Computer Science 1001.py

Lecture 22[†]: Text Compression

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

Lecture 21: Topics

- Intro to text compression.
- Fixed length and variable length codes.
- Prefix free codes.
- Letters' frequencies in natural languages.
- Huffman code: Preprocessing.

Lecture 22: Plan

- Huffman Code: Compression and decompression.
- Codebook compression.
- Ziv–Lempel compression: Riding on the back of repetitions.

Building the Huffman Tree: A Small Example

```
>>> small_count=char_count("this is an example of a huffman tree")
>>> sorted(small_count.items(),key=lambda x:x[1])
[('l', 1), ('o', 1), ('p', 1), ('r', 1), ('u', 1), ('x', 1),
('i', 2), ('h', 2), ('m', 2), ('n', 2), ('s', 2), ('t', 2),
('f', 3), ('a', 4), ('e', 4), (' ', 7)]
>>> small_tree=build_huffman_tree(small_count)
>>> small_tree
[[['a','e'],['i','h'],['m','n']],
[[['s','t'],['l','o'],['p','r']],[[['u','x'],'f'],' ']]]
```

It is indeed not obvious to see what is going on here.

Some parsing of this list/tree may help:

Building the Huffman Tree: A Small Example, cont.

It is indeed not obvious to see what is going on here.

Some parsing of this list/tree may help:

```
>>> len(small_tree)
2    #   as expected
>>> small_tree[0]
[['a','e'],['i','h'],['m','n']]
>>> small_tree[0][0]
['a','e']
>>> small_tree[0][0][0]
'a'    # 'a' will be encoded by 000
>>> small_tree[0][1]
[['i', 'h'], ['m', 'n']]
>>> small_tree[0][1][1]
['m', 'n']
>>> small_tree[0][1][1][1]
'n'    # 'n' will be encoded by 0111
>>> small_tree[1]
[[['s','t'],['l','o'],['p','r']],[[['u','x'],'f'],' ']]
>>> small_tree[1][1]
[[['u','x'],'f'],' ']
>>> small_tree[1][1][1]
' '    # ' ' will be encoded by 111
```

Observe that indeed more frequent letters get shorter encodings.

From Huffman Tree To Huffman Code (Recursively)

Assign the empty string to the root. Then recursively assign 0 for left subtree, 1 for right subtree.

```
def generate_code(tree_node, prefix=""):
    """ receives a tree node with embedded encoding,
        and a prefix of encodings.
        returns a dictionary where characters are
        keys and associated binary strings are
        values.
        *** assumes more than a single leaf *** de"""
    if isinstance(tree_node, str): # a leaf
        return {tree_node: prefix}
    else:
        lchild, rchild = tree_node
        res = {}
        res.update( generate_code(lchild, prefix+'0'))
        res.update( generate_code(rchild, prefix+'1'))
        #   oh, the beauty of recursion...
        return res
```

From Huffman Tree To Huffman Code: A Small Example

Assign the empty string to the root. Then recursively assign 0 for left subtree, 1 for right subtree.

```
>>> small_count=char_count("this is an example of a huffman tree")
>>> sorted(small_count.items(),key=lambda x:x[1])
[('l', 1), ('o', 1), ('p', 1), ('r', 1), ('u', 1), ('x', 1),
('i', 2), ('h', 2), ('m', 2), ('n', 2), ('s', 2), ('t', 2),
('f', 3), ('a', 4), ('e', 4), (' ', 7)]
>>> small_tree=build_huffman_tree(small_count)
>>> generate_code(small_tree)
{'a':'000',' ':'111','e':'001','f':'1101','i':'0100','h':'0101',
'm':'0110','l':'10100','o':'10101','n':'0111','p':'10110',
's':'1000','r':'10111','u':'11000','t':'1001','x':'11001'}
```

The length of encodings vary from 3 to 5.

More frequent letters are assigned shorter encodings.

