Representing Things With Bits

Introduction

I said above, that what a bit pattern meant depended upon the context. We often want to represent things like characters and numbers in a bit pattern so that we can conveniently store, manipulate and retrieve them from a computer system. To do that we must have some agreed way of encoding these entities in bit patterns. This section discusses some of the possibilities.

Character Codes

The easiest example of encoding to start with is the example of text characters (English ones!). In order to encode these we need to encode upper case (26), lower case (26), digits (10), brackets and punctuation (20) and odd bits (33). The total of "printable" characters is 95 or thereabouts (depends what you encode under "odds") plus some "unprintable" or control characters (e.g. "space", "carriage return", "line feed") and so on. This conveniently brings the total to 128 (actually the total has to be 2^n) which form a "7-bit code" ($2^7 = 128$). The most common 7-bit code is ASCII which stands for "American Standard Code for Information Interchange". If you add parity to ASCII it becomes an 8-bit code.

For example:

Character	7-bit ASCII	7-bit ASCII + odd parity
T	54(h) 1010100(b)	54(h) 01010100(b)
h	68(h) 1101000(b)	68(h) 01101000(b)
e	65(h) 1100101(b)	D5(h) 11100101(b)

If you want to use more than 128 things, then you need an 8-bit code of which "EBCDIC" (Extended Binary Coded Decimal Interchange Code) and the "IBM Character Set" used in PCs are the prime examples.

For example:

Character	**EBCDIC**
T	E3(h) 11100011(b)
h	88(h) 10001000(b)
e	85(h) 10000101(b)

If you want to encode fewer things then you can use fewer bits. For example "Baudot code" uses five bits to represent text. Those of you who are astute will realise that this isn't enough to get even upper case + figures encoded (26 + 10 > 32) so there are a couple of "shift" characters which are used to change the representation of all subsequent text until an "unshift" appears. This is exactly the same as a "caps lock" on a typewriter. You obviously keep your commonly used characters in the unshifted set and then put the rarer ones in the shifted set.

For example:

Character	**5-bit BAUDOT**	**Comment**
A	03(h) 00011(b)	
N	0C(h) 01100(b)	
S	05(h) 00101(b)	
	1B(h) 11011(b)	shift to FIGS
=	1D(h) 11110(b)	
1	17(h) 10111(b)	
	1F(h) 11111(b)	shift to LETTERS
V	1E(h) 11110(b)	

One further thing to note about codes is "distance". If a character goes wrong then it probably isn't too serious. Txis sextence is readaxle exen txouxh some of txe letxxrs axe gxrbled. However numbers are non-negotiable. Therefore in the "best" codes changes in a single bit of a number representation do not produce another number - they produce something else. Number errors are therefore automatically flagged. None of the above codes to that - which is a general defect.

Code (hex)	**Letters**	**CCITT Alphabet #2**
00	Blank	Blank
01	E	3
02	L. Feed	L.Feed
03	A	-
04	Space	Space
05	S	'
06	I	8
07	U	7
08	C. Retn.	C. Retn.
09	D	WRU
0A	R	4
0B	J	Bell
0C	N	,
0D	F	
0E	C	:
0F	K	(
10	T	5
11	Z	+
12	L	)
13	W	2
14	H	
15	Y	6
16	P	0
17	Q	1
18	O	9
19	B	?
1A	G	
1B	Figs.	Figs.
1C	M	.
1D	X	/
1E	V	=
1F	Letters	Letters

5-bit (Telex) Codes - The "letters" column is the basic character set. The other column is the shifted set activated by the "Figs" code 1B(h). "Letters" is regained with code 1F(h).

