Lecture 21: Ziv–Lempel Compression

Instructors: Daniel Deutch, Amir Rubinstein
Teaching Assistants: Amir Gilad, Michal Kleinbort

Founding Instructor: Benny Chor

School of Computer Science
Tel-Aviv University, Fall Semester, 2017
http://tau-cs1001-py.wikidot.com
Lecture 20: Topics

- Huffman code and its application in text compression.
- Codebook compression (using word frequencies).
Lecture 21: Plan

- Ziv–Lempel text compression.
  - Basic idea: Exploiting text repetitions.
  - Lossless compression
  - Basis of zip, winzip, Tar, and numerous other commercial compression packages.
Huffman Code: One Chart to Capture It All*

Flow diagram of Huffman compression process

- corpus
- char_count
- Frequencies dictionary \{char:count\}
- build_huffman_list
- Huffman tree (as list)
- generate_code
- Text
- Compressed text
- Compressed text
- Reversed codes dictionary \{binary string : char\}
- decompress
- compress
- build_decoding_dict

Source: Anonymous Researcher.

*with apologies to John Ronald Reuel Tolkien (1892–1973)
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 either shared by both sides before communication starts, or have to be explicitly transmitted as part of the communication.

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 here to share any data before transmission commences.
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 position $j$ and at position $p = j + m$ ($m > 0$), namely:


**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 occurrence, $j$, and the length of the repetition, $k$. 
Suppose we have a $k$ long repetition ($k > 0$) at positions $j, p$ ($j < p$): $T[j]T[j+1] \ldots T[j+k-1] = T[p]T[p+1] \ldots T[p+k-1]$.

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 long text, representing $j$ itself may take a large number of bits.
Suppose we have a \( k \) long repetition \( (k > 0) \) at positions \( j, p \) \( (j < p) \):
\[
\]

There are two natural ways to represent the starting point, \( j \). Either by \( j \) itself, or as an offset from the second occurrence, 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 4096-1 most recent characters.

This choice has the disadvantage that repeats below the horizon, i.e. earlier than 4096-1 most recent characters, will not be detected. It has the advantage that \( m \) can be represented succinctly (12 bits for 4096-1 size window).
Using a Finite Window, cont.

Another reason to use a finite window is its adaptivity: If the source of the text changes its properties, this will be reckoned with automatically when the window slides fully to be within the new parameters.

Interestingly, Ziv and Lempel have proved that their algorithm achieves optimal compression with respect to texts produced by finite state Markov models (not seen in our course. This is an asymptotic result, and will not be shown here).

A finite state Markov model is not a good model for human generated text. Yet, years of practice have shown that the Ziv-Lempel is effective in compressing human generated text. In fact, most of you are routinely using it in standard text compression software (zip, winzip, etc.).
High Level LZ Compression

To encode the string \( T[0]T[1] \ldots T[n-1] \), using a sliding window with \( W \) characters:

1. Loop over position in \( T \), starting with the index \( p=0 \)
2. While the text was not exhausted
   2.1 Find largest match for \( T[p::] \) starting at \( T[p-m::] \) for some \( 0 < m \leq W \).
   2.2 Suppose this match is of length \( k \), \( T[p-m:p-m+k] \), \( 2 \leq k \).
   2.3 Output \( m, k \).
3. Update the text location: \( p = p+k \).

Notice that the overlapping segment \( T[p-m:p-m+k] \) may go beyond \( T[p] \) (into the “future”).

Remark: Recording repetitions of length 1 is wasteful in terms of bits used vs. bits saved. Thus the restriction \( 2 \leq k \).
LZ Compression: Some Details

We have already mentioned that the size of the window, $W$, is typically restricted to 4,095. Thus the offset, $m$, can be represented by a fixed length, 12 bits number. The length of the match, $k$ is also limited, typically to 31. So the length, $k$, can be represented by a fixed length, 5 bits number.

Finding a maximum match quickly is also an important issue, determining the efficiency of the compression algorithm. Hashing and a trie data structure (to be discussed in the data structures course) are two possible approaches to speed up the computation. In both cases, we should be able to efficiently update it before it becomes obsolete.
We present a simple iterative procedure for the task, which does not employ any sophisticated data structures. Its performance (both in terms of running time, and of compression ratio) will not be as good as the optimized, commercial packages. But unlike the packages, you will understand what goes on here.