Huffman Code of Wikipedia Huffman Page

```
>>> sorted(generate_code(wiki_tree).items(),key=lambda x:len(x[1]))
[(' ','1100'),('n','0000'),('t','0111'),('a','0100'),('e','1010'),
('i','1000'),('"','11010'),('<','01011'),('>','01100'),
('d','01101'),('h','00011'),('l','10111'),('r','10011'),
('/','00111'),('c','00101'),('o','11101'),('s','11110'),
('b','000100'),('f','110111'),('p','111111'),('=','101101'),
('g','100100'),('m','111000'),('u','010100'),('w','001001'),
('\n','1110010'),('.','0101010'),('0','0011001'),(':','0011011'),
('v','0011000'),('\t','0101011'),('-','1110011'),(';','0001011'),
('_','0001010'),('k','1001011'),('y','1011001'),(',','11011000'),
('2','00110100'),('x','11011011'),('%','10110000'),('1','11111001')
('&','100101011'),('6','001000111'),('8','110110100'),
('D','001000101'),('H','111110101'),('L','001000001'),

# many items removed

('T','001000100'),(')','001000000'),('3','100101010'),
('G','110110011101'),('U','001000011001'),('*','100101000011'),
('X','001000110000'),('+','110110011100 0'),('K','110110011001'),
('Q','100101000010'),('Y','001000011000 10'),
('^','001000011000111'),('~','001000011000110')]
```

The length of encodings vary from 4 to 15.

Understanding Huffman Code of Wikipedia Huffman Page

We will sort the code by length of codewords, and compare to character counts.

```
>>> wikicount=char_count(wikitext)
>>> wikitree=build_huffman_tree(wikicount)
>>> wiki_code=generate_code(wikitree)
>>> sorted_code=sorted(wiki_code.items(),key=lambda x: len(x[1]))

>>> sorted_code[:20]   # characters with 20 shortest encodings
[(' ','1100'),('n','0000'),('t','0111'),('a','0100'),('e','1010'),
('i','1000'),('"','11010'),('<','01011'),('>','01100'),
('d','01101'),('h','00011'),('l','10111'),('r','10011'),
('/','00111'),('c','00101'),('o','11101'),('s','11110'),
('b','000100'),('f','110111'),('p','111111')]

>>> sorted_count=sorted(wikicount.items(),key=lambda x: x[1])

>>> sorted_count[-20:]  # characters with 20 largest counts
[('f',1842),('m',1849),('p',2085),('h',2212),('c',2508),('/',2544),
('<',2752),('>',2752),('d',2849),('r',3328),('l',3480),('"',3540),
('o',3918),('s',3963),('n',4179),('a',5254),('t',5893),('i',6216),
('e',6824),(' ',6938)]
```

So indeed, popular characters are assigned shorter codewords.

Employing Huffman Encoding for String Compression

We go over the input string, one character at a time. For each character (non ascii characters are ignored), we access its value in the encoding dictionary. This value is a binary string, which we concatenate (`join`) to the forming output (binary string).

```
def encode_text(text, encoding_dict):
    assert isinstance(text, str)
    return ''.join(encoding_dict[ch]
            for ch in text if ord(ch)<=128)
```

Recall that strings are immutable.

Dictionary for Decompression

Given the binary string, which is the compression under a Huffman dictionary, we want to decompress it. The "holistic" way to do it employs the Huffman tree. Starting at the root, read one bit at a time. On `0` we go to the left subtree. whereas on `1` we go to the right subtree. When a leaf is reached, we append the letter at the leaf, and jump back to the root. This naturally gives rise to a fairly simple iterative procedure.

Holistic or not, a one liner code employs a dictionary which is the reverse dictionary to that produced by `generate_code`. For example, if `'x':'11001'` is an entry in the original dictionary, there will be a corresponding entry `'11001':'x'` in the reversed dictionary.

