CS 315, Day 35 – Huffman's Algorithm and LCS

Who has used the Unix/linux command `gzip`?

What does it do?

How does it work? Does it compress an ***executable*** (`.exe`) file very well?

To store a ***text*** file (comprised of characters) more compactly, we can use a binary character ***code***, representing each character with a unique binary character string. And of course our goal is to use a few bits as possible, both in the code and in the compressed file.

One thing we'll need is how often each character appears in the file, its ***frequency***. Most compression algorithms run through the file multiple times, and one of the early passes will be to determine the frequency.

Example: For a data file of 100,000 characters:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

We could use a ***fixed-length code***. If we do for this example, how many bits are required?

Two bits, $2^2 = 4$ isn't enough, because it could only represent 4 characters and we have 6.

So if we used three bits, $2^3 = 8$, we'll have enough.

One fixed-length code could simply be:

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

For this fixed-length code, we have 3 bits/character, and 100,000 characters, so the file would be 300,000 bits, or 37,500 bytes (37.5 K).

Can we do better?

Yes, for one, we didn't even use the frequency information. So our code should assign shorter bit strings to the more frequently occurring characters. Representing characters with different length bit strings is called a ***variable-length code***. Also, we would like to have an easy way to differentiate when one character ends and another begins. This isn't too difficult for a fixed-length code, but is more challenging in a variable-length code.

We'll restrict our focus to a particular type of variable-length code, called a ***prefix-free code***, which means no bit string is a prefix of any other bit string assigned.

For our example, let's assume that our prefix-free code (which we know is a variable-length code) is:

```
a     0
b     101
c     100
d     111
e     1101
f     1100
```

Use it to decode the following string `1000101`

Since it's a prefix-free code, we can just scan the input bits, and output each character we encounter.

100	0	101
c	a	b

Encode the word ***face*** using this code.

We just concatenate the codeword bit strings for each character:

1100	0	100	1101
f	a	c	e

Which, without spaces, gives `110001001101`

And based on the frequencies, we'll need fewer bits,

$$\begin{array}{rcl} 45{,}000 \cdot 1 & = & 45{,}000 \\ +\,13{,}000 \cdot 3 & = & 39{,}000 \\ +\,12{,}000 \cdot 3 & = & 36{,}000 \\ +\,16{,}000 \cdot 3 & = & 48{,}000 \\ +\,\;9{,}000 \cdot 4 & = & 36{,}000 \\ +\,\;5{,}000 \cdot 4 & = & 20{,}000 \\ \hline & = & 224{,}000 \text{ bits} \end{array}$$

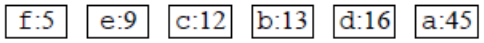
So, how do we construct this prefix-free code? We will construct a prefix tree, which will be binary, and full, using the frequencies. The algorithm we will follow here is called ***Huffman's Algorithm***.

Huffman's algorithm builds the tree from the bottom up. It repeatedly selects two nodes with the lowest frequency, and makes them children of a new node whose frequency is the sum of the two nodes' frequencies.