Text to ASCII (reminder):

```python
def str_to_ascii(text):
    """ Gets rid of non ascii characters in text""
    return ''.join(ch for ch in text if ord(ch)<128)
```
Maximum Match

Our first task is locating the maximum matches.

The function `maxmatch` returns the offset and the length of a maximum length match $T[p:p+k] == T[p-m:p-m+k]$ within prescribed window size backwards and maximum match size.

The function `maxmatch(T, p, w, max_length)` has four arguments:

- $T$, the text (a string).
- $p$, an index within the text.
- $w$, a size of window within which matches are sought.
- `max_length`, the maximal length of a match that is sought

The last two arguments will have the default values $2^{12} - 1, 2^5 - 1$, respectively. With these default values, the offset can be encoded using 12 bits, and a match length can be encoded using 5 bits.
```python
def maxmatch(T, p, w=2**12 - 1, max_length=2**5 - 1):
    """ finds a maximum match of length k <= 2**5 - 1 in a
    w long window, T[p:p+k] with T[p-m:p-m+k].
    Returns m (offset) and k (match length) """
    assert isinstance(T, str)
    n = len(T)
    maxmatch = 0
    offset = 0
    for m in range(1, min(p+1, w)):
        current_length = 0
        for k in range(0, min(max_length, n-p)):
            # at this point, T[p-m:p-m+k] == T[p:p+k]
            if T[p-m+k] != T[p+k]:
                break
            else:
                current_length = k + 1
        if maxmatch < current_length:
            maxmatch = current_length
            offset = m
    return offset, maxmatch

# returned offset is smallest one (closest to p) among
# all max matches (m starts at 1)
```

Maximum Match: Run Time

For any location, $p$, this function takes up to $w \cdot \text{max\_length}$ many operations in the worst case.

For the default parameters, this is $2^{12} \cdot 2^5 = 2^{17}$ per one position, $p$. Running $\text{maxmatch}(T,p)$ over all text locations will this take up to $2^{17}$ times the length of $T$ operations.

This is the major consumer of time in our compression procedure.

We will later sketch (and only sketch) ideas for improving this bottleneck.
Maximum Match: A Few Experiments

```python
>>> s="aaabbbbaabbbbaaa"
>>> lst=[maxmatch(s,i) for i in range(len(s))]
>>> print(lst)
[(0, 0), (1, 2), (1, 1), (0, 0), (1, 2), (1, 1), (6, 9), (6, 8), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (1, 2), (1, 1)]

>>> s='how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood'
>>> lst=[maxmatch(s,i) for i in range(len(s))]
>>> print(lst)
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (7, 1), (5, 1), (7, 1), (9, 1), (1, 1), (0, 0), (5, 3), (5, 2), (4, 1), (11, 1), (0, 0), (6, 2), (6, 1), (0, 0), (14, 1), (0, 0), (15, 6), (15, 5), (15, 4), (15, 3), (9, 2), (5, 1), (23, 2), (9, 1), (26, 2), (3, 1), (0, 0), (6, 7), (6, 6), (6, 5), (6, 4), (6, 3), (6, 2), (6, 1), (0, 0), (0, 0), (24, 16), (24, 15), (24, 14), (24, 13), (24, 12), (24, 11), (24, 10), (24, 9), (24, 8), (18, 7), (18, 6), (18, 5), (18, 4), (18, 3), (18, 2), (45, 7), (45, 6), (45, 5), (45, 4), (45, 3), (12, 10), (12, 9), (12, 8), (12, 7), (12, 6), (12, 5), (12, 4), (23, 6), (23, 5), (23, 4), (23, 3), (11, 2), (5, 1), (0, 0), (77, 2), (18, 6), (18, 5), (18, 4), (18, 3), (18, 2), (15, 1), (6, 5), (6, 4), (6, 3), (6, 2), (6, 1), (12, 2), (18, 3), (18, 2), (5, 1)]
```
Our Version of LZ77 Compression

Instead of producing a string composed of bits right away, we decompose this task to two: An intermediate output (in an intermediate format), and then a final output. The intermediate format will be easier to understand (and to debug, if needed).

At the first stage, we produce a list, which either encodes single characters (in case of a repeat of length smaller than 2), or a pair $[m, k]$, where $m$ is an offset, and $k$ is a match length. The default bounds on these numbers are $0 < m < 2^{12}$ (12 bits to describe) and $1 < k < 2^5$ (5 bits to describe).