Dictionary for Decompression: Code

```
def build_decoding_dict(encoding_dict):
    """build the "reverse" of the encoding dictionary"""
    return {y:x for (x,y) in encoding_dict.items()}
    # return {y:x for x,y in encoding_dict.items()} # is OK too
```

We compute the encoding and decoding dictionaries for a small example.

```
>>> count=char_count("this is an example of a huffman tree")
>>> tree=build_huffman_tree(count)
>>> huf_code=generate_code(tree)
>>> huf_code
{'a': '000', ' ': '111', 'e': '001', 'f': '1101', 'i': '0100',
'h': '0101', 'm': '0110', 'l': '10100', 'o': '10101', 'n': '0111',
'p': '10110', 's': '1000', 'r': '10111', 'u': '11000',
't': '1001', 'x': '11001'}
>>> build_decoding_dict(huf_code)
{'0110': 'm', '0111': 'n', '11000': 'u', '11001': 'x',
'0101': 'h',, '0100': 'i', '001': 'e', '000': 'a', '111': ' ',
'1001': 't', '1101': 'f', '10111': 'r', '10101': 'o',
'10100': 'l', '10110': 'p', '1000': 's'}
```

Decoding (Decompressing): Python Code

Once we have the decoding dictionary, decoding compressed text is easy. We collect bits from the compressed text, one by one, until their concatenation is equal to one of the keys in the decoding dictionary. We find the value associated with this key, which is an ascii character, and add it to the forming output text.

```
def decode_text(bits, decoding_dict):
    prefix = ''
    result = []
    for bit in bits:
        prefix += bit
        if prefix not in decoding_dict:
            continue # continues with the next iteration of the loop
        result.append(decoding_dict[prefix])
        prefix = ''
    assert prefix == '' # must finish last codeword
    return ''.join(result)  # converts list of chars to string
```

The Full Cycle: Coding and Decoding Dictionaries

```
def purify(text):
  """ discards non ascii characters from text"""
  return ''.join(ch for ch in text
                 if ord(ch)<128)

>>> import urllib.request
>>> req = urllib.request.Request \
      (url='http://en.wikipedia.org/wiki/Huffman_coding',
         headers={'User-Agent': "Mozilla/5.0"})
>>> btext = str_to_bin(purify(urllib.request.urlopen(req).read()))
            # first removes non ascii chars, then converts to bits
>>> wikitext = btext.decode("utf8")
>>> wikicounts=char_count(wikitext)
>>> huff=build_huffman_tree(wikicounts)

>>> huf_code=generate_code(huff)
>>> huf_decode=build_decoding_dict(huf_code)
```

Comment: Wikipedia site is not happy about being accessed by programs. As of Jan. 17, 2012, access via the mechanism above is disabled (it did work until Jan. 16). You can still access it manually, then cut and pasted the text.

The Full Cycle: Encoding

Let us now compress and decompress a few sentences.

```
>>> S="""Select Alan Perlis Quotations:
(1) It is easier to write an incorrect program
    than understand a correct one.
(2) LISP programmers know the value of everything
     and the cost of nothing."""
>>> len(str_to_bin(S))
1344       # length of S (ascii) in bits
>>> T=encode_text(S,huf_code)
>>> len(T)
973      # 72% of original
```

The Full Cycle: Compressed String

```
>>> T    # in case you are curious. Enjoy!
'111000100100111010101011110100010101111100111000001101110100000011
0011111010001010100111011110001111011000011111111001011000111010111
0100011110001110100001111010010011110001111110110111111001111110111
0110010110001010111110010001111011001010010011110100010101001111000
1111110111000010001001110000111101011000100000011001000000000101111
0110011100111010001010111110011111100111101011001100110100110111 1
1100011100110011001100011100011010000001100011000000001101101010011
1111001110100000001101110001001100001011110110011100111010001010111
1100111010000101000111011110001111110110100111110111110111011000001
0101110110001010010011101111101000110011111110011111010110011001101
0011011111011110101001111110110010010100000111010010001100011100011
1010110000111000100101110110001010110011101111001110010100011100101
0100111011001011100011100000000110011110001110011001100110001000000
0110111000111000111010110000101111011111100111110011101111001110000 0
011101011100011100000000110010011101'
```

