Computer Science 1001.py [†]

Lecture 19: Introduction to Text Compression Huffman compression

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

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

Lecture 19-22: Algorithmis on Text

- Text representation with Unicode/ASCII
- String Matching: Karp–Rabin randomized algorithm using fingerprints.
- Introduction to text compression
  - Huffman compression
  - Lempel-Ziv compression

Lecture 21: Topics

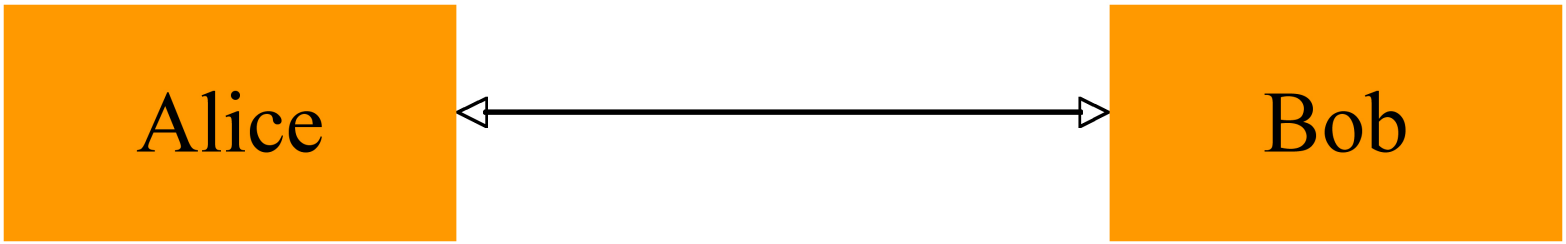
- Introduction to text compression
  - lossless vs. lossy compression schemes
  - Impossibility of universal lossless compression.

- Codes
  - Letters' frequencies in natural languages (reminder).
  - Fixed length and variable length codes.
  - Prefix free codes.

- Huffman code for compression
  - letter-by-letter compression.
  - Optimality
  - Implementation in Python

Communication

Two parties, traditionally named Alice and Bob, have access to a communication line between them, and wish to exchange information.



This is a very general scenario. It could be two kids in class sending notes, written on pieces of paper or (god forbid) text messages under the teacher's nose. Could be you and your brother talking using "traditional" phones, cell phones, or Skype. Could be an unmanned NASA satellite orbiting Mars and communicating with Houston using radio frequencies. It could be the hard drive in your laptop communicating with the CPU over a bus, or your laptop running code in the "cloud" via the "net".
In each scenario, the parties employ communication channels with different characteristics and requirements.

Three Basic Challenges in Communication

1. Reliable communication over unreliable (noisy) channels.

2. Secure (confidential) communication over insecure channels.

3. Frugal (economical) communication over expensive channels.

Three Basic Challenges in Communication

1. Reliable communication over unreliable (noisy) channels.

   Solved using error detection and correction codes.
2. Secure (confidential) communication over insecure channels.

   Solved using cryptography (encryption/ decryption).
3. Frugal (economical) communication over expensive channels.

   Solved using compression (and decompression).

We treat each requirement separately (in separate classes). Of course, in a real scenario, solutions should be combined carefully so the three challenges are efficiently addressed (*e.g.* usually compression should be applied before encryption).

Today, we will discuss compression.

Lossless Compression

A lossless compression scheme consists of two parts:

Compression, C, and decompression, D.

Both C and D are functions from binary strings to binary strings. The major goal of a good compression algorithm is to have `len(C(x)) < len(x)`.

This, by itself, is obviously not hard to achieve. For example, we could simply delete every second bit of `x`. However, under such compression, it is not possible to reconstruct the original string, `x`.

So, in addition to the goal `len(C(x))< len(x)`, we also require: For every `x`, if `y=C(x)`, then `D(y)=x`.

Lossless vs. Lossy Compression

When compressing text, we normally want no data loss.

However, when the string in question represent (discretized samples of) analogue data, for example audio, images or video, lossy compression, in which part of the data is lost, is often used.

If the human eye or ear (at least the average eye or ear) cannot distinguish between the original and the compressed version, then lossy compression is typically acceptable and used. Compression can be achieved, for example, by removing high frequencies from the audio or image, or by reducing the number of color combinations. Video is often compressible in 100:1 ratio, audio in 10:1, and images in 5:1, with hardly any noticeable change in quality.

MP3 for audio, JPEG for images, and MPEG-4 for video are among the well known, lossy compression schemes.

Universal Lossless Compression is Impossible

We say that a compression scheme is universal if for every `x`, `len(C(x)) < len(x)`.

Claim: There is no universal, lossless compression scheme.

Proof: A simple counting argument:

A lossless compression algorithm (mapping binary strings to binary strings) must be one to one on its domain, $\{0,1\}^*$. In particular, it should map $\{0,1\}^n$ in a one-to-one mapping to binary strings of smaller lengths.

But there are not enough strings to map into: 2^n strings in the domain, only $2^n - 1$ strings in the range. ♠

Universal Lossless Compression is Impossible, cont.

The counting argument means we cannot compress everything.

For example, compression algorithms applied to random text will not compress. They may even expand the text.

But it does not imply we cannot compress a small fraction of all text (or image, audio, or video) sources.