The algorithm scans the input text, character by character. At each position, $p$, it invokes maxmatch(text, $p$). If the returned match value, $k$, is 0 or 1, the current character, text[$p$], is appended to the list. Otherwise, the pair $[m, k]$ is appended.
def lz77_compress(text, w=2**12 - 1, max_length=2**5 - 1):
    """LZ77 compression of an ascii text. Produces a list comprising of either ascii character or by a pair [m,k] where m is an offset and k is a match (both are non negative integers)""
    result = []
    n = len(text)
    p = 0
    while p < n:
        m, k = maxmatch(text, p, w, max_length)
        if k < 2:
            result.append(text[p])  # char, as opposed to a pair
            p += 1
        else:
            result.append([m, k])  # two or more chars in match
            p += k
    return(result)  # produces a list composed of chars and pairs
Intermediate Format LZ77 DeCompression: Python Code

Of course, compression with no decompression is of little use.

```python
def lz77_decompress(compressed, w=2**12 - 1, max_length=2**5 - 1):
    """LZ77 decompression from intermediate format to ascii text""
    result = []
    n = len(compressed)
    p = 0
    while p < n:
        if type(compressed[p]) == str:  # char, as opposed to a pair
            result.append(compressed[p])
            p += 1
        else:
            m, k = compressed[p]
            p += 1
            for i in range(0, k):
                # append k times to result;
                result.append(result[-m])
                # fixed offset m "to the left", as result itself grows
    return lst_to_string(result)

def lst_to_string(lst):
    """converting a list of chars to a string""
    return ''.join(ch for ch in lst)
```
Intermediate Format LZ77 Compression and DeCompression: A Small Example

```python
>>> s="abc"*20
>>> 7*len(s)
420
>>> inter=lz77_compress(s)
>>> s
'abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc'
>>> inter
['a', 'b', 'c', [3, 31], [3, 26]]
```

So indeed, compression could pass the current location: We can have \( j+k-1>p \) in \( T[j]T[j+1]...T[j+k-1]=T[p]T[p+1]...T[p+k-1] \).
Intermediate Format LZ77 Compression and DeCompression: Another Small Example

```python
>>> s = """"how much wood would the wood chuck chuck if
the wood chuck would chuck wood should could hood"""

>>> r = lz77_compress(s)
>>> r
['h', 'o', 'w', ' ', 'm', 'u', 'c', 'h', ' ', 'w', 'o', 'o', 'd', [5, 3], 'u', 'l', [6, 2], 't', 'h', 'e', ' ', 'w', 'o', 'o', 'd', ' ', 's', [77, 2], [18, 5], [6, 5], [12, 2], [18, 2]]

>>> t = lz77_decompress(r)
>>> t
'how much wood would the wood chuck chuck if
the wood chuck would chuck wood should could hood'
```

```python
>>> print(t)
how much wood would the wood chuck chuck if
the wood chuck would chuck wood should could hood
```
Performance Time for Compression

The major consumer of time here is the `maxmatch(text,p,w,max_length)` procedure. It is invoked for every location \( p \) that was not skipped over. For locations far enough from the boundary, and for the default parameters, this is \( 2^{12} \cdot 2^5 = 2^{17} \) operations per invocation.

Let us assume that one half of the text is skipped over because of repetitions (a realistic assumption in human generated text). If \( n \) denotes the text length, compression will require \( 2^{17} n/2 = 2^{16} n \) operations. For \( n = 2^{20} \), this estimate gives \( 2^{16} \cdot 2^{20} = 2^{36} \approx 10^{11} \) operations. On my iMac (5 years old desktop), this task took 97.5 seconds.

```python
>>> cholera = open("Vibrio_cholerae.txt").read() # proteome of Cholera
>>> len(cholera)
3040279
>>> elapsed("lz77_compress(cholera[:2**16])")
97.504324 # over 1.5 minutes for compression of first 2**16 chars
```
Performance Time for Compression

```python
>>> import timeit
>>> cholera = open("Vibrio_cholerae.txt").read()  # proteome of Cholera
>>> len(cholera)
3040279
>>> elapsed("lz77_compress(cholera[:2**16])")
97.504324
>>> len(cholera)/2**16
46.39097595214844
>>> 46.39097595214844*97.504324
4523.320749914489
>>> 4523.320749914489/60
75.38867916524148
```

So compressing the complete proteome of Cholera would take approx. an hour and a quarter. This obviously is unacceptable in real life contexts.

