Extended Introduction to Computer Science CS1001.py

Module H (Text): Introduction to Text Compression Huffman compression

Instructors: Elhanan Borenstein, Michal Kleinbort
Teaching Assistants: Noam Parzanchevsky, Asaf Cassel, Shaked Dovrat, Omri Porat

School of Computer Science
Tel-Aviv University

Spring Semester 20-21
http://tau-cs1001-py.wikidot.com

Module H - Text Related Algorithms: Overview

- CYK parsing algorithm
- Text compression
  - Huffman compression (today)
  - Lempel-Ziv compression (next)

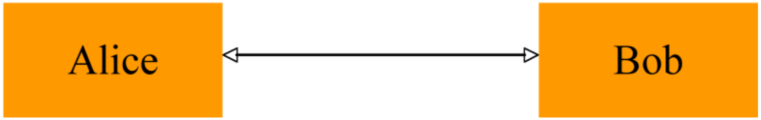
Lecture Plan

- Introduction to text compression
  - lossless vs. lossy compression schemes
  - Impossibility of universal lossless compression.
- Codes
  - Letters' frequencies in natural languages.
  - Fixed length and variable length codes.
  - Prefix free codes.
- Huffman code for compression
  - Demo
  - Implementation in Python

Introduction

Communication

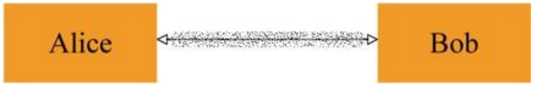
Two parties, traditionally names Alice and Bob, have access to a communication line between them, and wish to exchange information.



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 friend 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.

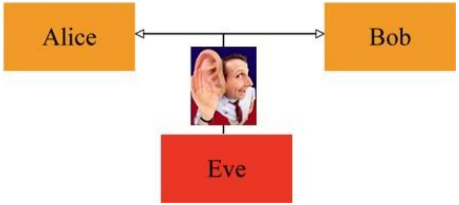
Three Basic Challenges in Communication

1. Reliable communication over unreliable (noisy) channels.



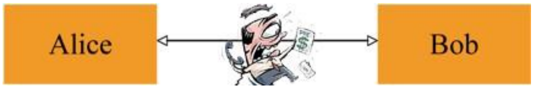
✓

2. Secure (confidential) communication over insecure channels.



✓

3. Frugal (economical) communication over expensive channels.

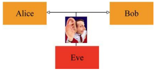


Three Basic Challenges in Communication

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 $\text{len}(C(x)) < \text{len}(x)$.

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 $\text{len}(C(x)) < \text{len}(x)$, 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.

Note: for 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 decompressed 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, $\text{len}(C(x)) < \text{len}(x)$.

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 real text is not random...

Huffman Code

We will now delve into Huffman text compression algorithm.

We will start be specifying two desired properties of that code: variable-length, and prefix-free.

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”)

Two Properties of Huffman Compression Code

- Variable length code
- Prefix-free code

Fixed vs. Variable Length Codes

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

A code is called fixed-length code if all characters are mapped to binary strings of the same length. ASCII is a notable example.
In a variable-length code, different characters are encoded by binary strings of different lengths. For example: Unicode.

Huffman code is also a variable-length code.

Basic Idea:

Frequent letters will be encoded using short binary strings.
Rare letters will be encoded using long binary strings.

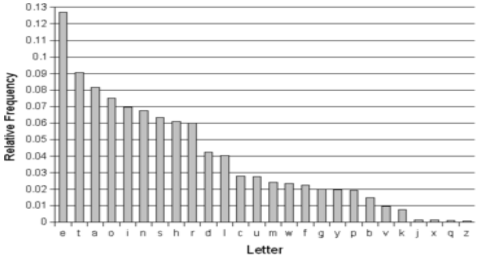
This way, the encoding of a typical string will be shorter, since it contains more frequent letters (where we save length) than rare ones (where we pay extra length).

Frequencies of Letters in Natural Languages

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 12.8% of all letters, whereas "z" accounts for just 0.1%.

Frequencies of Letters in Natural Languages

The distribution of single letters in any natural languages' texts is highly non uniform. According to the diagram below (taken from Wikipedia), the four most frequent letters are *{e, t, a, o}*. The letter e appears in 12.8% of written English, t's frequency is 9.05%, a's is 8.1%, and o's is 7.6%.



The frequencies in this diagram are based some "representative text", or corpus (for example, a dictionary).

More on Frequencies of Letters

Letter frequencies in different languages differ substantially. For example, while in many European languages the letter e had the highest frequency (like in English), the second most frequent letter is t in English, n in German, and a in Spanish. In fact, the most frequent letter in Portuguese is a (e is the second).
(data taken from Wikipedia.)

Using Non-uniformity in Letter Distribution for Comprression

An important application of the non uniform frequency of letters is text compression. The Huffman method encodes frequent letters using short binary strings, and infrequent letters using longer binary strings. A typical text contains more frequent letters than infrequent letters. This enable the Huffman method to compress typical, human produced texts and save as much as 40% (compressed text vs. original text).