The key to compression are redundancies (*e.g.* repetitions, similarities, etc.). Lossless (and lossy) algorithms look for these and try to exploit them in describing the data more succinctly.

Codes and Codewords

In our context today, a code is a one-to-one mapping from single characters to binary strings, called codewords.

In an earlier class, we talked about Unicode representation, and saw this function, which translates from text to bits, restricting each character to 8 bits:

```
def text2bit_stream(text):
    return "".join([bin(ord(c))[2:].zfill(8) for c in text])
```

(However, Unicode assigns more than 8 bits for most characters)

ASCII as a Fixed Length (7 bits) Code

Recall that to keep Unicode backward compatible (to ASCII), all 128 ASCII values were preserved, and a a leading '0' was added to them. Values beginning with a '1' represent non-ASCII characters.

Today we will restrict ourselves, for simplicity, to pure ASCII values, thus each character can be represented using only 7 bits:

```
def ascii2bit_stream(text):
    return "".join([bin(ord(c))[2:].zfill(7) for c in text])
```

```
>>> ascii2bit_stream("abc")
'110000111000101100011'
>>> len(ascii2bit_stream("abc"))
21
```

A code is called fixed length code if all characters are mapped to binary strings of the same length.
(ASCII, as opposed to Unicode, is a fixed length code).

Huffman Code

In Huffman code, we abandon the paradigm of a fixed length code. Instead, we employ a variable length code, where different letters are encoded by binary strings of different lengths.

Basic Idea:
Frequent letters will be encoded using short binary strings (shorter than 7 bits).
Rare letters will be encoded using long binary strings (typically longer than 7 bits).

This way, the encoding of a typical string will be shorter, since it contains more frequent letters (where we save length) than rare ones (where we pay extra length).
(Note that this approach will not work for random strings.)

The rest is just details.
(but, as you surely know, the devil is in the details).

Huffman Code (cont.)

Huffman code:

- assigns short coding to letters occurring frequently, and longer coding to letters occurring infrequently.

This mean it is a variable length code. Encoding strings with

Huffman code:

- makes it possible to translate back to the original.
- requires a smaller number of bits (usually, not always).

These two properties mean Huffman code can be used as a lossless compression scheme.

A Short Detour: The String Method `join`

```
>>> help(str.join)
Help on method_descriptor:
join(...)
    S.join(iterable) -> str
    Return a string which is the concatenation of the strings
    in the iterable.  The separator between elements is S.


>>> string="abcdefg"
>>> "".join(ch for ch in string)
'abcdefg'
>>> text=""
>>> text.join(ch for ch in string)
'abcdefg'
>>> text
''              # no wonder: strings are immutable
>>> t = (ch for ch in string)
>>> t
<generator object <genexpr> at 0x000000000352F9D8>
>>> "".join(t)
'abcdefg'
```

Recall that strings are immutable.

Note that `(ch for ch in string)` is a generator.

The String Method `join`, cont.

Another example:

```
>>> string="abcdefg"
>>> "xy".join(string)
'axybxycxydxyexyfxyg'
```

Recall the two equivalent ways to writing this:

```
>>> string="abcdefg"
>>> "".join(ch for ch in string)
'abcdefg'
>>> str.join("",(ch for ch in string))  # need the parentheses
'abcdefg'
```

Alternatively, we could have used string concatenation, `+`. This assigns a new string object in every iteration, and will typically be slower (and less easy on memory) than using `join` when generating very long strings.

```
>>> newtext=""
>>> for ch in string:
    newtext=newtext+ch
>>> newtext
'abcdefg'
```

Frequencies of Letters in Natural Languages (reminder)

The distribution of single letters in any natural languages' texts is highly non uniform.
We can compute these frequencies by taking a "representative text", or corpus, and simply count letters. For example, in English, "e" appears approximately in 12.5% of all letters, whereas "z" accounts for just 0.1%. In Hebrew, "Yud" (`ord=1497`) appears approximately in 11% of all letters (not counting spaces, digits, etc.) while "Za'in" (`ord=1494`) appears approximately in just 0.8% of all letters.

We will use a function char_count, similar to the one we saw in a previous class.

Frequencies of Letters in Natural Languages (reminder)

We will use a function char_count, similar to the one we saw in a previous class:

```
def char_count(text):
    """ Counts the number of each character in text.
        Returns a dictionary, with keys being the observed character
        values being the counts """
    d = {}
    for ch in text:
        if ch in d:
            d[ch] += 1
        else:
            d[ch] = 1
    return d
```

Prefix Free Codes

A code is called prefix free code if for all pairs of characters γ, τ that are mapped to binary strings $C(\gamma), C(\tau)$, no binary string is a prefix of the other binary string.

Note that fixed length implies prefix free.

As a concrete example, consider the following three codes, both mapping the set of six letters $\{a, b, c, d, e, f\}$ to binary strings.

Code 1:

a	b	c	d	e	f
000	001	010	011	100	101

Code 2:

a	b	c	d	e	f
0	10	110	1110	11110	111110

Code 3:

a	b	c	d	e	f
0	1	00	01	10	11

Prefix Free Codes and Ambiguity

Code 1:

a	b	c	d	e	f
000	001	010	011	100	101

Code 2:

a	b	c	d	e	f
0	10	110	1110	11110	111110

Code 3:

a	b	c	d	e	f
0	1	00	01	10	11

Code 1 is a fixed length code, hence it is also a prefix free code.