We’ll briefly discuss approaches to speed up the compression phase later in class. We will not implement them, though.
Performance: Time for Decompression

Unlike compression, decompression involves just fetching elements from either the input list or from the forming text. The run time will be $O(\ell)$, where $\ell$ is not the input length but the output length.

```python
>>> import timeit
>>> cholera=open("Vibrio_cholerae.txt").read() # proteome of Cholera
>>> inter=lz77_compress(cholera[:2**16])
>>> inter[110:120]
['N', 'N', 'G', 'S', 'G', 'V', 'L', [8, 2], 'A', 'D']
>>> elapsed("lz77_decompress(inter)")
0.04022099999997408
```

So indeed, decompression is vastly faster than compression.

In practical terms, this asymmetry means a distributor of media or software, presumably with strong computing resources, can do the compression off-line. The “end user”, with presumably weaker computational resources, can efficiently perform decompression.
Performance: Measuring Compression Ratio

To distinguish between a single character and an \([m,k]\) entry, a '0' will be placed before a single ascii character, while a '1' will be placed before an \([m,k]\) entry.

Using the default parameters, \(m\) is 12 bits long, while \(k\) is 5 bits long. So a “single ascii character” entry is 8 bits long, while an \([m,k]\) entry is \(1+12+5=18\) bits long. If the number of ascii entries in the intermediate output is \(l_1\), and the number of entries of the second type is \(l_2\), then the final output will be of length \(8l_1 + 18l_2\).

\[
\begin{align*}
\text{len}(\text{cholera [:2**16]}) \times 7 & = 458752 \\
\text{l1} &= \text{sum}(1 \text{ for } x \text{ in inter if type(x)==str)} \\
\text{l2} &= \text{sum}(1 \text{ for } x \text{ in inter if type(x)!=str)} \\
\text{l1} &= 1709 \\
\text{l2} &= 21379 \\
8\text{l1}+18\text{l2}; (8\text{l1}+18\text{l2})/458752 & = 398494 \quad 0.8686479840959821 \quad # \text{86\% of original. Not so impressive!} \\
\text{Cholera proteome is not human generated and has few repetitions.}
\end{align*}
\]
Cholera Compression Ratio: Ziv-Lempel vs. Huffman

We saw that the Ziv-Lempel algorithm compresses the Cholera proteome to only 86.8% of its original size. The Cholera proteome is (to the best of our knowledge) not man made. So some properties common in human generated text, like repetitions, are not very frequent. Thus the Ziv-Lempel compression ratio is not very impressive here.

But, since most of the text is over the amino acid alphabet, which has just 20 characters. The vast majority of the characters in the text can thus be encoded using under 5 bits on average. This indicates that maybe Huffman could do better.
def process_cholera():
    cholera = open("Vibrio_cholerae_B33.txt").read()
    print("cholera length in bits", len(cholera)*7)
    cholera_count = char_count(cholera)
    cholera_list = build_huffman_list(cholera_count)
    cholera_encode_dict = generate_code(cholera_list)
    cholera_decode_dict = build_decoding_dict(cholera_encode_dict)
    cholera_compressed = compress(cholera, cholera_encode_dict)
    print("compressed cholera text length in bits",
          len(cholera_encoded_text))
    print("compression ratio",
          len(cholera_compressed)/(len(cholera)*7))
    cholera_decoded_text = decode_text(cholera_encoded_text,
                                        cholera_decode_dict)
    return cholera, cholera_decoded_text,
           cholera_encode_dict, cholera_decode_dict
Cholera Compression by Huffman: Execution

```python
>>> cholera_text, cholera_decoded_text,
    cholera_encode_dict, cholera_decode_dict = process_cholera()

cholera length in bits 21281953
compressed cholera_text length in bits 15650235
compression ratio 0.7353758839707991

>>> cholera_text == cholera_decoded_text
True  # sanity check

>>> count = char_count(cholera)
>>> scount = sorted(count.items(), key=lambda x:x[1])
>>> scount[-10:]  # 10 most popular chars
[(‘D’, 119431), (‘Q’, 121244), (‘T’, 122391), (‘I’, 141045),
 (‘E’, 147650), (‘S’, 148305), (‘G’, 156634), (‘V’, 172055),
 (‘A’, 217096), (‘L’, 252522)]

>>> cdict = cholera_encode_dict
>>> sdict = sorted(cdict.items(), key=lambda x:len(x[1]))

>>> sdict[:10]  # 10 shortest encoding
 (‘G’, ’0101’), (‘I’, ’0001’), (‘S’, ’0011’), (‘D’, ’11010’),
 (‘F’, ’10001’), (‘N’, ’01111’)]
```
Intermediate Format LZ77 Compression: A Small Improvement

To encode a repetition using the default parameters, it takes one bit to indicate this is a pair, 12 bits for $m$, and 5 bits for $k$. All by all, such encoding is $1+12+5=18$ bits long for all $k, k < 2^5$. A “single ascii character” entry is 8 bits long, so two single characters take 16 bits.

