing two trees and
  - an insertion for the new tree

  in the heap.
- Total running time

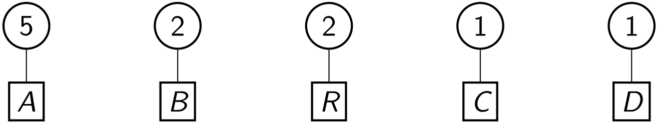
$$\sum_{i=k}^{1} O(\log i) = O(k \log k)$$

Commentary: We have proven the above equality in tutorial problems!

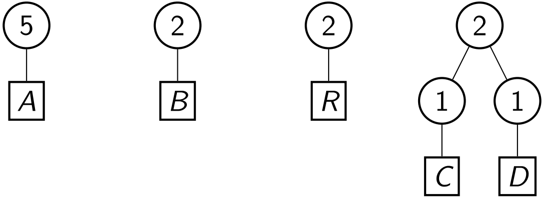
Example: Huffman encoding

Example 12.10

After initialization.

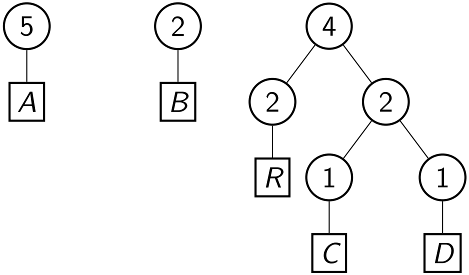


We choose nodes labeled with 1 to join and create a larger tree.

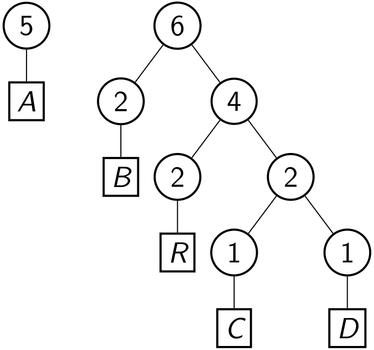


Example: Huffman encoding(2)

After the next recursive step

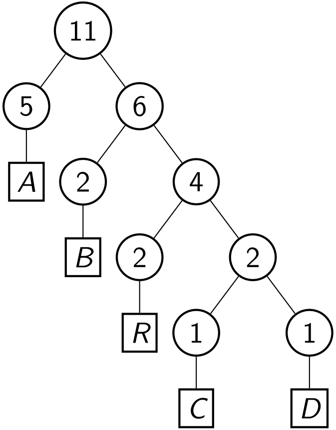


After another recursive step:

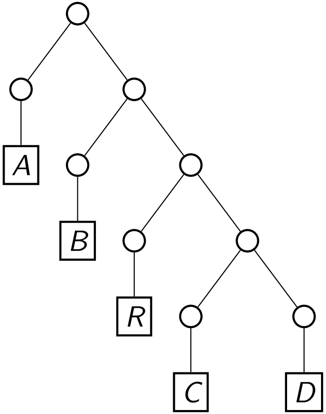


Example: Huffman encoding(3)

After the final recursive step:



We scrub the frequency labels.



Exercise 12.4

How many bits do we need to encode ABRACADABRA?

Using Huffman in compression

To compress a file, we need to compute the frequencies of the symbols.

The number of symbols may be constant with respect to the file size.

Therefore, the cost of computing Huffman is constant time if the frequencies are given.

Frequency (alternative definition!)

Definition 12.4 (Equivalent definition)

The frequency f_c of a character c in a text T is the fraction (or %) of times c occurs in T.

Example 12.11

The frequencies of the characters in ABRACADABRA are as follows.

- $f_A = 5/11$
- $f_B = 2/11$
- $f_R = 2/11$
- $f_C = 1/11$
- $f_D = 1/11$

Huffman can work with the fractions without any change. (Why?)

Topic 12.4

Proof of optimality of Huffman encoding

Is Huffman optimal?

Exercise 12.5

Let us suppose a file contains *a*, *b*, *c*, and *d* with frequencies 25%, 25%, 25%, and 25% respectively.

a. Should you be able to compress this file?

b. Do Huffman codes compress this file?

We need to prove that Huffman encoding indeed produces optimal encoding.