Code 2 and 3 are variable length codes.

Code 2 is a prefix free code: No codeword is a prefix of another.
Code 3 is not. For example, 1 is a prefix of 11.

Why do we care? Suppose you get the binary string 100 (no spaces!).
How would you decode it, according to each of the three codes?
100 encodes e by Code 1. It encodes ba by Code 2.

But by Code 3, it could encode baa or bc or ea, so we got ambiguity here, which is bad. This is why variable length codes must be prefix free.

Optimality of Huffman Code

Given a set of n characters $\Sigma = \{a_1, a_2, \ldots, a_n\}$, and a set of positive weights (counts), $W = \{w_1, w_2, \ldots, w_n\}$, corresponding to the characters,

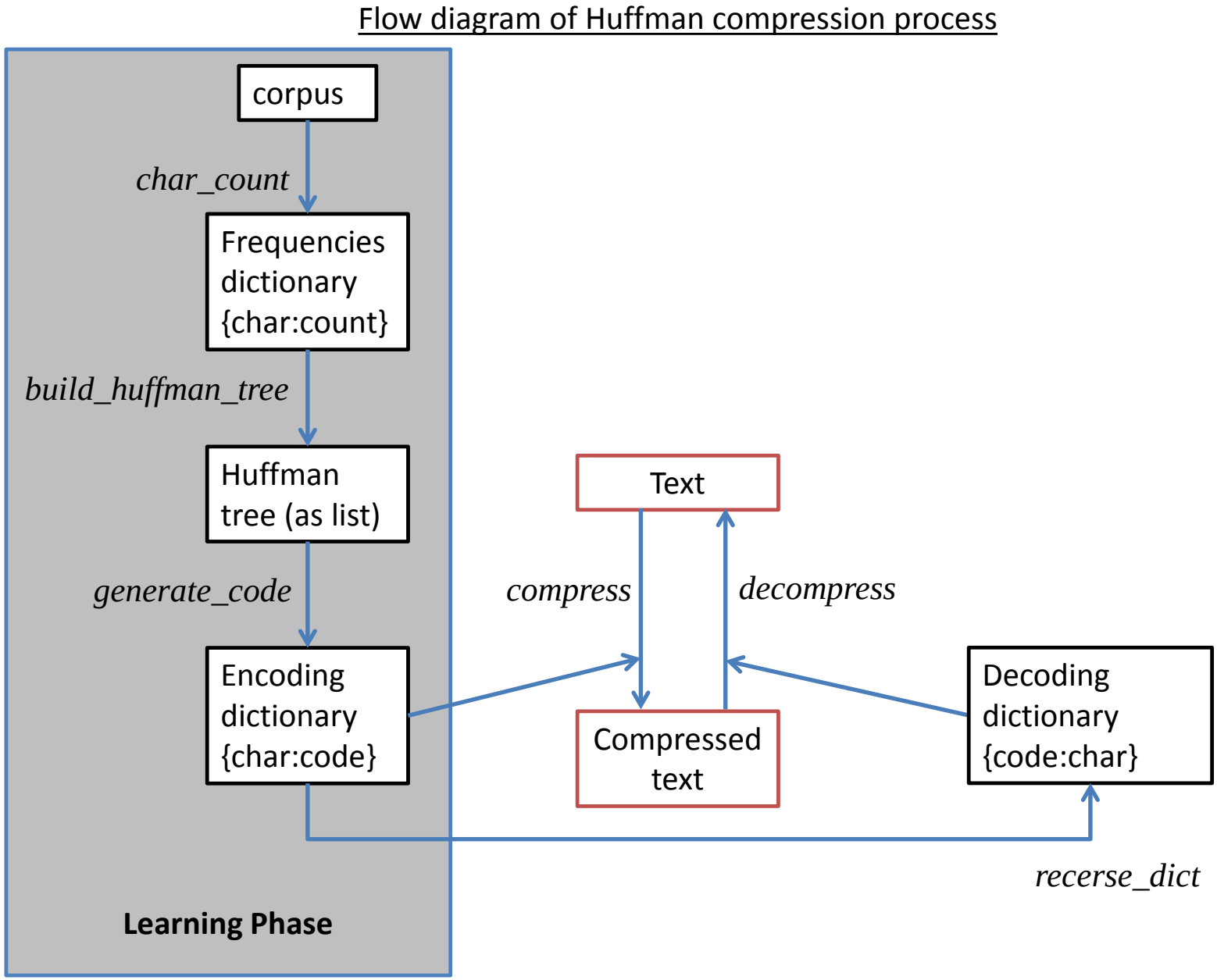
A variable length, prefix free code, $C(\Sigma, W) = \{c_1, c_2, \ldots, c_n\}$, where each c_i is a binary string.

The weighted length of a code $C = \{c_1, c_2, \ldots, c_n\}$ with respect to the set of weights $W = \{w_1, w_2, \ldots, w_n\}$ is defined as $L_W(C) = \sum_{i=1}^{n} w_i \cdot \mathsf{len}(c_i)$.