We conclude that encoding a two character repeat takes 18 bits, which is always longer than encoding the two separately. Furthermore, once in a while the second character will be the beginning of a different, longer repeat.

In addition, the existing decompression code works just fine for both the “older” and the “improved” versions of compression.
Intermediate Format LZ77 Compression Improvement: Code

```python
def lz77_compress2(text, w=2**12 - 1, max_length=2**5 - 1):
    """ LZ77 compression of an ascii text. Produces a list comprising of either ascii character or by a pair [m,k] where m is an offset and k is a match (both are non negative integers)""
    result = []
    n=len(text)
    p=0
    while p<n:
        m,k=maxmatch(text,p,w,max_length)
        if k<3:  # modified from k<2
            result.append(text[p])  # a single char
            p+=1
        else:
            result.append([m,k])  # two or more chars in match
            p+=k
    return(result)  # produces a list composed of chars and pairs
```

Testing the Improvement

```python
>>> cholera = open("Vibrio_cholerae_B33.txt").read()
>>> inter = lz77_compress(cholera[:2**16])
>>> l1 = sum(1 for x in inter if type(x) == str)
>>> l2 = sum(1 for x in inter if type(x) != str)
>>> 8*l1 + 18*l2
398494
```

```python
>>> inter2 = lz77_compress2(cholera[:2**16])
>>> l1 = sum(1 for x in inter2 if type(x) == str)
>>> l2 = sum(1 for x in inter2 if type(x) != str)
>>> 8*l1 + 18*l2
359682
```

```python
>>> 359682/398494
0.9026033014298835  # approx. 10% better!
```

```python
>>> 2**16*7
458752
```

```python
>>> 359682/458752
0.7840445382254464
```

# compression ratio of improved (w.r.t. original text)
There and Back Again

Let us complete our LZ77 tour by going from the intermediate format to the compressed string of bits, and vice versa. The code for both transformations is simple and efficient.
def inter_to_bin(lst, w=2**12-1, max_length=2**5-1):
    """ converts intermediate format compressed list to a string of bits"""
    offset_width = math.ceil(math.log(w, 2))
    match_width = math.ceil(math.log(max_length, 2))
    result = []
    for elem in lst:
        if type(elem) == str:
            result.append("0")
            result.append('{:07b}'.format(ord(elem)))
        elif type(elem) == list:
            result.append("1")
            m, k = elem
            result.append('{num:0{width}b}'.format(num=m, width=offset_width))
            result.append('{num:0{width}b}'.format(num=k, width=match_width))

    return ".join(ch for ch in result)

Don’t forget to import math for the logarithm.
def bin_to_inter(compressed, w=2**12 - 1, max_length=2**5 - 1):
    """ converts a compressed string of bits to intermediate compressed format ""
    offset_width = math.ceil(math.log(w, 2))
    match_width = math.ceil(math.log(max_length, 2))
    result = []
    n = len(compressed)
    p = 0
    while p < n:
        if compressed[p] == "0":  # single ascii char
            p += 1
            char = chr(int(compressed[p:p+7], 2))
            result.append(char)
            p += 7
        elif compressed[p] == "1":  # repeat of length > 2
            p += 1
            m = int(compressed[p:p+offset_width], 2)
            p += offset_width
            k = int(compressed[p:p+match_width], 2)
            p += match_width
            result.append([m, k])
    return result

Don’t forget to import math for the logarithm.
There and Back Again: The Compress/Decompress Cycle

Don't forget to import math for the logarithm.

```python
>>> text = """how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood"
""
>>> inter = lz77_compress2(text)
>>> comp = inter_to_bin(inter)
>>> comp
'01101000011011110111011100100000011011010111010101010110001101101000
00100000011101110111011110110111101100100100000000001010001101110
1010110110001100100001000000111010001101000110001101011000000001111
001100110001101101000001101010101010100011010101011100000000011000110
110100101100011010000000110001000100000010110100110100000001100010
010001000000101110001101110011011010010001000000010010001101000000
000110001010110010001000000010010000111 '
>>> inter2 = bin_to_inter(comp)
>>> inter == inter2
True
>>> lz77_decompress(inter2)
'how much wood would the wood chuck chuck if the wood chuck would chuck wood should could hood'
```

