DATA COMPRESSION: PART I

Contents of today's lecture:

- Introduction to Data Compression
- Techniques for Data Compression
  - Compact Notation
  - Run-length Encoding
  - Variable-length codes: Huffman Code

References:

FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Section 6.2 (Data Compression).
CORMEN, LEISERSON, RIVEST AND STEIN, Introduction to Algorithms, 2001, 2nd ed. Section 16.3 (Huffman codes).

Data Compression = Encoding the information in a file in such a way that it takes less space.

Using Compact Notation

Ex: File with fields: lastname, province, postal code, etc.
Province field uses 2 bytes (e.g. 'ON', 'BC') but there are only 13 provinces and territories which could be encoded by using only 4 bits (compact notation).

16 bits are encoded by 4 bits (12 bits were redundant, i.e. added no extra information)

Disadvantages:

- The field "province" becomes unreadable by humans.
- Time is spent encoding ('ON' $\rightarrow$ 0001) and decoding (0001 $\rightarrow$ 'ON').
- It increases the complexity of software.

Run-length Encoding

Good for files in which sequences of the same byte may be frequent.

Example: Figure 6.1 in page 205 of the textbook: image of the sky.

- A pixel is represented by 8 bits.
- Background is represented by the pixel value 0.

The idea is to avoid repeating, say, 200 bytes equal to 0 and represent it by (0, 200).

If the same value occurs more than once in succession, substitute by 3 bytes:

- a special character - run length code indicator (use 1111 1111 or FF in hexadecimal notation)
- the pixel value that is repeated (FF is not a valid pixel anymore)
- the number of times the value is repeated (up to 256 times)

Encode the following sequence of Hexadecimal bytes:

```
22 23 24 24 24 24 24 24 24 25
26 26 26 26 26 26 25 24
```

Run-length encoding:

```
22 23 FF 24 07 25 FF 26 06 25 24
```

18 bytes reduced to 11.

Variable-Length Codes and Huffman Code

Example of a variable-length code:

Morse Code (two symbols associated to each letter)

```
A    ._
B    _...
...
E    .
F    .._.
...
T    _
U    .._
...
```

Since E and T are the most frequent letters, they are associated to the shortest codes (. and - respectively)

Huffman Code

Huffman Code is a variable length code, but unlike Morse Code the encoding depends on the frequency of letters occurring in the data set.

Example of Huffman Code:
Suppose the file content is:

I		A	M		S	A	M	M	Y

Total: 10 characters

Letter	A	I	M	S	Y	/b
Frequency	2	1	3	1	1	2
Code	00	1010	11	1011	100	01

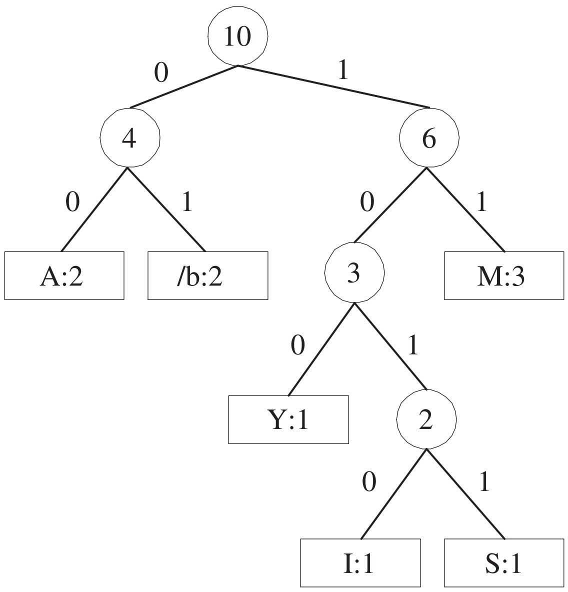
Huffman Code is a prefix code: no codeword is a prefix of any other.
(we are representing the space as "/b")

Encoded message:

`1010010011011011001111100`

25 bits rather than 80 bits (10 bytes)!

Huffman Tree (for easy decoding)



Consider the encoded message:

`101001001101...`

- Interpret the 0's as "go left" and the 1's as "go right".
- A **codeword** for a character corresponds to the path from the root of the Huffman tree to the leaf containing the character.