Minimum weighted path length

Definition 12.5

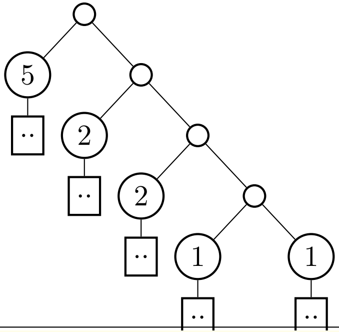
Given frequencies $f_{c_1}, ..., f_{c_k}$, minimum weighted path length $MWPL(f_{c_1}, ..., f_{c_k})$ is the minimum weighted path length among the tries that encode $c_1, ..., c_k$.

We say a trie is a witness of $MWPL(f_{c_1}, ..., f_{c_k})$ if it encodes $c_1, ..., c_k$ and it produces encoding of length $MWPL(f_{c_1}, ..., f_{c_k})$ for a text with frequencies $f_{c_1}, ..., f_{c_k}$

Example 12.12

We have seen $MWPL(5, 2, 2, 1, 1) = 23$.

The witness trie is on the right.

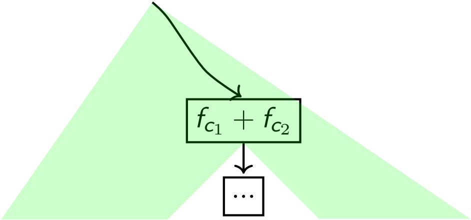


Commentary: The MWPL is the property of the frequency distribution.

A recursive relation

Theorem 12.1
$MWPL(f_{c_1}, ..., f_{c_k}) \leq f_{c_1} + f_{c_2} + MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$

Proof.
Let trie T be a witness of $MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$ containing a node labeled with $f_{c_1} + f_{c_2}$ with a terminal child.

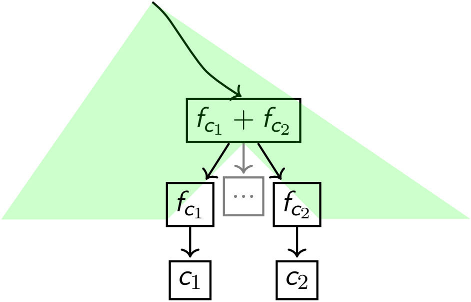


...

A recursive relation(2)

Proof(contd.)

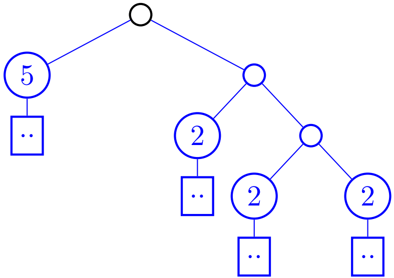
We construct a trie for frequencies $f_{c_1}, ..., f_{c_k}$ such that the weighted path length of the trie is $f_{c_1} + f_{c_2} + MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$.



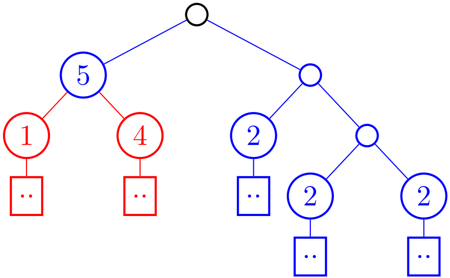
Therefore, $MWPL(f_{c_1}, ..., f_{c_k})$ must be less than equal to the above expression. □

Example: $MWPL(f_{c_1}, ..., f_{c_k}) \leq f_{c_1} + f_{c_2} + MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$

Example 12.13



Witness for $MWPL(5, 2, 2, 2)$



The weighted path length of the above is $1 + 4 + MWPL(5, 2, 2, 2)$

The witness of $MWPL(1, 4, 2, 2, 2)$ must have weighted path length $\leq$ the above right trie.

$$MWPL(1, 4, 2, 2, 2) \leq 1 + 4 + MWPL(5, 2, 2, 2)$$

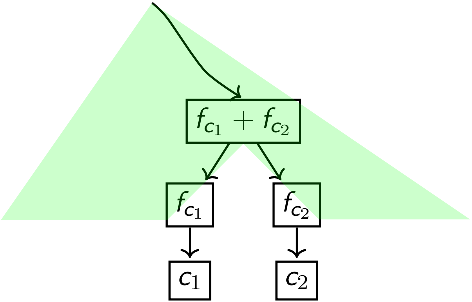
Reverse recursive relation

Theorem 12.2

If f_{c_1} and f_{c_2} are the minimum two, $MWPL(f_{c_1}, ..., f_{c_k}) = f_{c_1} + f_{c_2} + MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$.