Code	Char	Comment	Code	Char	Comment	Code	Char	Comment	Code	Char	Comment
00	NUL	Blank	20	SP	Space	40	@		60	`	
01	SOH	Start of Header	21	!		41	A		61	a	
02	STX	Start of Text	22	"		42	B		62	b	
03	ETX	End of Text	23	#		43	C		63	c	
04	EOT	End of Transmission	24	$		44	D		64	d	
05	ENQ	Enquiry	25	%		45	E		65	e	
06	ACK	Acknowledge (positive)	26	&		46	F		66	f	
07	BEL	Bell	27	'	Close Single Quote	47	G		67	g	
08	BS	Backspace	28	(		48	H		68	h	
09	HT	Horizontal Tab	29	)		49	I		69	i	
0A	LF	Line Feed	2A	*		4A	J		6A	j	
0B	VT	Vertical Tab	2B	+		4B	K		6B	k	
0C	FF	Form Feed	2C	,	Comma	4C	L		6C	l	
0D	CR	Carriage Return	2D	-	Hyphen	4D	M		6D	m	
0E	SO	Shift Out	2E	.	Period	4E	N		6E	n	
0F	SI	Shift In	2F	/		4F	O		6F	o	
10	DLE	Data Link Escape	30	0		50	P		70	p	
11	DC1	Device Control 1	31	1		51	Q		71	q	
12	DC2	Device Control 2	32	2		52	R		72	r	
13	DC3	Device Control 3	33	3		53	S		73	s	
14	DC4	Device Control 4	34	4		54	T		74	t	
15	NAK	Acknowledge (negative)	35	5		55	U		75	u	
16	SYN	Synchronisation	36	6		56	V		76	v	
17	ETB	Eind of Text Block	37	7		57	W		77	w	
18	CAN	Cancel	38	8		58	X		78	x	
19	EM	End of Medium	39	9		59	Y		79	y	
1A	SUB	Substitute	3A	:		5A	Z		7A	z	
1B	ESC	Escape	3B	;		5B	[		7B	{	
1C	FS	File Separator	3C	<		5C	\	Reverse Slant	7C	|	Vertical Line
1D	GS	Group Separator	3D	=		5D	]		7D	}	
1E	RS	Record Separator	3E	>		5E	^	Circumflex	7E	~	Tilde
1F	US	Unit Separator	3F	?		5F	_	Underline	7F	DEL	Delete/ Rubout

7-bit American Standard Code for Information Interchange (ASCII)
The most common character representation in use.

© *James R. Drummond - September 1997*

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61		81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	EOB/ ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	PRE/ ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09	RLF	29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	|	8A		AA		CA		EA	
0B	VT	2B		4B	.	6B	'	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	-	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30	0	50	&	70		90		B0		D0	}	F0	0
11	DC1	31	1	51		71		91	j	B1		D1	J	F1	1
12	DC2	32	2	52		72		92	k	B2		D2	K	F2	2
13	DC3	33	3	53		73		93	l	B3		D3	L	F3	3
14	RES	34	4	54		74		94	m	B4		D4	M	F4	4
15	NL	35	5	55		75		95	n	B5		D5	N	F5	5
16	BS	36	6	56		76		96	o	B6		D6	O	F6	6
17	IL	37	7	57		77		97	p	B7		D7	P	F7	7
18	CAN	38	8	58		78		98	q	B8		D8	Q	F8	8
19	EM	39	9	59		79	\	99	r	B9		D9	R	F9	9
1A	CC	3A	:	5A	!	7A	:	9A		BA		DA		FA	
1B		3B	;	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	<	5C	*	7C	@	9C		BC		DC		FC	
1D	IGS	3D	=	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E	>	5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	?	5F	¬	7F	"	9F		BF		DF		FF	

8-bit Extended Binary Coded Decimal Interchange Code (EBCDIC). EBCDIC is obsolete.