A code $D(\Sigma, W)$ is called optimal if for any code, $C(\Sigma, W)$, $L_W(D) \leq L_W(C)$.

Note that optimality here is defined with regards to given Σ, W. Huffman code is optimal, but we will not prove it.

Building the Huffman Code: A Road Map



Huffman Code: The main steps

- Collect character counts from a representative corpus.
- Construct Huffman tree.
  The Huffman tree is a binary tree whose leaves represent letters, and the path from the root to a leave represents the encoding of the letter, considering left as 0, and right as 1.
- Turn the tree into an encoding dictionary (ascii characters to binary strings).
- Reverse the dictionary to get a decoding dictionary (binary strings to ascii characters).

The first step of the road map

We illustrate the first step of the road map with a small example, adapted from Structure and Interpretation of Computer Programs, by Abelson and Sussman):

```
>>> t="aaaaaaaabbbcdefgh"
>>> char_count_dict = char_count(t)
>>> char_count_dict
{'a': 8, 'b': 3, 'c': 1, 'd': 1, 'e': 1, 'f': 1, 'g': 1, 'h': 1}
```

Next, we construct the Huffman Tree.

Constructing the Huffman Tree

- Create a priority queue (list) to represent letters' counts. Its members will represent binary trees.
- Initially, each element in the priority queue includes a letter and its frequency (which is considered its weight). These are viewed as single node trees. They will be the leaves of the tree in the end of the process
- Iterate the following: Remove the two smallest-weight items from the priority queue. Join them to form a new, compound item, whose weight equals the sum of the two weights, and place it in the priority queue
- It represents a new tree, whose root is the new node, and whose children are the two trees that were joined.
- So each node includes a set of characters and its total weight (sum of counts)
- The end result is a priority queue (list) with one compound item.
- It represents the Huffman tree.

Building the Huffman Tree: using the same example

```
>>> char_count_dict = char_count("aaaaaaaabbbcdefgh")
>>> char_count_dict
{'a': 8, 'b': 3, 'c': 1, 'd': 1, 'e': 1, 'f': 1, 'g': 1, 'h': 1}
```

The initial value of the priority queue:

```
>>> [(c,cnt) for (c,cnt) in char_count_dict.items()]
[('a', 8), ('b', 3), ('c', 1), ('d', 1), ('e', 1), ('f', 1),
 ('g', 1), ('h', 1)]
```

`c.items()` returns a list of pairs of the dictionary `c`. This is the initial value of the `priority queue` we use next.
A `priority queue` is a data structure that supports the operations `insert` and `remove smallest`. Elements of the priority queue are binary trees, represented as list structures, where the priority is the weight of the tree. We will use a very naive implementation of `priority queue`: a list, where we `insert` elements at the end, and `extract minimum` is done by finding the minimal element and removing it. You will study a more efficient implementation of `priority queue` in the Data Structures course.

Reminder: List methods pop, append and index

```
>>> lst = [5,3,6,2,8,9]
>>> lst.pop(4)  # pop removes item at specified location
8                          # pop mutates list and returns item
>>> lst
[5, 3, 6, 2, 9]
>>> lst.append(7)
>>> lst
[5, 3, 6, 2, 9, 7]
>>> min(lst)
2
>>> lst.index(min(lst))  # index returns the index of
3                  # the element found by linear search
>>> lst
[5, 3, 6, 2, 9, 7]
>>> lst.pop(lst.index(min(lst)))
2
>>> lst
[5, 3, 6, 9, 7]
```

The operation pop returns the value that it removes, and also mutates the list . The operation append (always at the end) also mutates the list. Both do not change the list identity.

Building the Huffman Tree, represented as a List

We implement a priority queue, prioritized by counts of nested sets of characters, initialized from a dictionary of characters counts.

```
def build_huffman_tree(char_count_dict):
    """ Recieves dictionary with char:count entries
        Generates a LIST structure representing
        the binary Huffman encoding tree """
    queue = [(c,cnt) for (c,cnt) in char_count_dict.items()]

    while len(queue) > 1:
        #print(queue)
        # combine two smallest elements
        A, cntA = extract_min(queue)     # smallest in queue
        B, cntB = extract_min(queue)     # next smallest
        chars = [A,B]
        weight = cntA + cntB             # combined weight
        queue.append((chars, weight))    # insert combined node

    # only root node left
    #print("final queue:", queue)
    root, weight_trash = extract_min(queue) # weight_trash unused
    return root    #a LIST representing the tree structure
```

Extract Minimum from the Priority Queue

We iterate over the queue and look for the minimal count:

```
def extract_min(queue):
    """ queue is a list of 2-tuples (x,y).
        remove and return the tuple with minimal y """
    min_pair = queue[0]
    for pair in queue:
        if pair[1] < min_pair[1]:
            min_pair = pair

    queue.remove(min_pair)
    return min_pair
```

We could also use lambda expressions and Python's min function:

```
def extract_min(queue): #the shorter, "Pythonic" way
    min_pair = min(queue, key = lambda pair: pair[1])
    queue.remove(min_pair)
    return min_pair
```

Building the Huffman List: same Small Example[‡]