Why (else) We Care About Frequencies of Letters

- Breaking encryption
- Prove or disprove authorship of texts
- Layout of characters on mechanical typwriters

Not only character frequencies, but also bigram, trigram, word frequencies, word length, and sentence length can be calculated and used for various purposes.

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.

A Demonstration of Huffman Code Construction

Constructing Huffman Tree

Assume that the character count is

[('a', 8), ('b', 3), ('h', 1), ('d', 1),

('e', 1), ('f', 1), ('g', 1), ('c', 1)]

The example is adapted from

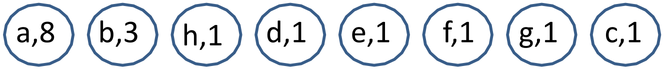
Structure and Interpretation of

Computer Programs,

by Harold Abelson, Gerald Jay Sussman

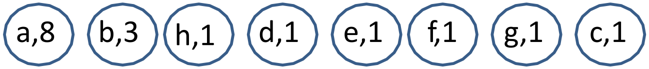
Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

Initialization of the algorithm:
each character is a single-node tree.

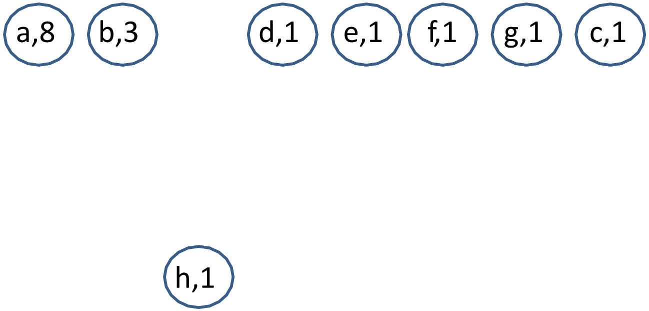


- We insert all these into a priority queue: a data structure that supports the operations insert and extract smallest.
- We use a naive implementation of priority queue: Python's dict (other, more efficient data structures for priority queue will be learned in the data Structures course).

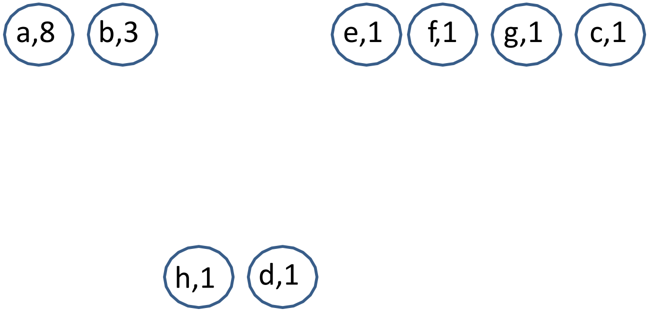
Extract minimum twice:



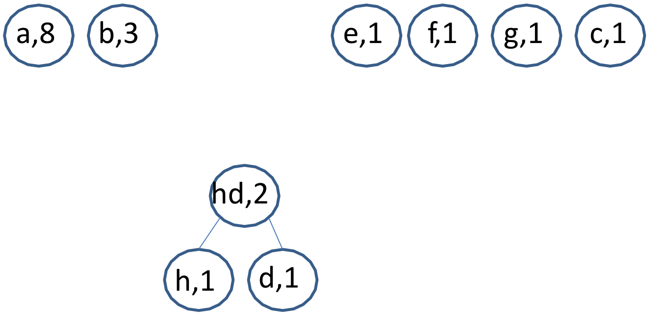
Extract minimum twice:



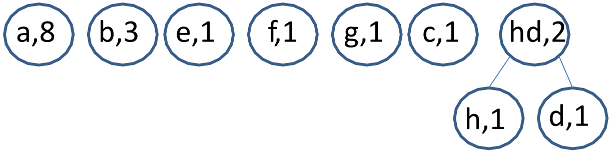
Extract minimum twice:



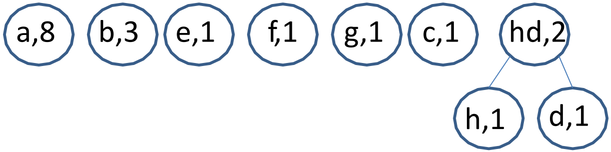
join the 2 extracted minimal nodes into a new tree:



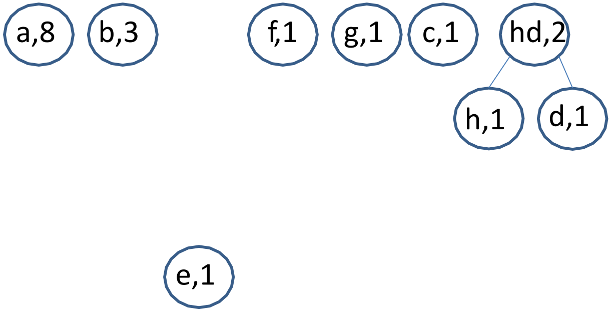
Insert the new tree into the priority queue:



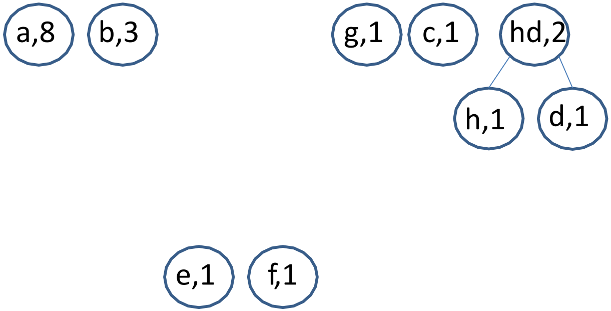
Now repeat the process:

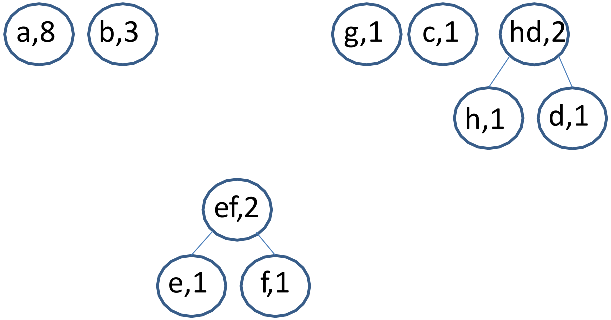


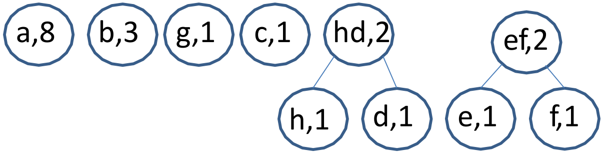
Now repeat the process:

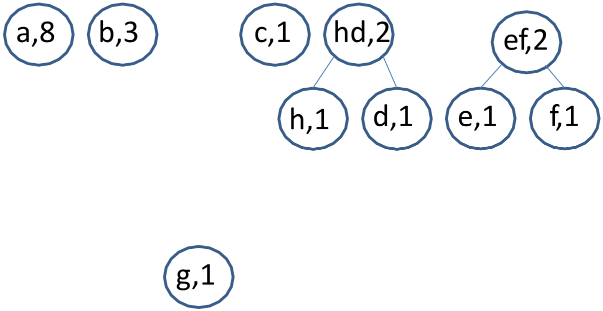


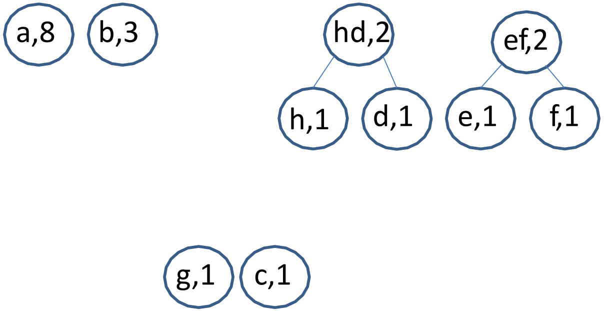
Now repeat the process:

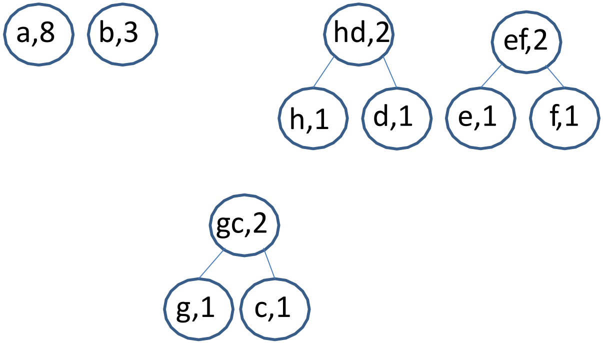


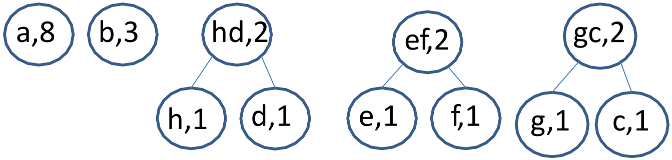

a,8
b,3
g,1
c,1
hd,2
h,1
d,1
ef,2
e,1
f,1

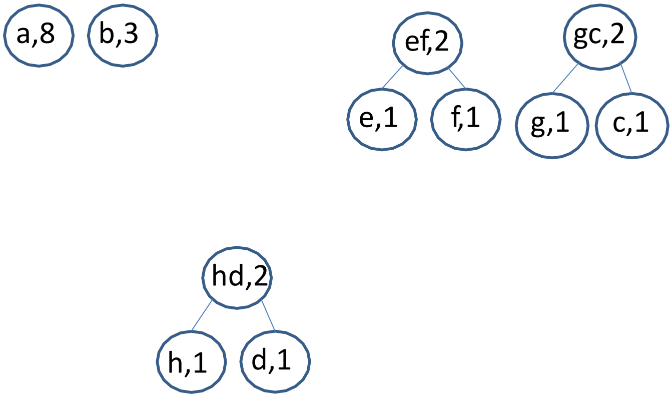

a,8
b,3
g,1
c,1
hd,2
h,1
d,1
ef,2
e,1
f,1

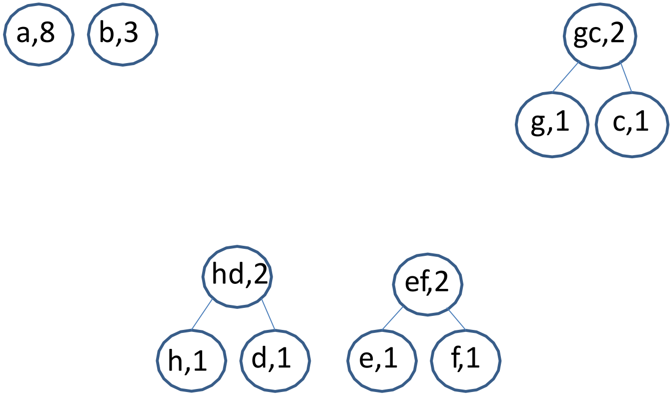

a,8
b,3
c,1
hd,2
h,1
d,1
ef,2
e,1
f,1
g,1

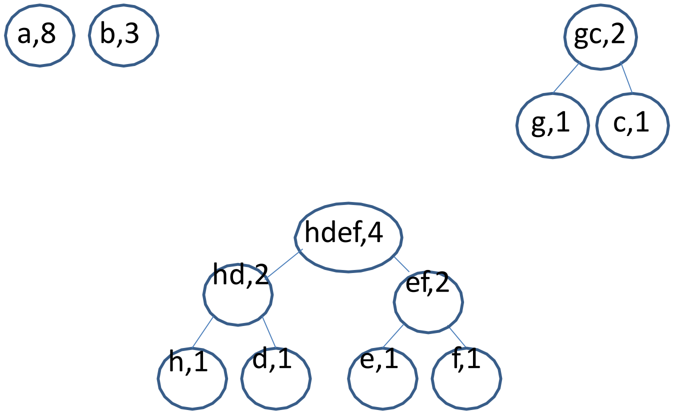

a,8
b,3
hd,2
ef,2
h,1
d,1
e,1
f,1
g,1
c,1

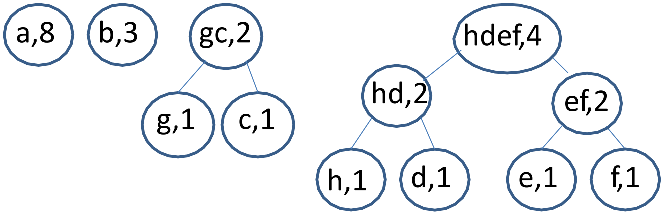

a,8
b,3
hd,2
h,1
d,1
ef,2
e,1
f,1
gc,2
g,1
c,1

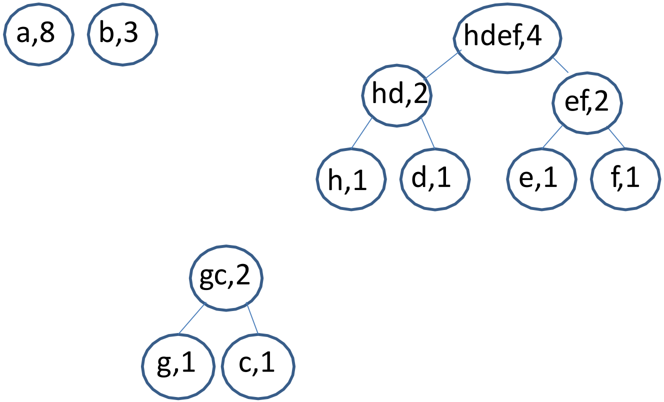

a,8
b,3
hd,2
h,1