© James R. Drummond - September 1997

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
00		20	SP	40	@	60	`	80	Ç	A0	á	C0	.	E0	α
01	☺	21	!	41	A	61	a	81	ü	A1	í	C1	2	E1	ß
02	☻	22	"	42	B	62	b	82	é	A2	ó	C2	0	E2	Γ
03	♥	23	#	43	C	63	c	83	â	A3	ú	C3	/	E3	π
04	♦	24	$	44	D	64	d	84	ä	A4	ñ	C4	)	E4	Σ
05	♣	25	%	45	E	65	e	85	à	A5	Ñ	C5	3	E5	σ
06	♠	26	&	46	F	66	f	86	å	A6	ª	C6	G	E6	µ
07	●	27	'	47	G	67	g	87	ç	A7	º	C7	K	E7	τ
08	◘	28	(	48	H	68	h	88	ê	A8	¿	C8	9	E8	Φ
09	○	29	)	49	I	69	i	89	ë	A9	⌐	C9	6	E9	Θ
0A	◙	2A	*	4A	J	6A	j	8A	è	AA	¬	CA	=	EA	Ω
0B	♂	2B	+	4B	K	6B	k	8B	ï	AB	½	CB	;	EB	δ
0C	♀	2C	,	4C	L	6C	l	8C	î	AC	¼	CC	:	EC	∞
0D	♪	2D	-	4D	M	6D	m	8D	ì	AD	¡	CD	4	ED	φ
0E	♫	2E	.	4E	N	6E	n	8E	Ä	AE	«	CE	>	EE	ε
0F	☼	2F	/	4F	O	6F	o	8F	Å	AF	»	CF	N	EF	∩
10	►	30	0	50	P	70	p	90	É	B0	!	D0	J	F0	≡
11	◄	31	1	51	Q	71	q	91	æ	B1	"	D1	L	F1	±
12	↕	32	2	52	R	72	r	92	Æ	B2	#	D2	H	F2	≥
13	‼	33	3	53	S	73	s	93	ô	B3	*	D3	F	F3	≤
14	¶	34	4	54	T	74	t	94	ö	B4	1	D4	B	F4	⌠
15	§	35	5	55	U	75	u	95	ò	B5	I	D5	?	F5	⌡
16	▬	36	6	56	V	76	v	96	û	B6	M	D6	C	F6	÷
17	↨	37	7	57	W	77	w	97	ù	B7	D	D7	0	F7	≈
18	↑	38	8	58	X	78	x	98	ij	B8	@	D8	P	F8	°
19	↓	39	9	59	Y	79	y	99	Ö	B9	<	D9	-	F9	·
1A	→	3A	:	5A	Z	7A	z	9A	Ü	BA	5	DA	+	FA	·
1B	←	3B	;	5B	[	7B	{	9B	¢	BB	7	DB	$	FB	√
1C	∟	3C	<	5C	\	7C	|	9C	£	BC	8	DC	(	FC	^{n}
1D	↔	3D	=	5D	]	7D	}	9D	¥	BD	E	DD	%	FD	^{2}
1E	▲	3E	>	5E	^	7E	~	9E	₧	BE	A	DE	'	FE	■
1F	▼	3F	?	5F	_	7F	⌂	9F	ƒ	BF	,	DF	&	FF	

8-bit Character set used internally in an IBM PC and now externally as a character set. Note that the codes are use to represent more than just characters - graphics and technical symbols are included. The codes from 20(h) to 7E(h) have the same interpretation as the 7-bit ASCII set

Integer Numbers

Not everybody wants to represent numbers by their decimal digits - it is wasteful of space and in computer-to-computer communications, two conversions are necessary.

If we are to represent integers in the computer, the first question to settle is the one of how to represent negative numbers. There are two significant ways of doing this:

- Store the absolute value and store the sign separately (since the sign can only take two values, only one bit is required for the sign)
- Store the number in 2's complement form.

The (almost) universal choice in computers is to store the number in 2's complement form. The reason for this is that to add two 2's complement numbers together including getting the sign of the answer right involves simply adding the two numbers together as though they were simple binary numbers.