The step of the road map:

```
>>> t = "aaaaaaaabbbcdefgh"
>>> char_count_dict = char_count(t)
>>> char_count_dict
{'a': 8, 'b': 3, 'c': 1, 'd': 1, 'e': 1, 'f': 1, 'g': 1, 'h': 1}
```

To view

```
>>> [(c,cnt) for (c,cnt) in char_count_dict.items()]
[('a', 8), ('b', 3), ('c', 1), ('d', 1), ('e', 1), ('f', 1),
 ('g', 1), ('h', 1)]
```

We now build the tree:

```
>>> huff = build_huffman_tree(c)
>>> huff
['a', [['c', 'd'], ['e', 'f']], [['g', 'h'], 'b']]
>>> huff[0]
'a'           # the code for a is 0
>>> huff[1][0][1][0]
'e'           # the code for e is 1010
```

[‡]Adapted from Structure and Interpretation of Computer Programs, by Abelson and Sussman

Building the Huffman List: Another Small Example

```
>>> text = "giddddddddddddddddddddddddday,mate"
>>> len(text)
34
>>> char_count(text)
{'a':2,'e':1,'d':25,'g':1,'i':1,'m':1,',':1,'t':1,'y':1}
>>> len(char_count(text))
9
>>> build_huffman_tree(char_count(text))
[[[[['i','m'],[',','t']],['g','y'],['e','a']]],'d']]
```

A Comment on the Hufman Tree Structure

During the construction of the Huffman tree, whenever there is more than one element with the same minimal weight, the element that is extracted may depend on the implementation. Hence, different implementation may yield a different tree for the same initial dictionary.

Such different trees may have the same structure with some letters changing positions, but they may also have different structure.

However all these trees will yield an optimal code: if the code obtained from the tree is D, then for any code, C, $L_W(D) \leq L_W(C)$ where $L_W(.)$ is as defined earlier.

The Dictionary Method update

Python built-in dictionary method update takes an existing dictionary and updates its entries by those of another dictionary. Existing entries (key:value pairs) are "run over".

```
>>> code1={'a': '11', 'c': '10', 'b': '0'}
>>> code1.update({'a': '110',"d":"111"})
>>> code1
{'a': '110', 'c': '10', 'b': '0', 'd': '111'}
```

We will find the update method useful for generating a Huffman code from a Huffman tree. The Huffman code will be a dictionary, where keys are characters, and values are the binary strings encoding them.

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. Returns a dictionary, with keys being characters at leaves.

```
def generate_code(huff_tree, prefix=""):
    """ Receives a Huffman tree with embedded encoding,
        and a prefix of encodings.
        returns a dictionary where characters are
        keys and associated binary strings are values."""
    if isinstance(huff_tree, str): # a leaf
        return {huff_tree: prefix}
    else:
        lchild, rchild = huff_tree[0], huff_tree[1]
        codebook = {}

        codebook.update( generate_code(lchild, prefix+'0'))
        codebook.update( generate_code(rchild, prefix+'1'))

        return codebook
```

Oh, the beauty of recursion...

From Huffman tree To Huffman Code: A Small Example

Explanation: In each call, the value of prefix is the path from the root to the current node. If the node is a leaf, create a dictionary with a single entry. Otherwise use two recursive calls to create two disctionaries, and combine them.

A small example:

```
>>> t = "aaaaaaaabbbcdefgh"
>>> char_count_dict = char_count(t)
>>> [(c,cnt) for (c,cnt) in char_count_dict.items()]
[('a', 8), ('b', 3), ('c', 1), ('d', 1), ('e', 1),
 ('f', 1), ('g', 1), ('h', 1)]
>>> tree = build_huffman_tree(c)
>>> generate_code(tree)
{'a': '0', 'b': '111', 'c': '1000', 'd': '1001', 'e': '1010',
 'f': '1011', 'g': '1100', 'h': '1101'}
```

The length of encodings vary from 1 to 4.
More frequent letters are assigned shorter encodings.

Counting Characters: A Real Example

Of course, the previous small examples are far too small. We found it appropriate to collect letters' frequencies from Wikipedia's own Huffman's code entry.

Counting Characters: Wikipedia Huffman Code Entry

```
import urllib.request

def get_html_text(path):
    text = urllib.request.urlopen(path).read()
    text = text.decode('utf -8')
    return text


def clean_non_ascii(text):
    """ Gets rid of non-ASCII characters in text"""
    return ''.join(ch for ch in text if ord(ch)<128)


>>> wiki=get_html_text("https://
                            en.wikipedia.org/wiki/Huffman_coding")
>>> wiki_clean = clean_non_ascii(wiki)
>>> counts =  char_count(wiki_clean)
```