d,1
ef,2
e,1
f,1
gc,2
g,1
c,1

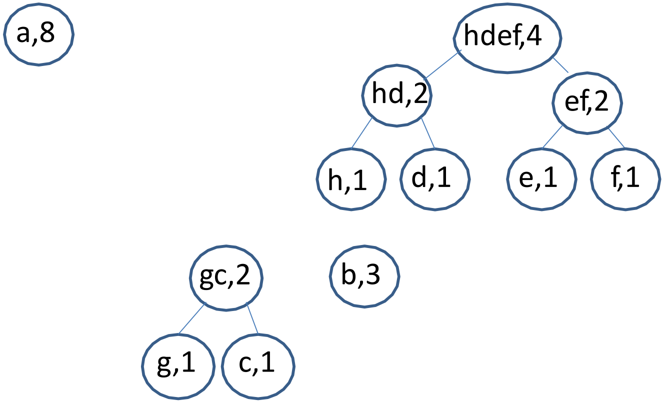

a,8
b,3
ef,2
e,1
f,1
gc,2
g,1
c,1
hd,2
h,1
d,1

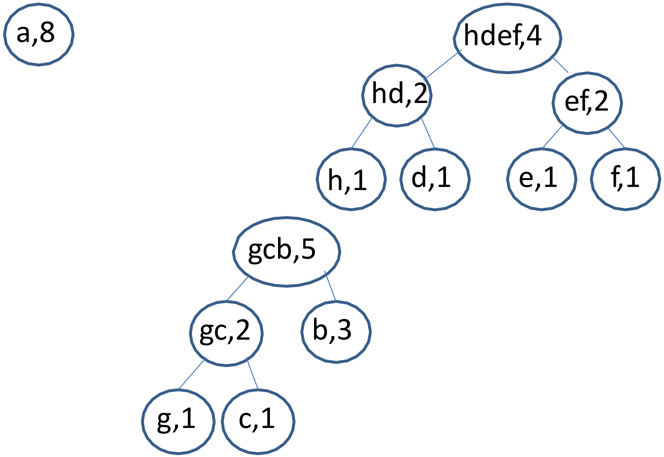

a,8
b,3
gc,2
g,1
c,1
hd,2
h,1
d,1
ef,2
e,1
f,1

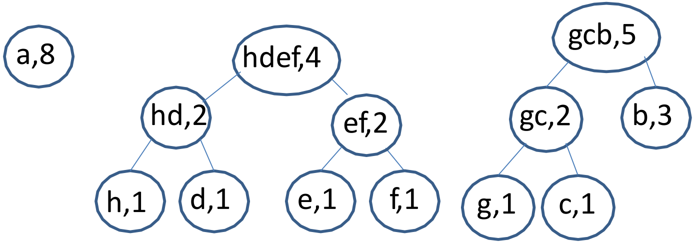

a,8
b,3
gc,2
g,1
c,1
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1

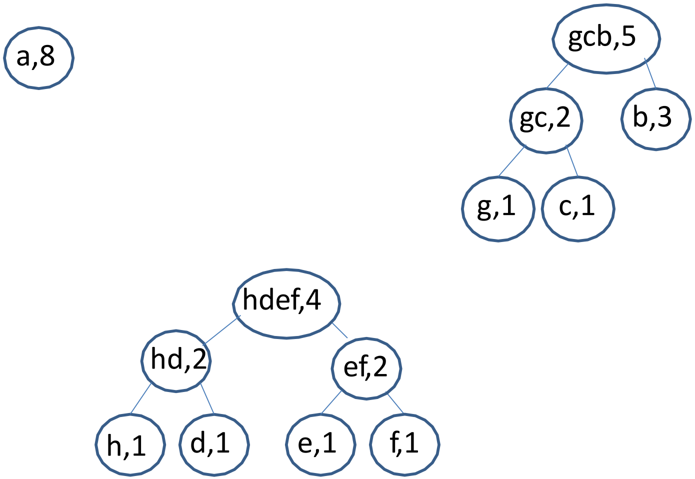

a,8
b,3
gc,2
g,1
c,1
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1

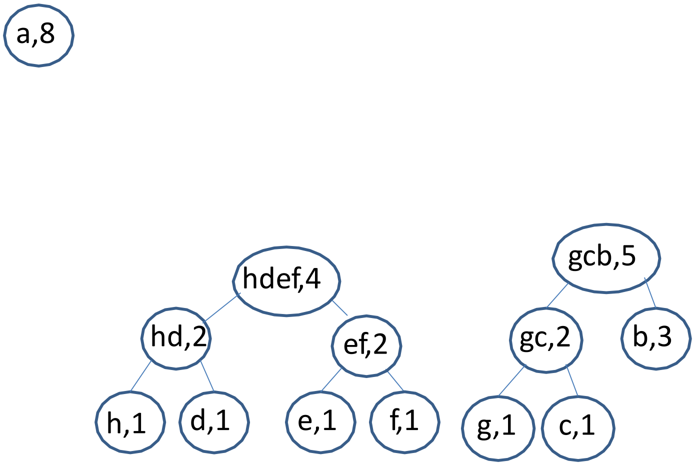

a,8
b,3
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1