Proof.

There is a witness of $MWPL(f_{c_1}, ..., f_{c_k})$ where the parents of c_1 and c_2 are siblings. (Why?)

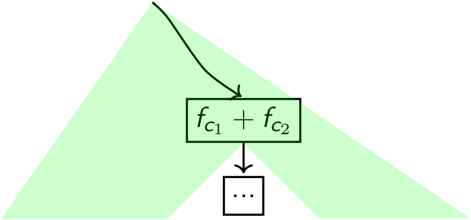


Commentary: Explaining why: Show that the smallest frequency symbol can always be moved to the last level to improve weighted path length. Furthermore, since there must be a sibling at the last level, the second last frequency symbol can also be moved to the sibling to improve the weighted path length.

Reverse recursive relation(2)

Proof(contd.)

We construct a tree for frequencies $f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k}$ such that the weighted path length of the tree is $MWPL(f_{c_1}, ..., f_{c_k}) - f_{c_1} - f_{c_2}$.

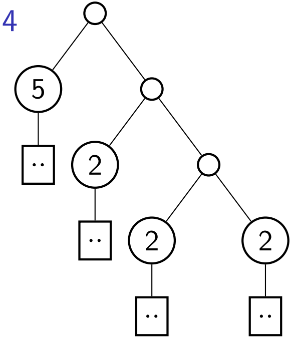


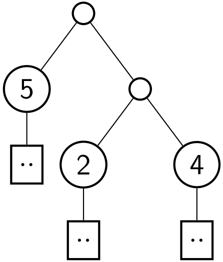
Therefore, $MWPL(f_{c_1}, ..., f_{c_k}) - f_{c_1} - f_{c_2} \geq MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$.

Due to the previous theorem, $MWPL(f_{c_1}, ..., f_{c_k}) = f_{c_1} + f_{c_2} + MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k})$. □

Example: $MWPL(f_{c_1} + f_{c_2}, f_{c_3}, ..., f_{c_k}) \leq MWPL(f_{c_1}, ..., f_{c_k}) - f_{c_1} - f_{c_2}$

Example 12.14





Witness for $MWPL(5, 2, 2, 2)$. Since 2 and 2 are the least two frequencies, they are on the longest path.

The weighted path length of the above is $MWPL(5, 2, 2, 2) - 2 - 2$

The witness of $MWPL(5, 2, 4)$ must have weighted path length $\leq$ the above right trie.

$$MWPL(5, 2, 4) \leq MWPL(5, 2, 2, 2) - 2 - 2$$

Proof compatible BuildTree

Our BuildTree does not follow the pattern of updates of theorem 12.2. So, writing a proof over the algorithm is hard. We consider the following version of BuildTree for writing the proof.

Algorithm 12.3: BuildTree2(Nodes $T_1, ..., T_k$)

if $k == 1$ **then**
  return T_1

Find T_i and T_j such that $value(T_i)$ and $value(T_j)$ are minimum;

$T_{new} :=$ CreateNode($value(T_i) + value(T_j)$, *Null*, *Null*);

$T :=$ BuildTree2($T_1, ..., T_{i-1}, T_{i+1}, ..., T_{j-1}, T_{j+1}, ..., T_k, T_{new}$);

$left(T_{new}) := T_i$;

$right(T_{new}) := T_j$;   //Only highlighted parts are different from BuildTree.

return T

Exercise 12.6

Show that algorithms 12.2 and 12.3 are equivalent.

Commentary: In the proof, the induction step is non-trivial to understand.

Correctness of HUFFMAN

Theorem 12.3

HUFFMAN$(f_{c_1}, ..., f_{c_k})$ always returns a tree that is a witness of $MWPL(f_{c_1}, ..., f_{c_k})$.

Proof.

We assume HUFFMAN is calling BUILDTREE2. We prove inductively over k in the call of BUILDTREE2$(T_1, .., T_k)$.

Base case:

Trivial. There is a single tree with a single node and we return the node.