The Full Cycle: Decoding

```
>>> U=decode_text(T,huf_decode)
>>> U
'Select Alan Perlis Quotations:\n(1) It is easier to write an incor
than understand a correct one.\n(2) LISP programmers know the value
and the cost of nothing.'
>>> print(U)
Select Alan Perlis Quotations:
(1) It is easier to write an incorrect program
    than understand a correct one.
(2) LISP programmers know the value of everything
     and the cost of nothing.
```

The Full Cycle with Larger Texts

We could try compressing larger size texts. For example, Project Gutenberg lets you download over 33,000 free e-books. We will download two books, build Huffman dictionaries, then compress each of them using its own dictionary as well as the other dictionary. We start with "The Odyssey", by Homer (~700 BC).

```
>>> btext = urllib.request.urlopen(
'http://www.gutenberg.org/ebooks/3160.txt.utf8').read()
>>> homer_text = btext.decode('utf-8')
>>> len(str_to_bin(homer_text))
5110805   # length (ascii) in bits
>>> count=char_count(homer_text)
>>> tree=build_huffman_tree(count)
>>> homer_encode=generate_code(tree)
>>> homer_decode=build_decoding_dict(homer_encode)
```

The Full Cycle with Larger Texts (2)

Next, we turn to "Romeo and Juliet", by William Shakespeare (1564-1616).

```
>>> btext = urllib.request.urlopen(
'http://www.gutenberg.org/ebooks/1112.txt.utf8').read()
>>> william_text = str_to_bin(btext.decode('utf-8')+asci)
            # remove non ascii chars
>>> len(str_to_bin(william_text))
1184260        # length (ascii) in bits
>>> count=char_count(william_text)
>>> tree=build_huffman_tree(count)
>>> william_encode=generate_code(tree)
>>> william_decode=build_decoding_dict(william_encode)
```

The Full Cycle with Larger Texts (3)

We are finally ready to compress and decompress

```
>>> len(str_to_bin(homer_text))
5110805
>>> Odd1=encode_text(homer_text,homer_encode)
>>> len(Odd1)
3389460  # 66% of original
>>> decode_text(Odd1,homer_decode)[0:21]
'The Project Gutenberg'
>>> decode_text(Odd1,homer_decode)[30:77]
' The Odyssey of Homer, trans. by Alexander Pope'

>>> Odd2=encode_text(homer_text,william_encode)
Traceback (most recent call last):
  File "<pyshell#169>", line 1, in <module>
    Odd2=encode_text(homer_text,william_encode)
  File "/Users/admin/Documents/IntroCS_Course/Compression/compress.
    return ''.join(encoding_dict[ch] for ch in text if ord(ch)<=128
  File "/Users/admin/Documents/IntroCS_Course/Compression/compress.
    return ''.join(encoding_dict[ch] for ch in text if ord(ch)<=128
KeyError: '&'
```

Despite this unpleasant failure, we do hope you will consider using Project Gutenberg. Or even, god forbid, read an old fashion, hard copy, version of one of those e-books.

Source of Problem

We got an error message when trying to encode The Odyssey using the Romeo and Juliet induced dictionary, `william_encode`. Let us try to understand this problem:

```
>>> len(william_encode.items())
88
>>> len(homer_encode.items())
91
>>> "&" in william_encode
False
>>> "&" in homer_encode
True
>>> homer_encode["&"]
'01000000100110100'
```

The Odyssey's dictionary and Romeo and Juliet's dictionary are of different sizes, and some characters, like `"&"`, appear in the first but not in the second. When the encoding procedure encounters such character, it reports a `KeyError: '&'`.

Fixing this Problem

We could meticulously go over the missing characters and add them to the dictionaries. But this approach is rather tedious. Instead, we will form a new string, which will be the concatenation of all strings in the ascii code. We will concatenate this string, termed `asci` (since `ascii` is already taken) to the original text, thus making sure every ascii character is represented.