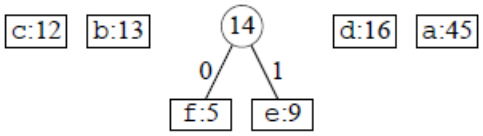
And how can we find the nodes with the minimum frequency? With a min-priority-queue!

```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n − 1
      allocate a new node z
      x = EXTRACT-MIN(Q)
      y = EXTRACT-MIN(Q)
      z.left = x
      z.right = y
      z.freq = x.freq + y.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

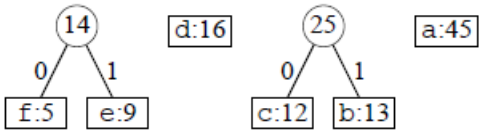
(a)



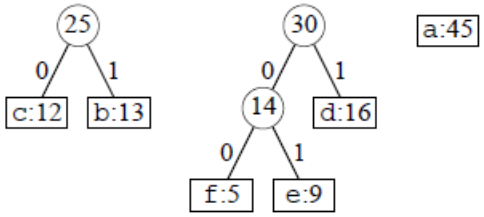
(b)



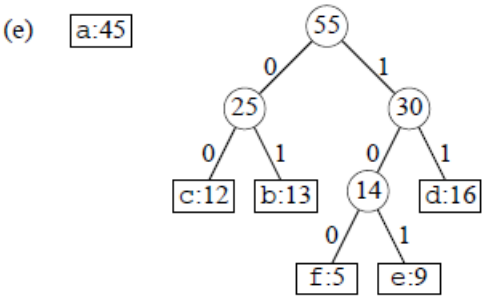
(c)



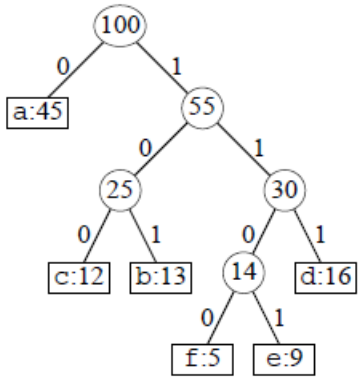
(d)



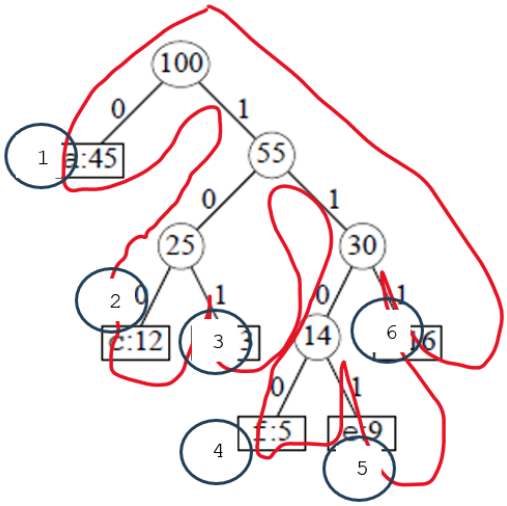
(e)



(f)



Note, in a prefix tree, a preorder traversal will result in ***lexicographic/"phone-book"*** sorted order, remember the edge values.



The first node encountered in the preorder traversal	a	0
The second node encountered in the preorder traversal	c	100
The third node encountered in the preorder traversal	b	101
The fourth node encountered in the preorder traversal	f	1100
The fifth node encountered in the preorder traversal	e	1101
The sixth node encountered in the preorder traversal	d	111

Besides searching for a "complete" match of a pattern P in a string T, oftentimes we need to check for similarity:

- DNA matching
- Changes in a source code library (`git`)
- Changes in a text file library (`diff`)
- Web crawlers (webpage updates)

One way to maximize similarity is to find the Longest Common Subsequence (LCS) between two strings.

Brute Force?

How would this be implemented?

We would need to compare the sub-sequences of X and Y.

What are the total number of sub-sequences?

$Y = <$ **B , D , C , A , B , A** $>$

Y has 6 elements, we'll build the sub-sequences iteratively and look for a pattern.

1 element, **B**, 1 sub-seq: **B**

2 elements, **BD**, 3 sub-seq: **B**, **D**, **BD**

3 elements, **BDC**, 7 sub-seq: **B, D, BD**, **C**, **BC, DC, BDC**

And so on…

N elements leads to $2^N - 1$ sub-sequences.

Brute-force is Exponential!

So we can define the problem,

$X = < x_1 , x_2 , \ldots , x_m >$

$Y = < y_1 , y_2 , \ldots , y_n >$

And let $Z = < z_1 , z_2 , \ldots , z_k >$ be a LCS of X and Y

If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1}

If $x_m \mathrel{!}= y_n$, then $z_k \mathrel{!}= x_m$ implies Z is a LCS of X_{m-1} and Y

If $x_m \mathrel{!}= y_n$, then $z_k \mathrel{!}= y_n$ implies Z is a LCS of X and Y_{n-1}

which leads to the recursive formula

$$c[i,j] = \begin{cases} 0 & \textbf{if } i=0 \text{ or } j=0 \\ c[i-1,j-1] + 1 & \textbf{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \textbf{if } x_i \mathrel{!}= y_j \end{cases}$$

What structure does `c[i,j]` imply?

A matrix

What is the associated running time?

It's at least better than exponential!

$O(m \cdot n)$

The LCS exhibits an optimal substructure. If we did all the work, but then truncated one (or both) input strings, the smaller problem would have a solution. The solution is recursive, although the code isn't. This is an example of a ***Dynamic Programming*** algorithm.

This algorithm will find one LCS, but not all. Switching rows and columns might lead to a different LCS!