Induction step:

We assume the recursive call BUILDTREE2 returns witness T of
$MWPL(value(T_1), ..., value(T_{i-1}), value(T_{i+1}), ..., value(T_{j-1}), value(T_{j+1}), ..., value(T_k), value(T_{new}))$.

Therefore, T is a witness of
$MWPL(value(T_1), ..., value(T_{i-1}), value(T_{i+1}), ..., value(T_{j-1}), value(T_{j+1}), ..., value(T_k), value(T_i) + value(T_j))$.

Subsequently, we insert nodes T_i and T_j in T according to the scheme of theorem 12.2.

Therefore, the T at line 10 is a witness of $MWPL(value(T_1), ..., value(T_k))$. □

Topic 12.5

Repeated string (LZ77)

LZ77 for repeated string

In LZ77, if a string is repeated within the sliding window on the input stream, the repeated occurrence is replaced by a reference, which is a pair of the length of the string and offset.

The references are viewed as yet another symbol on the input stream.

Example 12.15

Before encoding *ABRACADABRA* using a trie, the string will be transformed to

$$ABRACAD[4,7].$$

We run Huffman on the above string.

Multiple repetitions

Example 12.16

Consider the following input text of 16 characters.

$$abababababababab$$

We will transform the text as follows.

$$ab[14, 2]$$

Topic 12.6

DEFLATE

Practical Huffman

When we compress a file, we do not compute the frequencies for the entire file in one go.

- We compute the encoding trie of a block of bytes.
- We check if the data allows compression, if it does not we do not compress the block
- If the block is small, we use a precomputed encoding trie.

Exercise 12.7

How many bits are needed per character for 8 characters if frequencies are all equal?

DEFLATE

The Linux utility gzip uses the DEFLATE algorithm for compression, which combines Huffman encoding and the LZ77 algorithm.

DEFLATE compresses one file in blocks. Each block may be compressed in one of three modes.

- No compression
- Dynamically computed Huffman coding
- Fixed encoding

To compress multiple files, we first use a tar utility that concatenates the files into one file.

Let us look the content of the gzip file

Example 12.17

Let us consider a file "name.txt" that contains "abracadabra".

We compress the file using the following command.

```
gzip -kf name.txt
```

The command will generate file `name.txt.gz`. We may view the content of the file as follows.

```
xxd -b name.txt.gz
```

The contents are displayed in the next slide.

gzip output file format https://www.rfc-editor.org/rfc/rfc1952

```
00000000: 00011111 10001011 00001000 00001000 00000101 01001101
          |- magic number-| |-algo-| |Flags-| |---time stamp---

00000006: 00010010 01100101 00000000 00000011 01101110 01100001
          ----------------| |-XFL--| |--OS--| |---file name----

0000000c: 01101101 01100101 00101110 01110100 01111000 01110100
          -----------------------------------------------------

00000012: 00000000 01001011 01001100 00101010 01001010 01001100
          -------| |-------- DEFLATE stream -------------------

00000018: 01001110 01001100 01001001 00000100 01010010 01011100
          -----------------------------------------------------

0000001e: 00000000 01000101 11001010 11000101 01100111 00001100
          -------| |---- checksum (CRC-32)-----------| |-------

00000024: 00000000 00000000 00000000
          uncompressed filesize----|
```

Commentary: Meaning of flag bits:
0x01 FTEXT Text or Binary
0x02 FHCRC header checksum (CRC-16) is present
0x04 FEXTRA extra field is present
0x08 FNAME original file name is present
0x10 FCOMMENT comment is present
0x20 Reserved
0x40 Reserved
0x80 Reserved

The content of the compressed file "name.txt.gz" for a file "name.txt" containing "abracadabra".

A printing problem

The print in the previous slide is generated via the following command.

```
xxd -b name.txt.gz
```

It prints the bytes from MSB to LSB. So, we are seeing each byte reversed.

We need to reverse each byte to see the actual DEFLATE stream.

Example 12.18

The first byte of the DEFLATE stream is printed as `01001011`, which should be read as `11010010`.

DEFLATE stream https://www.rfc-editor.org/rfc/rfc1951

The following is the DEFLATE stream of 12 bytes from the previous slide.

1 indicates that it is a last block.