This will change the counts, but the change is so slight we will hardly notice it (and yes, we do know that 33 of the ascii encoded characters are obsolete).

```
>>> asci = ''.join(chr(i) for i in range(128))
>>> asci[:16]
'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'
>>> asci[17:32]
'\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f'
>>> asci[32:90]
' !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXY'
>>> asci[90:]
'Z[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~\x7f'
```

Fixing this Problem (Homer)

```
def process_homer():
  import urllib.request
  btext = urllib.request.urlopen(
    'http://www.gutenberg.org/ebooks/3160.txt.utf8').read()
  asci ='' . join ( chr(i) for i in range (128))
  homertext = purify(btext.decode('utf8')+asci)
  print("homer length in bits", len(homertext)*7)
  homer_count=char_count(homertext)
  homer_tree=build_huffman_tree(homer_count)
  homer_encode_dict=generate_code(homer_tree)
  homer_decode_dict=build_decoding_dict(homer_encode_dict)
  homer_encoded_text=encode_text(homertext,homer_encode_dict)
  print("homertext encoded length in bits", len(homer_encoded_text)
  print("compression ratio",len(homer_encoded_text)/(len(homertext)
  homer_decoded_text=decode_text(homer_encoded_text,
                homer_decode_dict)

  return homertext, homer_decoded_text,\
         homer_encode_dict,homer_decode_dict
```

Just bundle everything together so it can be run by a one line invocation of a zero arguments function.

Fixing this Problem (Shakespeare)

```
def process_william():
  import urllib.request
  btext = urllib.request.urlopen(
    'http://www.gutenberg.org/ebooks/1112.txt.utf8').read()
  asci ='' . join ( chr(i) for i in range (128))
  williamtext = purify(btext.decode('utf8')+asci)
      # remove non ascii chars
  print("william length in bits", len(williamtext)*7)
  william_count=char_count(williamtext)
  william_tree=build_huffman_tree(william_count)
  william_encode_dict=generate_code(william_tree)
  william_decode_dict=build_decoding_dict(william_encode_dict)
  william_encoded_text=encode_text(williamtext,william_encode_dict)
  print("williamtext encoded length in bits",
   len(william_encoded_text))
  print("compression ratio",len(
    william_encoded_text)/(len(williamtext)*7))
  william_decoded_text=decode_text(william_encoded_text,
    william_decode_dict)

  return williamtext, william_decoded_text,\
         william_encode_dict,william_decode_dict
```

Just bundle everything together, like before.

Sanity Check

```
>>> h_original,h_new,h_encode,h_decode=proceess_homer()
homer length in bits 5111701
homertext encoded length in bits 3391176
compression ratio 0.6634143898479196
>>> h_original==h_new
True
>>>
>>> w_original,w_new,w_encode,w_decode=proceess_william()
william length in bits 1185156
williamtext encoded length in bits 781439
compression ratio 0.6593553928765495
>>> w_original==w_new
True
```

Cross Compression Performance

```
>>> homer_using_homer=encode_text(h_original,h_encode)
>>> len(homer_using_homer)
3391176

>>> homer_using_william=encode_text(h_original,w_encode)
>>> len(homer_using_william)
3476021

>>> len(homer_using_william)/len(homer_using_homer)
1.025019344321851 8  # just 2 percent more

>>> decode_text(homer_using_william,w_decode)==h_original
True
```

More Cross Compression Performance

```
>> william_using_william=encode_text(w_original,w_encode)
>>> len(william_using_william)
781439

>>> william_using_homer=encode_text(w_original,h_encode)
>>> len(william_using_homer)
807061

>>> len(william_using_homer)/len(william_using_william)
1.032788278719132   # just 3 percent more

>>> decode_text(william_using_homer,h_decode)==w_original
True

>>> decode_text(william_using_homer,w_decode)==w_original
False
>>> decode_text(william_using_homer,h_decode)[:50]
'This Etext file is presented by Project Gutenberg,'
>>> decode_text(william_using_homer,w_decode)[:50]
's lh k  \rac od  onod rT\r t e Cwhe\r h)n -\rOeloJa\nea'
   ## garbage in, garbage out
```

Huffman Code: Time and Space Complexity

Let $\mathtt{S}$ be an n character long corpus.
Let T be an m character long text.