Following the labeled edges in the Huffman tree we decode the above message.

```
1010    leads us to I
01      leads us to /b
00      leads us to A
11      leads us to M
01      leads us to /b
etc.
```

Properties of Huffman Tree

- Every internal node has 2 children;
- Smaller frequencies are further away from the root;
- The 2 smallest frequencies are siblings;
- The number of bits required to encode the file is minimized:

$$B(T) = \sum_{(c \in C)} f(c).d_T(c),$$

where:
$B(T)$ = number of bits needed to encode the file using tree T,
$f(c)$ = frequency of character c,
$d_T(c)$ = length of the codeword for character c.

In our example:

$$B(T) = 2 \times 2 + 1 \times 4 + 3 \times 2 + 1 \times 4 + 1 \times 3 + 2 \times 2 = 25$$

What is the average number of bits per encoded letter ?

Average number of bits per letter =
= B(T)/total number of characters = 25/10 = 2.5

The way Huffman Tree is constructed guarantees that B(T) is as small as possible!

How is the Huffman Tree constructed ?

The weight of a node is the total frequency of characters under the subtree rooted at the node.

Originally, form subtrees which represent each character with their frequencies as weights.

The algorithm employs a **Greedy Method** that always merges the subtrees of smallest weights forming a new subtree whose root has the sum of the weight of its children.

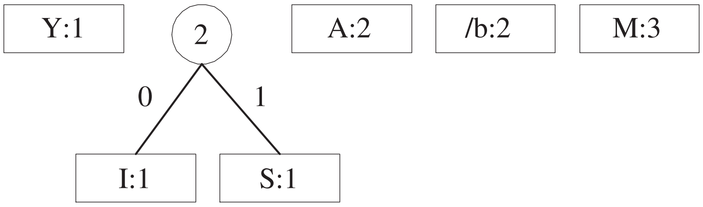
The algorithm in action

Using the letters and frequencies from the previous example:

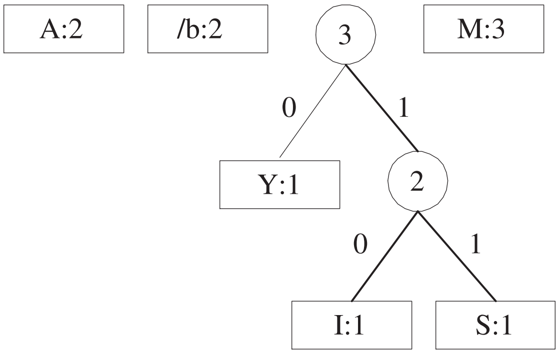
Subtrees: I:1 S:1 Y:1 A:2 /b:2 M:3

Merge the two subtrees of smallest weight (break ties arbitrarily) ...

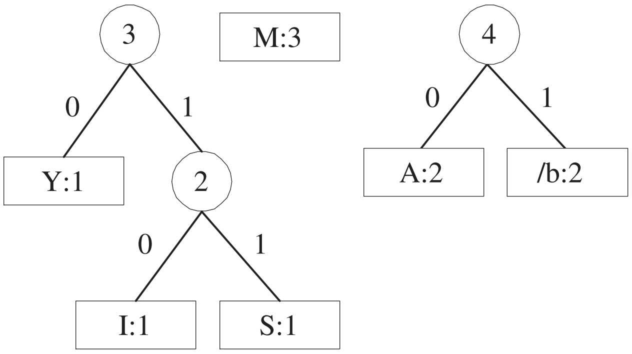
Subtrees:



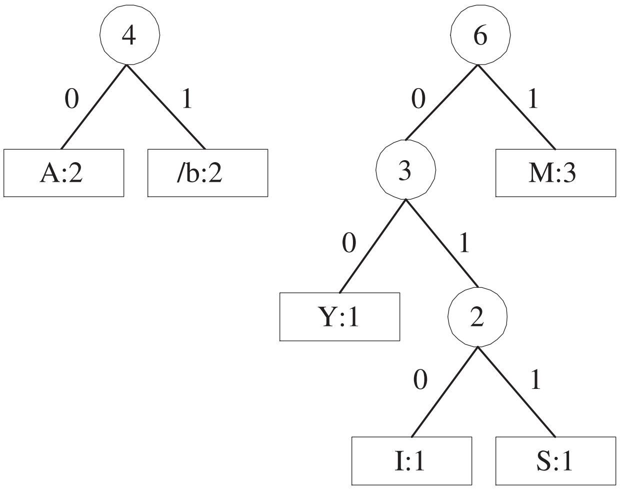
Subtrees:



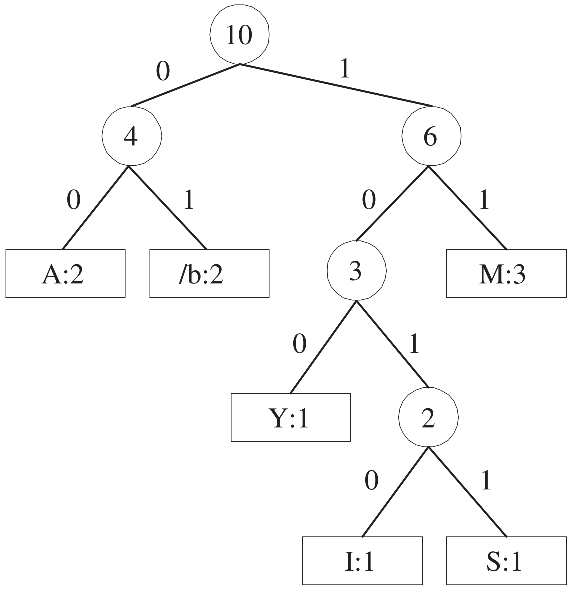
Subtrees:



Subtrees:



Final Tree:



Pseudo-Code for Huffman Algorithm:

A **priority queue** `Q` is used to identify the smallest-weight subtrees to merge. A priority queue provides the following operations:

- `Q.insert(x)`: insert `x` to `Q`
- `Q.minimum()`: returns element of smallest key
- `Q.extract-min()`: removes and returns the element with smallest key

Possible implementations of a priority queue:
Linked lists: Each of the three operations can be done in O(n)
Heaps: Each of the three operations can be done in $O(\log_n)$

Pseudo-Code: `Huffman`
Input: characters and their frequencies
`(c1, f[c1]), (c2, f[c2]), ..., (cn, f[cn])`
Output: returns the Huffman Tree

```
Make priority queue Q using c1, c2, ..., cn;
 for i = 1 to n - 1 do {
     z = allocate new node;
     l = Q.extract-min();
     r = Q.extract-min();
     z.left = l;
     z.right = r;
     f[z] = f[r] + f[l];
     Q.insert(z);
 }
 return Q.extract-min();
```

What is the running time of this algorithm if the priority queue is implemented as a ...

1. **Linked List ?**
   - `Make priority queue` takes $O(n)$.
   - `extract-min` and `insert` takes $O(n)$.
   - Loop iterates $n-1$ times.

   Total time: $O(n^2)$

2. **Heap (Array Heap) ?**
   - `Make priority queue` takes $O(n \cdot \log n)$ or $O(n)$.
   - `extract-min` and `insert` takes $O(\log n)$.
   - Loop iterates $n-1$ times.

   Total time: $O(n \cdot \log n)$.

- **Pack** and **unpack** commands in Unix use Huffman Codes byte-by-byte.
- They achieve 25 - 40% reduction on text files, but is not so good for binary files that have more uniform distribution of values.

DATA COMPRESSION: PART II

Contents of today's lecture:

- Techniques for Data Compression
  - Lempel-Ziv codes.

Reference: This notes.

Lempel-Ziv Codes

There are several variations of Lempel-Ziv Codes. We will look at LZ78.

Ex: Commands `zip` and `unzip` and Unix `compress` and `uncompress` use Lempel-Ziv codes.

Let us look at an example for an alphabet having only two letters:

`aaababbbaaabaaaaaaabaabb`

Rule : Separate this stream of characters into pieces of text, so that each piece is the shortest string of characters that we have not seen yet.

`a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb`

1. We see `"a"`.
2. `"a"` has been seen, we now see `"aa"`.
3. We see `"b"`.
4. `"a"` has been seen, we now see `"ab"`.
5. `"b"` has been seen, we now see `"bb"`.
6. `"aa"` has been seen, we now see `"aaa"`.
7. `"b"` has been seen, we now see `"ba"`.
8. `"aaa"` has been seen, we now see `"aaaa"`.
9. `"aa"` has been seen, we now see `"aab"`.
10. `"aab"` has been seen, we now see `"aabb"`.

Note that this is a dynamic method!