Counting Characters: Real Results

```
>>> counts
{'R': 196, 'l': 5057, '#': 101, '(': 194, '.': 1170, ',': 545,
'/': 3761, 'a': 7715, '5': 286, '<': 4234, 'B': 190, "'": 60,
'\t': 1006, '@': 3, 'Z': 68, ':': 699, '2': 617, 'D': 270,
'+': 78, 'n': 5827, 'w': 1835, 'z': 64, 'b': 1294, 'g': 2027,
';': 467, '8': 420, ']': 66, '?': 50, 'X': 170, 'i': 8323, 'O': 736
'P': 156, 'v': 836, 'q': 116, 'k': 1027, '6': 234, 'e': 9071,
'~': 2, 'A': 389, 'H': 334, 'c': 3415, 'h': 2832, '7': 258,
'%': 569, 'K': 17, 'U': 34, 'f': 2201, '1': 813, 'Y': 3,
'T': 303, 'N': 67, 't': 7788, '\n': 1691, 'G': 35, 'x': 673,
'S': 191, 'j': 70, '!': 32, '|': 8, '4': 259, 'E': 258, '$': 5,
'J': 121, 'y': 1132, 'C': 343, 'p': 2664, 'm': 4042, '&': 197,
'L': 211, '-': 1785, '\\': 211, 'o': 5129, 'u': 1794, '"': 4996,
'_': 799, ')': 194, '[': 66, '}': 196, '9': 381, 'W': 102,
'M': 292, 'r': 4507, 'F': 151, '0': 138, 'V': 51, 'Q': 27,
'^': 13, 's': 5499, ' ': 14318, '=': 2485, '{': 196, 'I': 136,
'3': 467, '>': 4234, 'd': 3253, '*': 5}
```

Counting Characters: Real Results Sorted

```
>>> sorted(counts.items(), key = lambda pair: -pair[1])
[(' ', 14318), ('e', 9071), ('i', 8323), ('t', 7788), ('a', 7715),
 ('n', 5827), ('s', 5499), ('o', 5129), ('l', 5057), ('"', 4996),
 ('r', 4507), ('<', 4234), ('>', 4234), ('m', 4042), ('/', 3761),
 ('c', 3415), ('d', 3253), ('h', 2832), ('p', 2664), ('=', 2485),
 ('f', 2201), ('g', 2027), ('w', 1835), ('u', 1794), ('-', 1785),
 ('\n', 1691), ('b', 1294), ('.', 1170), ('y', 1132), ('k', 1027),
 ('\t', 1006), ('v', 836), ('1', 813), ('_', 799), ('0', 736),
 (':', 699), ('x', 673), ('2', 617), ('%', 569), (',', 545),
 ('3', 467), (';', 467), ('8', 420), ('A', 389), ('9', 381),
 ('C', 343), ('H', 334), ('T', 303), ('M', 292), ('5', 286),
 ('D', 270), ('4', 259), ('E', 258), ('7', 258), ('6', 234),
 ('\\', 211), ('L', 211), ('&', 197), ('}', 196), ('R', 196),
 ('{', 196), ('(', 194), (')', 194), ('S', 191), ('B', 190),
 ('X', 170), ('P', 156), ('F', 151), ('O', 138), ('I', 136),
 ('J', 121), ('q', 116), ('W', 102), ('#', 101), ('+', 78),
 ('j', 70), ('Z', 68), ('N', 67), ('[', 66), (']', 66), ('z', 64),
 ("'", 60), ('V', 51), ('?', 50), ('G', 35), ('U', 34), ('!', 32),
 ('Q', 27), ('K', 17), ('^', 13), ('|', 8), ('$', 5), ('*', 5),
 ('@', 3), ('Y', 3), ('~', 2)]
```

Note that this text is far from "typical" English text.

Building the Huffman tree: A Larger Example, cont.

```
>>> wiki=get_html_text("https://
                              en.wikipedia.org/wiki/Huffman_coding")
>>> wiki_clean = clean_non_ascii(wiki)
>>> counts =  char_count(wiki_clean)
>>> tree = build_huffman_tree(char_count(wiki_clean))
>>> tree

[[[' ', [['c', ['-', 'u']], [[[[['L', '\\'], [['V', "'"],
    'q'], '6']], [';', '3']], 'w'], '/']]], [['a', 't'], [['m',
    ['g', ['\t', [[['J', ['z', ['!', [['|', ['Y', '$']], 'K']]]],
    '7'], ['E', '4']]]]], 'i']]], [[['<', '>'], [['k',
   [[[[']', '['], ['N', 'Z']], 'D'], ',']], 'f'], 'r']],
   ['e', [[[[[['I', 'O'], '5'], '%'], 'y'], ['.', [[[[['U', 'G'],
    'j'],  'F'], 'M'], '2']]], '"']]], [[['l', ['=', ['b',
   [['T', ['P', 'X']], 'x']]]], ['o', ['p', 'h']]], [['s', 'n'],
   [[[[['H', 'C'], ':'], ['0', [[['+', ['?', [[['*',
   ['~', '@']], '^'], 'Q']]], 'B'], '9']]], [[['S', '('], 'A'],
   [[')', 'R'], ['}', '{']]], ['_', '1']]], ['d', [[['&',
   ['#', 'W']], '8'], 'v'], '\n']]]]]]]
```

Now it is even harder to see what is going on here

Building the Huffman tree: A Larger Example, cont.

```
>>> tree[0][1][0]
['a', 't']
>>> tree[0][1][0][0]
'a'
>>> tree[0][1][0][1]
't'
```

Indeed, the characters, "a" and "t", which are among the most frequent, enjoy a short encoding (just 4 bits long).