10 indicates the fixed encoding for this block (see sec 3.2.3 of RFC)

00110000+ASCII('a') = 00110000+01100001 (see sec 3.2.6 of RFC)

```
1    10   10010001   10010010 10100010   10010001   10010011 10010001
BF   BT   |---a--|   |---b--| |---r--|   |---a--|   |---c--| |---a--|

          10010100   10010001 0000001   00101   0   00111010   0000000 0
          |---d--|   |---a--| |len=3|   |dist=7|   |--LF--|   |-End-|
```

257th symbol encodes length=3 (sec 3.2.5)
0000001 encodes 257th symbol (sec 3.2.6)

After a length symbol, a distance symbol of 5 bits is expected. 00101 indicates the distance of 7 or 8 and the 0 in the following one bit indicates 7. (sec 3.2.5)

Exercise 12.8

a. Check the RFC to validate the interpretation of the bits.

b. How does gzip identify repeated patterns?

Topic 12.7

Tutorial problems

Single-bit Huffman code

Exercise 12.9

a. In a Huffman code instance, show that if there is a character with a frequency greater than $\frac{2}{5}$ then there is a codeword of length 1.

b. Show that if all frequencies are less than $\frac{1}{3}$ then there is no codeword of length 1.

Predictable text

Exercise 12.10

Suppose that there is a source that has three characters a,b,c. The output of the source cycles in the order of a,b,c followed by a again, and so on. In other words, if the last output was a b, then the next output will either be a b or a c. Each letter is equally probable. Is the Huffman code the best possible encoding? Are there any other possibilities? What would be the pros and cons of this?

Compute Huffman code tree

Exercise 12.11

Given the following frequencies, compute the Huffman code tree.

a	20
d	7
g	8
j	4
b	6
e	25
h	8
k	2
c	6
f	1
i	12
l	1

Topic 12.8

Problems

True or False

Exercise 12.12

Mark the following statements True / False and also provide justification.

1. In Huffman encoding, the code length does not depend on the frequency of occurrence of characters.

Exercise: Huffman tree for fixed encoding of DEFLATE

Example 12.19

Draw the Huffman tree for the fixed encoding used in DEFLATE.

In DEFLATE, there are 288 symbols. 0-255 are the input byte, 257-287 are the length symbols, and the 256th symbol is for the end of a block. The following is from DEFLATE RFC.

```
The Huffman codes for the two alphabets are fixed, and are not
represented explicitly in the data.  The Huffman code lengths
for the literal/length alphabet are:

          Lit Value    Bits        Codes
          ---------    ----        -----
            0 - 143     8          00110000  through 10111111
          144 - 255     9          110010000 through 111111111
          256 - 279     7          0000000   through 0010111
          280 - 287     8          11000000  through 11000111
```

Exercise: UTF-8 encoding (Final 2024)

Exercise 12.13

Consider the following Tamil word.

காதல்

1. Give the sequence of character codes to represent the sentence in unicode. Please note that some letters are combinations of characters and modifiers.
2. Let us suppose this word is stored in UTF-8 file format in a file. Give the sequence of bytes stored in the file.

Tamil character codes

Tamil[1][2]

Official Unicode Consortium code chart (PDF)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+0B8x			◌ஂ	ஃ		அ	ஆ	இ	ஈ	உ	ஊ				எ	ஏ
U+0B9x	ஐ		ஒ	ஓ	ஔ	க				ங	ச		ஜ		ஞ	ட
U+0BAx				ண	த				ந	ன	ப				ம	ய
U+0BBx	ர	ற	ல	ள	ழ	வ	ஶ	ஷ	ஸ	ஹ					◌ா	◌ி
U+0BCx	◌ீ	◌ு	◌ூ				◌ெ	◌ே	◌ை		◌ொ	◌ோ	◌ௌ	◌்		
U+0BDx	ௐ							◌ௗ								
U+0BEx							௦	௧	௨	௩	௪	௫	௬	௭	௮	௯
U+0BFx	௰	௱	௲	௳	௴	௵	௶	௷	௸	௹	௺					

UTF-8 encoding of character codes/points

UTF-8 encodes code points in one to four bytes, depending on the value of the code point. In the following table, the characters u to z are replaced by the bits of the code point, from the positions U+uvwxyz:

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0yyyzzzz			
U+0080	U+07FF	110xxxyy	10yyzzzz		
U+0800	U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+010000	U+10FFFF	11110uvv	10vvwwww	10xxxxyy	10yyzzzz

End of Lecture 12