gc,2
g,1
c,1

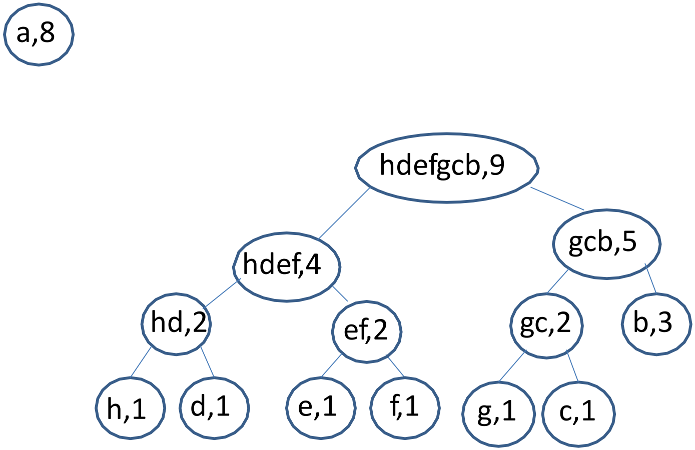

a,8
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1
gc,2
b,3
g,1
c,1

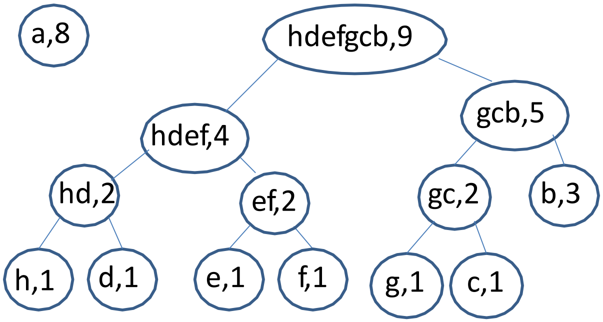

a,8
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1
gcb,5
gc,2
b,3
g,1
c,1

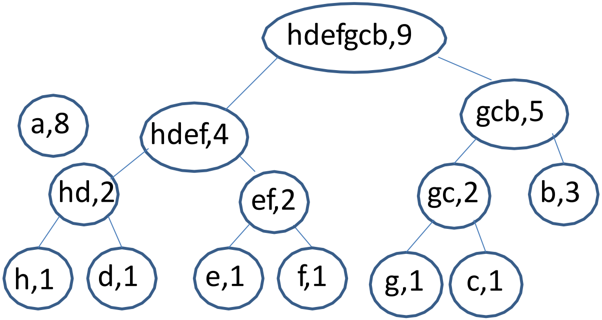

a,8
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1
gcb,5
gc,2
b,3
g,1
c,1

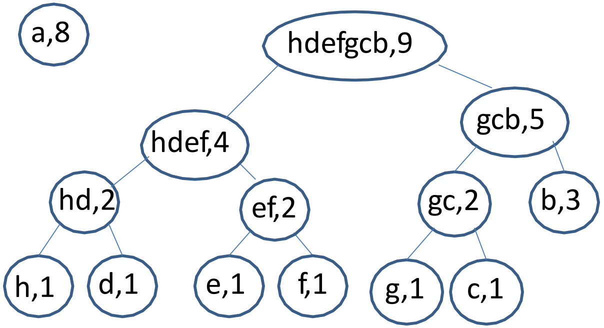

a,8
gcb,5
gc,2
b,3
g,1
c,1
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1