The rule for a 2's complement number is as follows: A positive number is represented as the binary equivalent. A negative number is represented by taking the binary equivalent of the absolute value, reversing (complementing) all the bits and then adding one to the result. The effect of this is to retain 0 as "all 0s" and to make -1 equal to "all 1s". The most negative number is represented by a 1 in the MSB, a 1 in the Lsb and zeros otherwise. The most positive number is represented by a 0 in the MSB and 1s elsewhere. If we start from the most negative number and then "count" by incrementing the number in a logical manner, we progress through -1 to zero and thence to the maximal positive number before "wrapping round" and starting again.

Here are some examples being represented as an 8-bit number

Number	Representation
17	00010001
1	00000001
126	01111110
127	01111111

© *James R. Drummond - September 1997*

Number	**Binary of absolute value**	**Complement bits**	**Add one to result**
-17	00010001	11101110	11101111
-1	00000001	11111110	11111111
-126	01111110	10000001	10000010
-127	01111111	10000000	10000001

The number 128 in this example would be 10000000 but that is also the representation of the number -128 so there is an ambiguity. In fact we can only represent numbers in the range -127 → 127 with 8 bits. The status of 128/-128 is ambiguous here and it is best left out of the range. If it is required to have one of them in the range than -128 fits best because then a “1" in the most significant bit always indicates a negative number (but it is not strictly a sign bit). You might like to work out why there is always going to be a difference in the positive and negative ranges in such a system whatever the length of the representation so long as all the codes are assigned.

A particular point to watch is that some machines store the two bytes of a 16-bit number in a different order to others The two great camps here are those who store numbers as "least significant byte in least significant address" - which is sometimes referred to as "INTEL format" - and those who store them the other way round - "Motorola format". As an example consider the sequence 0100(h), 0101(h), 0102(h) stored in memory locations 300(h)-305(h):

Address	300(h)	301(h)	302(h)	303(h)	304(h)	305(h)
Intel	00(h)	01(h)	01(h)	01(h)	02(h)	01(h)
Motorola	01(h)	00(h)	01(h)	01(h)	01(h)	02(h)

However these problems are reasonably easy to solve.

© James R. Drummond - September 1997

Floating Point Numbers

Floating point numbers are another story entirely. There are so many standards that interchange between computers is rarely simple, and often hair-raising! Most people (except IBM) store floating point numbers as something like $n * 2^m$ where n is between 0 and 2. However there is little agreement on the lengths and format of n and m. IBM stores numbers as $n * 16^m$.

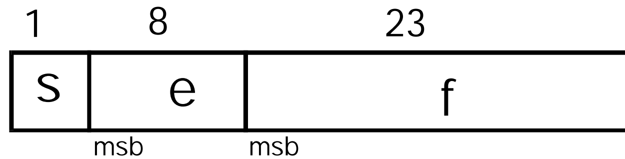
An evolving standard for floating point numbers is the "IEEE format" which defines a number of formats for floating point numbers based on the formula $s * n * 2^m$. Where s is the sign of the mantissa (+ 1 or -1), n is the value of the mantissa which is normalised to lie in the range $2.0 > n > 1.0$ and m is the exponent.

A typical single precision 32-bit floating point number would be stored as a sign bit (1 = -ve), a 24-bit mantissa and an 8-bit exponent. Those of you who are quick at arithmetic will realise that that doesn't add up - until you realise that the normalised form of the mantissa means that the most significant bit is always 1 and therefore that doesn't need to be stored. The exponent is stored in an offset form so that the number stored is always regarded as positive. A common bias is 128 which gives an exponent range of $2^{\pm 127} \approx 10^{\pm 38}$. Since the mantissa is 24 bits long, the resolution is $2^{-24} \approx 10^{-7}$.

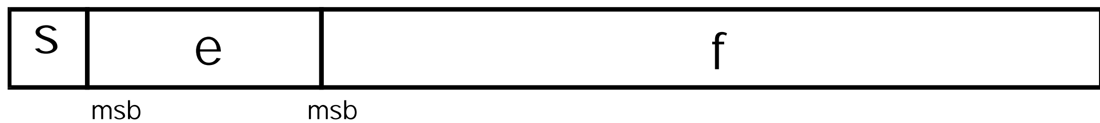
A typical double precision number extends the length of the mantissa and may (or may not) also extend the length of the exponent.