- Counting letters in corpus: $O(n)$ steps.
- Building Huffman tree: $O(|\Sigma|)$ insertions in a priority queue. (each insertion costs more than a constant!)
- Building Huffman encoding table from the tree: $O(|\Sigma| \log |\Sigma|)$ if tree traversal implemented efficiently.
- Building Huffman decoding table from the encoding table: $O(|\Sigma| \log |\Sigma|)$ (the log factor comes from going over and copying $|\Sigma| \log |\Sigma|$ many bits).

- Encoding and decoding text: $O((\ell + n) \log |\Sigma|)$, where ℓ equals length of compressed string, and assuming hash query takes $O(1)$ steps.
- Space used by hash tables: $O(|\Sigma| \log |\Sigma|)$.

Huffman Code: An Interesting Anecdote

Huffman proposed the Huffman code while he was a graduate student at MIT, as part of a term paper for Robert Fano's class. In Fano's words,

"... by 1950, I started teaching a graduate subject on information theory, and one of the students was named Dave Huffman, who wrote a term paper. I had given a number of possible topics. One of them was that while I developed the form of encoding, that did not assure that the coding would be optimum. Shannon, who at that time was at Bell Laboratories, was not sure. So I raised the question. I said, "It would be nice to know an optimum way of encoding." All of which Huffman developed as a term paper that he published, of course."

(1952 paper, "A Method for the Construction of Minimum Redundancy Codes")

Compressing Text Beyond Huffman

Huffman code is optimal with respect to a known distribution of single characters.

One possible way to improve upon it is to consider pairs or triplets of characters instead of single ones.
It may be somewhat harder to collect the relevant statistics, and the size of the encoding/decoding dictionaries will be substantially larger.
However, since dictionaries employ hashing, encoding and decoding times will hardly be effected.

Distribution of Single Words

We already saw that the distribution of single letters/characters in standard English text is far from uniform. In a similar manner, the distribution of single words in standard English text is far from uniform. It was claimed that "The first 25 make up about one-third of all printed material in English. The first 100 make up about one-half of all written material, and the first 300 make up about sixty-five percent of all written material in English." (source: this site).

The First Hundred

1. the	21. at	41. there	61. some	81. my
2. of	22. be	42. use	62. her	82. than
3. and	23. this	43. an	63. would	83. first
4. a	24. have	44. each	64. make	84. water
5. to	25. from	45. which	65. like	85. been
6. in	26. or	46. she	66. him	86. call
7. is	27. one	47. do	67. into	87. who
8. you	28. had	48. how	68. time	88. oil
9. that	29. by	49. their	69. has	89. its
10. it	30. word	50. if	70. look	90. now
11. he	31. but	51. will	71. two	91. find
12. was	32. not	52. up	72. more	92. long
13. for	33. what	53. other	73. write	93. down
14. on	34. all	54. about	74. go	94. day
15. are	35. were	55. out	75. see	95. did
16. as	36. we	56. many	76. number	96. get
17. with	37. when	57. then	77. no	97. come
18. his	38. your	58. them	78. way	98. made
19. they	39. can	59. these	79. could	99. may
20. I	40. said	60. so	80. people	100. part

Compressing Using Codebooks[‡]

- Preprocessing: Identify the 128 most frequent words of length greater than 1 in a suitable corpus.
- Construct a codebook with 256 entries: 128 Ascii characters, and the 128 frequent words.
- Codebook entries are encoded by a fixed length, 8 bit codeword.
- Encoding procedure: Read the text and parse it to words. If the next word is in the codebook, encode it by its 8 bit code. Otherwise, encode it letter by letter.

[‡]Codebooks are also called dictionaries in many texts. I chose to stick with codebooks in order not to cause a confusion with Python's dictionary, which is a hash table.