Huffman Code of Wikipedia Huffman Page

```
>>> code = generate_code(tree)
>>> sorted(code.items(), key = lambda pair:len(pair[1]))
[(' ', '000'), ('t', '0101'), ('i', '0111'), ('a', '0100'),
  ('e', '1010'), ('m', '01100'), ('s', '11100'),
  ('n', '11101'), ('/', '00111'), ('o', '11010'),


# many items removed


('$', '011011100111011'), ('*', '1111001100111000'),
('Y', '011011100111010'), ('@', '1111001100110011'),
('~', '1111001100110010')]
```

There are 96 different characters, and the lengths of their encodings vary from 4 to 16.

Understanding Huffman Code of Wikipedia Huffman Page

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

```
>>> sorted(counts.items(), key = lambda pair: -pair[1])]
[(' ', 14318), ('e', 9071), ('i', 8323), ('t', 7788),
 ('a', 7715), ('n', 5827), ('s', 5499), ('o', 5129),
  ('l', 5057), ('"', 4996), ('r', 4507), ('<', 4234),
  ... ]

>>> sorted(code.items(), key = lambda pair:len(pair[1]))
[(' ', '000'), ('t', '0101'), ('i', '0111'), ('a', '0100'),
  ('e', '1010'), ('m', '01100'), ('s', '11100'),
  ('n', '11101'), ('/', '00111'), ('o', '11010'),
  ...]
```

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 (using `join`) to the forming output (binary string).

```
def compress(text, encoding_dict):
    """ compress text using encoding dictionary """
    assert isinstance(text, str)
    return "".join(encoding_dict[ch] for ch in text)
```

Building the 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 directly. 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 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.

Reversing a Dictionary and Decompression: Code

```
def reverse_dict(d):
    """build the "reverse" of encoding dictionary"""
    return {y:x for (x,y) in d.items()}
```

```
def decompress(bits, decoding_dict):
    prefix = ""
    result = []
    for bit in bits:
        prefix += bit
        if prefix in decoding_dict:
            result.append(decoding_dict[prefix])
            prefix = ""  #restart
    assert prefix == "" # must finish last codeword
    return "".join(result)  # converts list of chars to a string
```

Dictionary for Decompression: Example

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

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

We are now ready to complete the full compress–decompress cycle.

The Full Cycle: Encoding, Compressing, Decompressing

```
def full_cycle(corpus, text):
  #generate Huffman code from corpus
  print("corpus:\n", corpus, end="\n\n")
  counts =  char_count(corpus)
  print(counts, end="\n\n")
  tree = build_huffman_tree(counts)
  print(tree, end="\n\n")
  code = generate_code(tree)
  print(code, end="\n\n")
  #compress text using code
  print("text:\n", text, end="\n\n")
  print("text len in bits:",len(ascii2bit_stream(text)),end="\n\n")
            # == len(text)*7
  print(ascii2bit_stream(text), end="\n\n")
  C = compress(text, code)
  print("compressed len in bits:", len(C), end="\n\n")
  print(C, end="\n\n")
  print("comp.ratio:",len(C)/len(ascii2bit_stream(text)),end="\n\n"
  #decompression, back to original code
  decode = reverse_dict(code)
  print(decode, end="\n\n")
  D = decompress(C, decode)
  print(D, end="\n\n")
  assert D == text #just making sure
```

The Full Cycle: Encoding, Compressing, Decompressing (cont.)

Let us now compress and decompress a few sentences.

corpus = """Selected Alan Perlis Quotations:
(1) It is easier to write an incorrect program
than understand a correct one.
(2) One man's constant is another man's variable. """

text = "fun"

```
>>> full_cycle(corpus, text)
```

Executions and explanations in class.

The Full Cycle: Encoding, Compressing, Decompressing (cont.)

On the first trial we got the code generated from the corpus, but when compressing, we got this error message:

```
Traceback (most recent call last):
  File "D:\huffman.py", line 182, in <module>
    full_cycle(corpus, text)
  File "D:\huffman.py", line 162, in full_cycle
    C = compress(text, code)
  File "D:\huffman.py", line 91, in compress
    return "".join(encoding_dict[ch] for ch in text)
  File "D:\huffman.py", line 91, in <genexpr>
    return "".join(encoding_dict[ch] for ch in text)
KeyError: 'f'
```

What does that mean?

Missing Characters in the Corpus

Indeed, the corpus did not include all characters in the text, and in particular the character 'f':

```
>>> 'f' in text
True
>>> 'f' in corpus
False
>>> code = generate_code(build_huffman_tree(char_count(corpus)))
>>> code['f']
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    code['f']
KeyError: 'f'
```

How should we fix this?

Missing Characters in the Corpus - Take 1

corpus = """Selected Alan Perlis Quotations:
(1) It is easier to write an incorrect program
than understand a correct one.
(2) One man's constant is another man's variable. """

text = "fun"

```
>>> full_cycle(corpus + 'f', text)   #added 'f' to corpus
```

Executions and explanations in class.

Compression Ratio - Take 1

```
corpus = """Selected Alan Perlis Quotations:
        (1) It is easier to write an incorrect program
              than understand a correct one.
        (2) One man's constant is another man's variable. """

text = "fun"
```

```
text len in bits: 21
110011011101011101110
compressed len in bits: 17
10000110000101110
compression ratio: 0.80952380
```

Missing Characters in the Corpus - Take 2

In this case the only missing character was 'f'.