a,8
hdef,4
hd,2
ef,2
h,1
d,1
e,1
f,1
gcb,5
gc,2
b,3
g,1
c,1

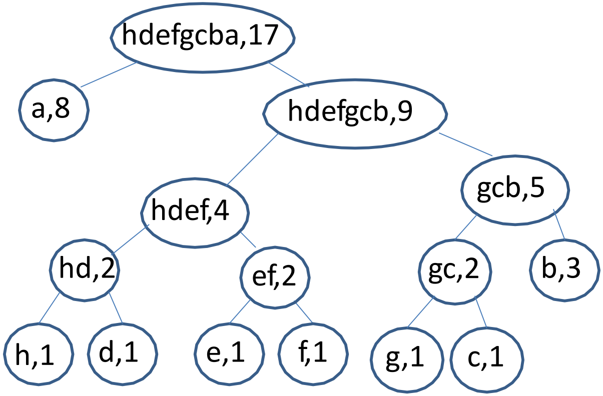

a,8
hdefgcb,9
hdef,4
gcb,5
hd,2
ef,2
gc,2
b,3
h,1
d,1
e,1
f,1
g,1
c,1

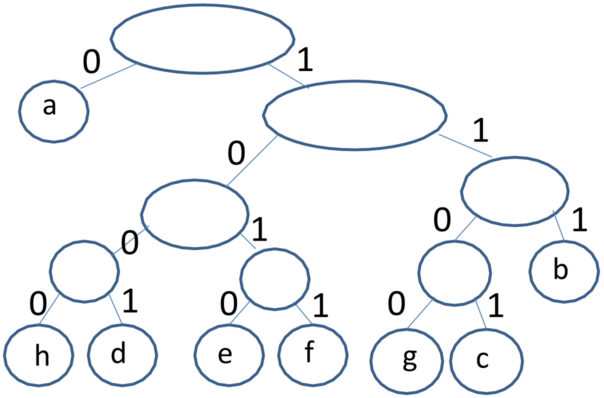

a,8
hdefgcb,9
hdef,4
gcb,5
hd,2
ef,2
gc,2
b,3
h,1
d,1
e,1
f,1
g,1
c,1


hdefgcb,9
a,8
hdef,4
gcb,5
hd,2
ef,2
gc,2
b,3
h,1
d,1
e,1
f,1
g,1
c,1


a,8
hdefgcb,9
hdef,4
gcb,5
hd,2
ef,2
gc,2
b,3
h,1
d,1
e,1
f,1
g,1
c,1

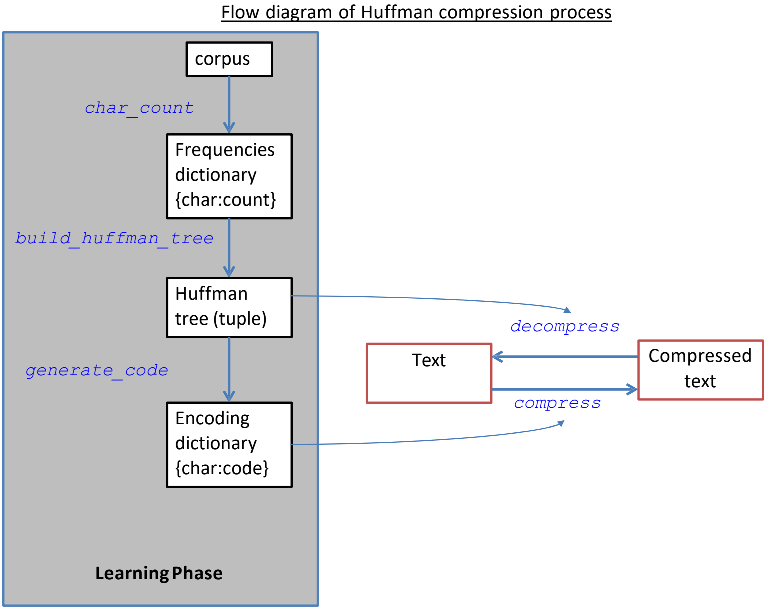
The queue contains a single element. Loop ends



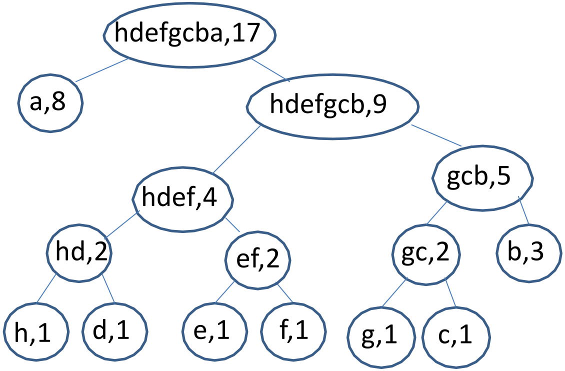


Now we can create the codes for each letter by following the paths from the root to the leaves.
“a”: 0, “h”: 1000, “f”: 1011, ….

Building the Huffman Code: A Road Map



Binary Tree Represented as a Recursive 2-Tuple



```
('a', ((('h', 'd'), ('e', 'f')), (('g', 'c'), 'b')))
```

Huffman Code Construction: 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 (characters to binary strings).

The first step of the road map: char count

We will use the function char count:

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

The first step of the road map

The following small example is 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 (dictionary) to represent letters' counts.
- Initially, each element in the priority queue includes a single character and its count (aka 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, combined 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 (dictionary) with one compound item representing the Huffman tree.

Dict methods pop, popitem and update

```
>>> d = {"a":8, "b":3, "h":1, "d":1, "e":1, "f":1, "g":1, "c":1}

>>> d.pop("c")
1
>>> d
{'a': 8, 'b': 3, 'h': 1, 'd': 1, 'e': 1, 'f': 1, 'g': 1}
>>> d.pop("f")
1
>>> d
{'a': 8, 'b': 3, 'h': 1, 'd': 1, 'e': 1, 'g': 1}

>>> d.update({("c", "f"):2})
>>> d
{'a': 8, 'b': 3, 'h': 1, 'd': 1, 'e': 1, 'g': 1, ('c', 'f'): 2}

>>> d.popitem() #remove last inserted item
(('c', 'f'), 2)
>>> d
{'a': 8, 'b': 3, 'h': 1, 'd': 1, 'e': 1, 'g': 1}
```

Building the Huffman Tree, represented as a Recursive 2-Tuple

```
def build_huffman_tree(char_count_dict):
    """ Recieves dictionary with char:count entries
        Generates Huffman tree represented as a tuple (left, right)
    queue = char_count_dict.copy() #keep input intact

    while len(queue) > 1:
        #print(queue)
        # Extract minimum twice
        A, cntA = extract_min_cnt(queue) # key, val with minimal va
        B, cntB = extract_min_cnt(queue) # next minimal
        # Combine two into one and insert
        AB = (A,B)
        cntAB = cntA + cntB
        queue.update({AB: cntAB})

    # now queue has only one pair {htree: total_cnt}
    htree, total_cnt = queue.popitem() # total count must be sum(qu
    #print(htree)
    return htree   # a tuple representing the tree structure
```

Extract Item with Minimal Count

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

```python
def extract_min_cnt(d):
    min_node = min(d, key = lambda k: d[k])
    min_cnt = d[min_node]
    d.pop(min_node)
    return min_node, min_cnt
```

Building the Huffman Tree: Same Small Example

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

We now build the tree:

```
>>> htree = build_huffman_tree(char_count_dict)
>>> htree
('a', ((('h', 'd'), ('e', 'f')), (('g', 'c'), 'b')))

>>> htree[0]
'a'          # the code for 'a' is 0
>>> htree[1][0][1][0]
'e'          # the code for 'e' is 1010
```