© James R. Drummond - September 1997

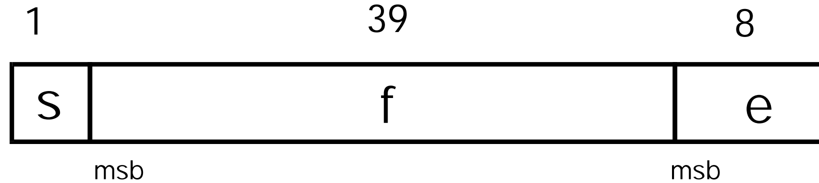
$$\text{Single (IEEE)} = (-1)^{s} \times 2^{(e-127)} \times (1.f)$$



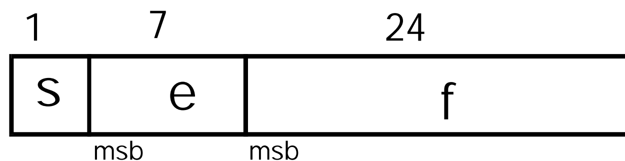
$$\text{Double (IEEE)} = (-1)^{s} \times 2^{(e-1023)} \times (1.f)$$



$$\text{Real (T. PASCAL)} = (-1)^{s} \times 2^{(e-129)} \times (1.f)$$



$$\text{Single (IBM)} = (-1)^{s} \times 16^{(e-64)} \times (0.f)$$



IEEE, PASCAL and IBM Floating Point Formats

Compression

"Any storage system once installed is full"[2]

The motivation for data compression comes from the above observation. Data compression consists of mapping the full representation of the data into a smaller (more compact) representation using a defined set of rules. There are two types of data compression:

Lossless - implies that the data can be recovered intact (bit-for-bit) by a reverse process

Lossy - implies that an approximation to the data can be recovered.

In most scientific applications only lossless compression is useful. Lossy compression is mainly used in image work where the human eye cannot resolve all the information presented anyway. We will concentrate on lossless compression.

There is a second use for "lossless" compression and that is in communication situations where bandwidth (rate at which you can send information) is limited. Obviously if you can reduce the number of bits to be sent - you save time and/or $$$.

Lossless data compression requires that there be some "pattern" to the data. It is impossible to compress random bit patterns. However most data is structured - a typical structure is ASCII text.

Huffman Coding

Huffman coding works on the principle that some codes occur more often than others. In English the most common code is that for the letter "e"[3]. In a static Huffman encoding the probability of occurrence of each symbol is assumed fixed and a "tree" is built on the following basis:

Find two of the lowest probability symbols. These are "leaves" and are combined into a

---

[2] Drummond's Law #?

[3] This observation is the basis for some old cypher-breaking techniques - see "the Adventure of the Dancing Men" in "The Return of Sherlock Holmes" by A. Conan Doyle

"node" which has the sum of the probabilities of the two leaves. Take the remaining symbols and this node and repeat until we have a single node with probability unity.

With both the transmitter and receiver in possession of this tree the compression proceeds as follows:

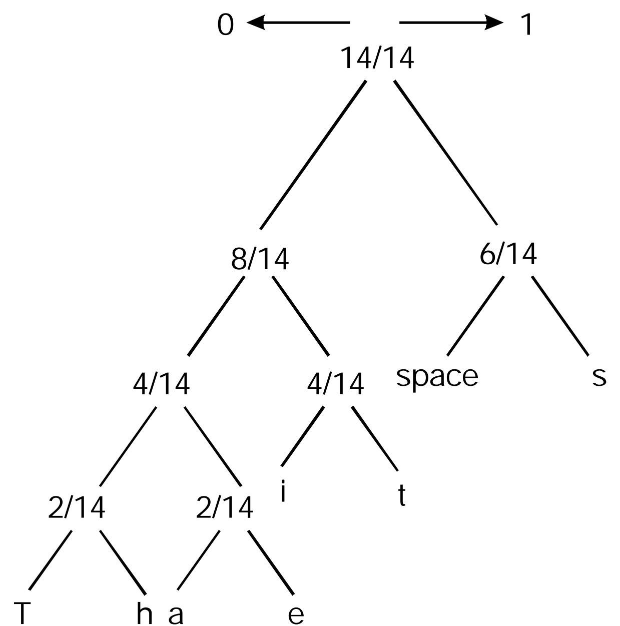
Take first character of input. Find location on tree. Start from top of tree. For each "left" branch on the way to the character output "0" and for each "right" branch output "1". Get next character and repeat.

The expansion algorithm can obviously proceed in the same manner.

Here is a simple example. The message to be compressed is: "This is a test". In 7-bit ASCII this requires 14*7= 98bits of storage.

First we will customise a tree:

Character	Probability
space	3/14
s	3/14
i	2/14
t	2/14
T	1/14
h	1/14
a	1/14
e	1/14



Huffman Tree - Fixed Encoding

Now we construct the output:

```
0000 0001 010 11 10 010 11 10 0010 10 011 0011 11 011
```

© *James R. Drummond - September 1997*

```
T   h   i   s    i   s    a      t   e    s  t
```

which requires only 40bits for the representation.

This is a pretty trivial example but you get the idea.

The trouble with static Huffman encoding is that you have to know the table and a table that compresses English text perfectly will not work well with French (or German, let alone Japanese!). We therefore need a dynamic method of constructing the tree.

The methodology is as follows:

A tree consists of nodes and leaves each of which has a weight which is equal to the number of times the symbol (leaf) or symbols below (node) have been used. There is one empty leaf (MT) with a weight of zero. The tree is initialised with one MT leaf.

Read in the next character. If it is in the tree, send out the tree code and increment the symbol weight by one. If it is not in the tree, then send the code of the MT leaf followed by the literal code of the new symbol. Replace the MT leaf by a node with the MT leaf on the "0" side and the new symbol on the "1" side with weight one. Now look at the tree and starting from the bottom ensure that the weights increase as you go towards the top. If not, starting with the lowest, swap the offending leaf or sub-tree with the one above with is nearest but below it in weight. In the event of two possible choices, take the one put in the tree first. Repeat the process until the weights progress properly. Now get the next character.

Both the compressor and the expander can construct the tree from this algorithm knowing only the bit length of the basic symbol (how many bits long is the literal code of a symbol).

The tree construction of the same message as above is given in the accompanying diagram. The output message will be:

```
"T" 0"h" 00" 000"s" 100space 11 101 1001 1000"a"
 T   h    i     s              i  s         a

01 10000"t" 10100"e" 00 110
      t        e      s  t
```

(I've abbreviated the symbol code for the ASCII set by using quotation marks)