Does it convince you the code is fine? As a toy example, it is not bad. But I would strongly recommend more extensive testing with substantially longer text, going through a larger number of cases.
There and Back Again: The NY Times Test

def process_nytimes():
    btext = urllib.request.urlopen('http://www.nytimes.com/').read()
    nytext = str_to_ascii( btext.decode('utf-8'))
    ny_inter=lz77_compress2(nytext)
    ny_bin=inter_to_bin(ny_inter)
    ny_inter2=bin_to_inter(ny_bin)
    nytext2= lz77_decompress(ny_inter2)
    print("NYT done")
    return nytext,ny_inter,ny_bin,ny_inter2,nytext2

>>> nytext,ny_inter,ny_bin,ny_inter2,nytext2=process_nytimes()
>>> nytext2==nytext
True

Now I am ready to believe the code is OK (of course this is by no means a proof of correctness).

>>> (len(nytext)*7,len(ny_bin))
(896105, 329290)
>>> 329290/896105
0.3674680980465459 # 37% of original

It seems that human generated text is much more amenable to Ziv-Lempel compression than the Cholera proteome.
There and Back Again: Some NY Times Text

>>> nytext, ny_inter, nybin, nyinter2, nytext2 = process_nytimes()
>>> nytext2 == nytext
True

>>> print(nytext2[310:480])
Find breaking news, multimedia, reviews & opinion on Washington, business, sports, movies, travel, books, jobs, education, real estate, cars & more.">
<meta name=

>>> print(nytext2[30191:30475])
Eastman Kodak Files for Bankruptcy Protection</a></h5>
<h6 class="byline">
By MICHAEL J. DE LA MERCED
<span class="timestamp">1 minute ago</span>
</h6>
<p class="summary">
The 131-year-old film pioneer said on Thursday it would continue operating normally during bankruptcy.

Lots of html code, evidently.

Incidentally, this is the 19 Jan 2012 NY Times issue. But we’ll try today’s (December 26, 2012) issue in class.
Improvements to LZ77: gzip

The gzip variant of LZ77 was created and distributed (in 1993) by the Gnu† Free Software Foundation. It contains a number of improvements that make compression more efficient time wise, and also achieves a higher compression ratio.

As we saw, finding the offset, match pairs \([m, k]\) is the main computational bottleneck in the algorithm. To speed it up, gzip hashes triplets of consecutive characters. When we encounter a new location, \(p\), we look up the entry in the hash table with the three character key \(T[p]T[p+1]T[p+2]\). The value of this key is a set of earlier indices with the same key. We use only these (typically very few) indices to try and extend the match.

† The name “GNU” is a recursive acronym for “GNU’s Not Unix!”; it is pronounced g-noo, as one syllable with no vowel sound between the g and the n.
Improvements to LZ77: gzip (cont.)

To prevent the hash tables from growing too much, the text is chopped to blocks, typically of 64,000 characters. Each block is treated separately, and we initialize the hash table for each.

Hashing improves the running time substantially. To improve compression, gzip further employs Huffman code\(^\dagger\). This is used both for the characters and for the offsets (typically close offsets are more frequent than far away ones) and the match lengths.

For every block, the decoding algorithm computes the corresponding Huffman code for all three components (characters, offsets, matches). This code is not known at the receiving end, so the small table describing it is sent as part of the compressed text.

\(^\dagger\)such combination is sometime termed the Deflate compression algorithm.
Compression: Concluding Remarks

There are additional variants of text compression/decompression algorithms, many of which use combinations of Ziv-Lempel and Huffman encoding. In many cases, it is possible to attain higher compression by employing larger blocks or longer windows.

Our compression algorithm as described so far is greedy: Any repeat of length 3 or more is reported and employed right away. Sometimes this is not optimal: We could have an \([m_1, k_1]\) repeat in position \(p\), and an \([m_2, k_2]\) repeat in position \(p+1\) or \(p+2\), with \(k_1 \ll k_2\). Thus a non-greedy algorithm may result in improved compression.

All such improvements would cost more time but produce better compression. In some applications, such tradeoff is well justified.

Compression of gray scale and color images, as well as of documents with a mixture of images and text, uses different approaches. These are based on signal processing techniques, which are often lossy, and are out of scope for our course.