Compressing Using Codebooks: Compression Estimate

- Assume the average length of a frequent word is 3 characters, and further that the frequent words make up 50% of a typical text.
- Then for these 50%, we use 8 bits instead of 21 bits (three ascii characters). For the rest, we use 8 bits instead of 7.
- So on the average, 21 original bits are encoded by $(8+24)/2=16$ bits. This is 76% of the original.
- This simple scheme can be further improved by employing Huffman encoding on the codebook entries. But some care need be exercised in computing frequencies: Overlapping letters in frequent words should not be counted twice.
- Both the simple and the improved schemes could make good homework problems.

Compressing Text Beyond Huffman (2)

A completely different approach was proposed by Yaacov Ziv and Abraham Lempel in a seminal 1977 paper ("A Universal Algorithm for Sequential Data Compression", IEEE transactions on Information Theory).

Their algorithm went through several modifications and adjustments. The one used most these days is by Terry Welch, in 1984, and known today as LZW compression.

Unlike Huffman, all variants of LZ compression do not assume any knowledge of character distribution. The algorithm finds redundancies in texts using a different strategy.

We will go through this important compression algorithm in detail.

Huffman vs. Ziv Lempel: Basic Difference

Both Huffman and the codebook compressions are static. They compute frequencies, based on some standard corpus. These frequencies are used to build compression and decompression dictionaries, which are subsequently employed to compress and decompress any future text.

The statistics (or derived dictionaries) are shared by both sides before communication starts.

By way of contrast, Ziv-Lempel compression(s) are adaptive. There is no precomputed statistics. The basic redundancies employed here are repetitions, which are quite frequent in human generated texts.

There is no need to share any data before transmission commences. In a sense, the decompressed text will serve as the basis for subsequent decompression.

Repetitions in Human Generated Text

This following text (found on a public toilet's door in a Canadian Rockies campground, some 27 yrs ago) is 132 characters long. A quick inspection reveals a large number of small repetitions.

"We are mountaineers with hairy ears. We sleep in caves and ditches. We wipe our ass with broken glass. And lough, because it itches."

A manual count exposes one length 6 repeat (the dot is an equal rights character), three length 3 repeats (space is not only an equal right character, but also very popular), one appearing trice, and six length 2 repeats, color coded below:

"We are mountaineers with hairy ears. We sleep in caves and ditches. We wipe our ass with broken glass. And lough, because it itches."

Ziv-Lempel: Riding on Text Repetitions

The basic idea of the Ziv-Lempel algorithm is to "take advantage" of repetitions in order to produce a shorter encoding of the text. Let `T` be an n character long text. In Python's spirit, we will think of it as `T[0]T[1]...T[n-1]`.

Suppose we have a k long repetition ($k > 0$) at positions $j, p = j + m$ ($m > 0$): `T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1]`.

Basic Idea: Instead of coding `T[p]T[p+1]...T[p+k-1]` character by character, we can fully specify it by identifying the starting point of the first occurance, j, and the length of the repetition, k.

Ziv-Lempel: How to Represent Repetitions

Suppose we have a k long repetition ($k > 0$) at positions j, p ($j < p$): `T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1]`.

There are two natural ways to represent the starting point, j. Either by j itself, or as an offset from the second occurance, namely m, where $m = p - j$.

The first option requires that we keep a full prefix of the text both while compressing and when decompressing. Since the text can be very long, this is not desirable. In addition, for a long text, representing j itself may take a large number of bits.

Ziv-Lempel: How to Represent Repetitions, cont.

Suppose we have a k long repetition ($k > 0$) at positions j, p ($j < p$): `T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1]`.

There are two natural ways to represent the starting point, j. Either by j itself, or as an offset from the second occurance, namely m, where $m = p - j$.

Instead of keeping all the text in memory, Ziv-Lempel advocates keeping only a bounded part of it. The standard recommendation is to keep the 4096 most recent characters.

This choice has the disadvantage that repeats below the horizon, *i.e.* earlier than 4096 most recent characters, will not be detected. It has the advantage that m can be represented succinctly (12 bits for 4096 size window).