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_hcode(htree, prefix=""):
    """ Receives a Huffman tree (tuple) with embedded encoding,
        and a prefix of encodings.
        Returns a dictionary where characters are
        keys and associated binary strings are values."""

    if isinstance(htree, str): # a leaf in the tree = single char
        return {htree: prefix}
    else:
        left_tree, right_tree = htree[0], htree[1]
        hcode = {}

        hcode.update(generate_hcode(left_tree,  prefix+'0'))
        hcode.update(generate_hcode(right_tree, prefix+'1'))

        return hcode
```

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 dictionaries, and combine them.

A small example:

```
>>> corpus = "aaaaaaaabbbcdefgh"
>>> char_count_dict = char_count(corpus)
>>> htree = build_huffman_tree(char_count_dict)
>>> htree
('a', ((('h', 'd'), ('e', 'f')), (('g', 'c'), 'b')))
>>> generate_code(htree)
{'a': '0', 'h': '1000', 'd': '1001', 'e': '1010', 'f': '1011',
 'g': '1100', 'c': '1101', 'b': '111'}
```

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

Employing Huffman Encoding for String Compression

We go over the text we want to compress, one character at a time.
For each character 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, hcode):
    """ compress text using encoding dictionary """
    assert isinstance(text, str)
    return "".join((hcode[ch] for ch in text))

>>> corpus = "aaaaaaaabbbcdefgh"
>>> hcode = generate_code(build_huffman_tree(char_count(corpus)))
>>> hcode
{'a': '0', 'h': '1000', 'd': '1001', 'e': '1010', 'f': '1011',
 'g': '1100', 'c': '1101', 'b': '111'}
>>> text = "abc"
>>> compress(text, hcode)
'01111101'
```

Decompression using the Huffman Tree: Code

```
def decompress(bits, htree):
    """ get a bit string representing compressed text,
        and the Huffman tree (tuple) used to compress it.
        Return original text """
    node = htree # htree = (htree_left, htree_right)
    result = []

    for bit in bits:
        node = node[int(bit)] #left or right?
        if isinstance(node, str): # leaf in the tree = single char
            result.append(node)
            node = htree # restart, back to root

    return "".join(result)  # converts list of chars to a string
```

Decompression using the Huffman Tree: Example

```
>>> corpus = "aaaaaaaabbbcdefgh"
>>> htree = build_huffman_tree(char_count(corpus))
>>> htree
('a', ((('h', 'd'), ('e', 'f')), (('g', 'c'), 'b')))
>>> hcode = generate_code(htree)
{'a': '0', 'h': '1000', 'd': '1001', 'e': '1010', 'f': '1011',
 'g': '1100', 'c': '1101', 'b': '111'}
>>> text = "abc"
>>> compress(text, hcode)
'01111101'
```

```
>>> decompress('01111101', htree)
'abc'
```

A Comment on the Huffman Tree Structure

Two degrees of freedom:

1) 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.
2) Selecting which of the two minimal elements will be used as the left offspring and who as the right offspring, can also be implementation depedent.

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 (definition coming soon).

The No Compression Alternative: Fixed Length Encoding

For simplicity, let's assume we use only ASCII characters, each requiring 7 bits.

```
def ascii2bits(text):
    """ Translates ASCII text to binary reprersentation using
        7 bits per character. Assume only ASCII chars """
    return "".join([bin(ord(c))[2:].zfill(7) for c in text])

>>> ascii2bit("abc")
'110000111000101100011'
>>> len(_)
21
```

The first parameter to join (an empty string in this case) is the "glue", with which the second parameter is joined.

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(ascii2bits(text)),end="\n\n")
            # == len(text)*7
  print(ascii2bits(text), end="\n\n")
  comp = compress(text, code)
  print("compressed len in bits:", len(comp), end="\n\n")
  print(comp, end="\n\n")
  print("comp.ratio:",len(comp)/len(ascii2bits(text)),end="\n\n")
  #decompression, back to original code
  decomp = decompress(comp, tree)
  print(decomp, end="\n\n")
  assert decomp == 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: 18
100001100000101110
compression ratio: 0.8571428571428571
```

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: 19
0110001011011011110
compression ratio: 0.9047619047619048
```

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_hcode(build_huffman_tree(char_count(corpus)))
    comp = compress(text, code)  #best corpus is text itself
    print("compression ratio:", len(comp) / len(ascii2bits(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?

Huffman Code: Time Complexity

Let *n* be the length of the corpus.
Let *m* be the length of the text to compress.
Let *b* be the length of the bit string after compression.
Assume the size of the alphabet (number of different characters) is O(1).

- char_count takes O(*n*) time on average (each access to the dictionary takes O(1) on average).
- build_huffman_tree takes O(1) time on average (the size of the dictionary is O(1)).
- generate_hcode takes O(1) time on average (tree has O(1) nodes).
- compress takes O(*m*) time on average.
- decompress takes O(*b*) time worst case (each tree move takes O(1) W.C).

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,

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

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 $L_W(C) = \sum_{i=1}^{n} w_i \cdot \text{len}(c_i)$.

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.

Compressing Text Beyond Huffman

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.