Index the pieces from 1 to n. In the previous example:

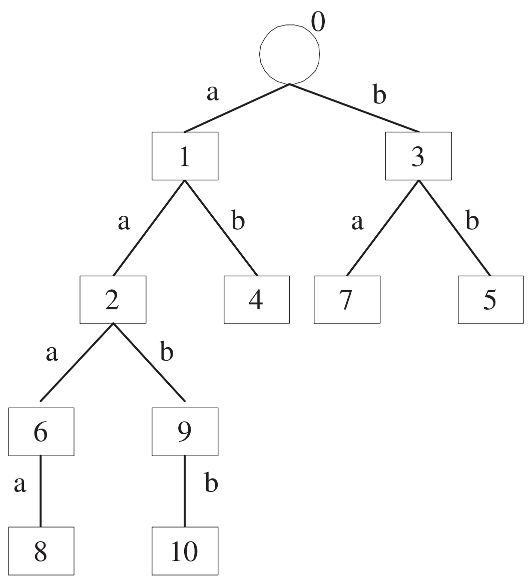
```
Index : 0 1 2  3 4  5  6   7  8    9   10
        0|a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb
```

0 = Null string

Encoding :

```
Index : 1  2  3  4  5  6  7  8  9  10
        0a|1a|0b|1b|3b|2a|3a|6a|2b|9b
```

Since each piece is the concatenation of a piece already seen with a new character, the message can be encoded by a previous index plus a new character.
Indeed a digital tree can be built when encoding:



When a node is inserted the code for the current piece becomes the parent node combined with the new character.
Note that this tree is not binary in general. Here, it is binary because the alphabet has only 2 letters.

Practice Exercises

Encode (using Lempel-Ziv) the file containing the following characters, drawing the corresponding digital tree:

`"aaabbcbcdddeab"`

`"I AM SAM. SAM I AM."`

Bit Representation of Coded Information

How many bits are necessary to represent each integer with index n ? The integer is at most $n-1$, so the answer is: at most the number of bits to represent the number $n-1$.

```
1   2   3   4   5   6   7   8   9   10
0a|1a|0b|1b|3b|2a|3a|6a|2b|9b
```

Index 1: no bit (always start with zero)
Index 2: at most 1, since previous index can be only 0 or 1.
Index 3: at most 2, since previous index is between 0-2.
Index 4: at most 2, since previous index is between 0-3.
Index 5-8: at most 3, since previous index is between 0-7
Index 9-16: at most 4, since previous index is between 0-15

Each letter is represented by 8 bits. Each index is represented using the largest number of bits possibly required for that position. For the previous example, this representation would be as follows:

```
<a>1<a>00<b>01<b>011<b>010<a>011<a>110<a>0010<b>1001<b>
```

Note that `<a>` and `<b>` above should be replaced by the ASCII code for `a` and `b`, which uses 8 bits. We didn't replace them for clarity and conciseness.

Total number of bits in the encoded example :
$(10 \times 8) + (0 + 1 + 2 \times 2 + 4 \times 3 + 2 \times 4) = 105$ bits

The original message was represented using $24 \times 8 = 192$ bits.

Decompressing

```
1   2   3   4   5   6   7   8   9   10
0a|1a|0b|1b|3b|2a|3a|6a|2b|9b


--------------------------
|    |previous |added    |
|    |pointer  |character|
--------------------------
| 0  |    -    |    -    |
| 1  |    0    |    a    |
| 2  |    1    |    a    |
| 3  |    0    |    b    |
| 4  |    1    |    b    |
| 5  |    3    |    b    |
| 6  |    2    |    a    |
| 7  |    3    |    a    |
| 8  |    6    |    a    |
| 9  |    2    |    b    |
| 10 |    9    |    b    |
--------------------------
```

As the table is constructed line by line, we are able to decode the message by following the pointers to previous indexes which are given by the table. Try it, and you will get:

```
a aa b ab bb aaa aaa ba aaaa aab aabb
```

Decode the following Lempel-Ziv encoded file:

`|0M|0A|0K|0E|0 |0L|2K|4 |0F|7E|`

decoded message:

number of bits in original message:

number of bits in encoded message:

Decode the following Lempel-Ziv encoded file:

`|0T|0H|0A|1 |0S|3M|0 |0I|7A|0M|0,|1H|3T|4S|6 |8 |6!|`

decoded message:

number of bits in original message:

number of bits in encoded message:

Irreversible Compression

All previous techniques : we preserve all information in the original data.

Irreversible compression is used when some information can be sacrificed.

Example :
Shrinking an image from 400-by-400 pixels to 100-by-100 pixels.
1 pixel in the new image for each 16 pixels in the original message.

It is less common than reversible compression.

Final Notes

In UNIX:

- **pack** and **unpack** use Huffman codes byte-by-byte.
25-40% for text files, much less for binary files (more uniform distribution)

- **compress** and **uncompress** use Lempel-Ziv.