Generally, we could meticulously go over the missing characters and add them to the corpus. 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, to the corpus, thus making sure every ascii character is represented.

This will change the counts. However, for a large corpus, the effect is so slight we will hardly notice it (and yes, we do know that many of the ASCII characters are obsolete and not really needed).

```
>>> 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'
```

Missing Characters in the Corpus - Take 2

corpus = """Selected Alan Perlis Quotations:
(1) It is easier to write an incorrect program
than understand a correct one.
(2) One man's constant is another man's variable. """

text = "fun"

```
>>> asci = "".join(chr(i) for i in range(128))
>>> full_cycle(corpus + asci, text) #added all 128 ascii to corpus
```

Executions and explanations in class.

Compression Ratio - Take 2

```
corpus = """Selected Alan Perlis Quotations:
        (1) It is easier to write an incorrect program
              than understand a correct one.
        (2) One man's constant is another man's variable. """

text = "fun"
```

```
text len in bits: 21
110011011101011101110
compressed len in bits: 20
01100010111011011110
compression ratio: 0.952380952
```

Compressing a Random String

```
import random

def random_string(n):
    """ Generate a random ascii sequence of length n """
    return "".join(chr(random.randrange(128)) for i in range(n))


def rand_compression_test(n):
    print("compressing a random text of lengrh", n)
    rand_text = random_string(n)
    corpus = text = rand_text
    code = generate_code(build_huffman_tree(char_count(corpus)))
    C = compress(text, code)  #best possible corpus is text itself
    print("compression ratio:", len(C) / len(ascii2bit_stream(text)
```

Compressing a Random String - Results

```
>>> rand_compression_test(10)
compressing a random text of lengrh 10
compression ratio: 0.4857142857142857
>>> rand_compression_test(100)
compressing a random text of lengrh 100
compression ratio: 0.8857142857142857
>>> rand_compression_test(1000)
compressing a random text of lengrh 1000
compression ratio: 0.9902857142857143
>>> rand_compression_test(10000)
compressing a random text of lengrh 10000
compression ratio: 1.0
```

So Huffman cannot save even a single bit out of long random strings.
On the other hand, it does not expand it either!
To understand why, recall that the frequency of all letters is equal.
How would the Huffman tree look like?

Additional Examples with Larger Texts - for self reference

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).

```
>>> homer_text = urllib.request.urlopen(
'http://www.gutenberg.org/ebooks/3160.txt.utf8').read()
>>> homer_text = homer_text.decode('utf-8')
>>> homer_text = clean_non_ascii(homer_text)
>>> len(ascii2bit_stream(homer_text))
5110805    # length (ascii) in bits
>>> count = char_count(homer_text)
>>> tree = build_huffman_tree(count)
>>> homer_encode = generate_code(tree)
>>> homer_decode = reverse_dict(homer_encode)
```

The Full Cycle with Larger Texts (2)

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

```
>>> william_text = urllib.request.urlopen(
    'http://www.gutenberg.org/ebooks/1112.txt.utf8').read()
>>> william_text = william_text.decode('utf-8')
>>> william_text = clean_non_ascii(william_text)
>>> len(ascii2bit_stream(william_text))
1184260        # length (ascii) in bits
>>> count = char_count(william_text)
>>> tree = build_huffman_tree(count)
>>> william_encode = generate_code(tree)
>>> william_decode =  reverse_dict(william_encode)
```

The Full Cycle with Larger Texts (3)

We are finally ready to compress and decompress

```
>>> len(ascii2bit_stream(homer_text))
5110805
>>> Odd1 = compress(homer_text, homer_encode)
>>> len(Odd1)
3389460  # 66% of original
>>> decompress(Odd1, homer_decode)[0:21]
'The Project Gutenberg'
>>> decompress(Odd1, homer_decode)[30:77]
' The Odyssey of Homer, trans. by Alexander Pope'

>>> Odd2 = compress(homer_text, william_encode)
Traceback (most recent call last):
  File "<pyshell#169>", line 1, in <module>
    Odd2=compress(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`. We already saw this problem before:

```
>>> 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 (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 = clean_non_ascii(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 = reverse_dict(homer_encode_dict)
  homer_encoded_text = compress(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 = decompress(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 = clean_non_ascii(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 = reverse_dict(william_encode_dict)
  william_encoded_text = compress(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 = decompress(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 = process_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=process_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 = compress(h_original, h_encode)
>>> len(homer_using_homer)
3391176

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

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

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

More Cross Compression Performance

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

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

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

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

>>> decompress(william_using_homer,w_decode) == w_original
False
>>> decompress(william_using_homer,h_decode)[:50]
'This Etext file is presented by Project Gutenberg,'
>>> decompress(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 (for reference only)

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. (In an efficient implementation of priority queue each insertion takes $O(\log |\Sigma|)$ steps for a total of $O(|\Sigma| \log |\Sigma|)$).
- 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: $O((\ell + m) \log |\Sigma|)$, where ℓ equals length of compressed string, and assuming hash query takes $O(1)$ steps. The log factor comes from the size in bits of each code.
- 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.