```
LCS-Length(X , Y)

  m = X.length
  n = Y.length

  for i = 0 to m
    c[i,0] = 0

  for j = 0 to n
    c[0,j] = 0

  for i = 1 to m
    for j = 1 to n
      if x[i] == y[j]
        c[i,j] = c[i-1 , j-1] + 1
        b[i,j] = "\"
      elseif c[i-1,j] >= c[i , j-1]
        c[i,j] = c[i-1 , j]
        b[i,j] = "|"
      else
        c[i,j] = c[i , j-1]
        b[i,j] = "--"
  return c[] and b[]
```

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0							
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |	
4	B	0							
5	D	0							
6	A	0							
7	B	0							

m=7

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |
4	B	0	1 \	1 |	2 |	2 |	3 \	3 --
5	D	0	1 |	2 \	2 |	2 |	3 |	3 |
6	A	0	1 |	2 |	2 |	3 \	3 |	4 \
7	B	0	1 \	2 |	2 |	3 |	4 \	4 |

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |	
4	B	0	1 \	1 |	2 |	2 |	3 \	3 --	
5	D	0	1 |	2 \	2 |	2 |	3 |	3 |	
6	A	0	1 |	2 |	2 |	3 \	3 |	4 \	A
7	B	0	1 \	2 |	2 |	3 |	4 \	4 |	

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |	
4	B	0	1 \	1 |	2 |	2 |	3 \	3 --	B
5	D	0	1 |	2 \	2 |	2 |	3 |	3 |	
6	A	0	1 |	2 |	2 |	3 \	3 |	4 \	A
7	B	0	1 \	2 |	2 |	3 |	4 \	4 |	

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	B
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |	C
4	B	0	1 \	1 |	2 |	2 |	3 \	3 --	B
5	D	0	1 |	2 \	2 |	2 |	3 |	3 |	
6	A	0	1 |	2 |	2 |	3 \	3 |	4 \	A
7	B	0	1 \	2 |	2 |	3 |	4 \	4 |	

m=7

	j	0	1	2	3	4	5	6	n=6
i		y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0	
1	A	0	0 |	0 |	0 |	1 \	1 --	1 \	
2	B	0	1 \	1 --	1 --	1 |	2 \	2 --	B
3	C	0	1 |	1 |	2 \	2 --	2 |	2 |	C
4	B	0	1 \	1 |	2 |	2 |	3 \	3 --	B
5	D	0	1 |	2 \	2 |	2 |	3 |	3 |	
6	A	0	1 |	2 |	2 |	3 \	3 |	4 \	A
7	B	0	1 \	2 |	2 |	3 |	4 \	4 |	

m=7

B C B A

Try the algorithm on a smaller problem:

i	j	0	1	2	3
		y_j	B	D	C
0	x_i	0	0	0	0
1	A	0			
2	B	0			
3	C	0			

i	j	0	1	2	3
		y_j	B	D	C
0	x_i	0	0	0	0
1	A	0	0 |	0 |	0 |
2	B	0			
3	C	0			

i	j	0	1	2	3
		y_j	B	D	C
0	x_i	0	0	0	0
1	A	0	0 |	0 |	0 |
2	B	0	1 \	1 --	1 --
3	C	0			

i	j	0	1	2	3
		y_j	B	D	C
0	x_i	0	0	0	0
1	A	0	0 |	0 |	0 |
2	B	0	1 \	1 --	1 --
3	C	0	1 |	1 |	2 \

We know the LCS for these two strings will be length 2. Starting at the bottom right, "take" the character from any cell with the \.

So here, the LCS would be BC.