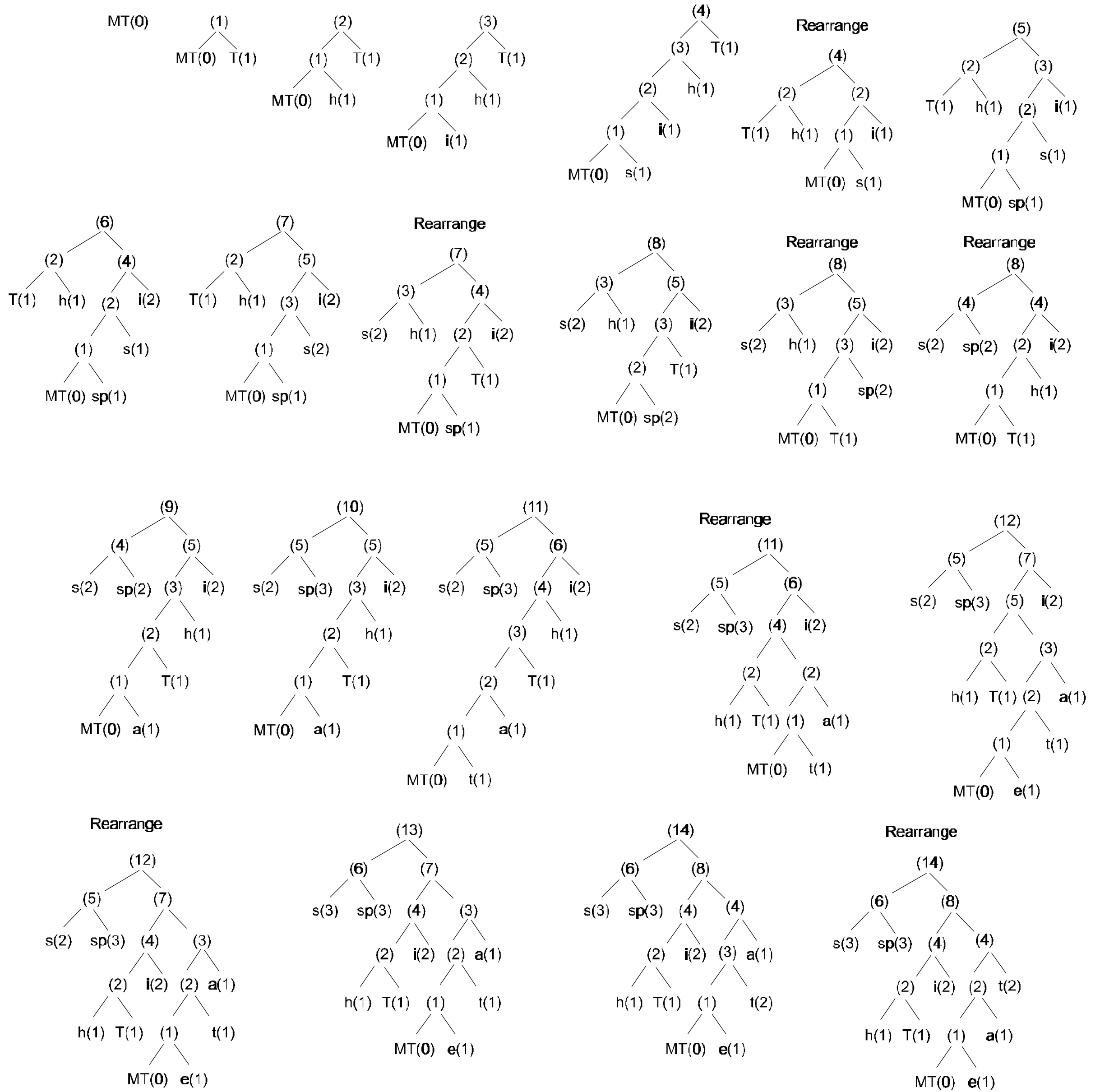
© *James R. Drummond - September 1997*

Notice two things about this message:

1) That a single ASCII character requires more than 8 bits to send because there is additional information to send. This is a general characteristic of compression algorithms. Because of this a compression algorithm doesn't necessarily produce an output that is shorter than the input - it may be longer.

2) That the code for a given character varies in this case. For example the representation of the "space" character is always different.

© *James R. Drummond - September 1997*



Huffman Tree - Dynamic Encoding

Lempel-Ziv-Welch (LZW) Compression

This compression methodology relies on the recognition of "strings" within the input and then effectively extends the "symbol set" to shorthand represent the strings. Thus with ASCII text the first 128 symbol codes can be allocated to the basic set and then codes greater than 256 are allocated to strings. (In practice 256 symbols are allocated to the basic set, but that's a detail) Note that we need to define how many bits are allocated to the total symbol set. A simple algorithm for this is to use 9-bit encoding until you have 512 symbols, then switch to 10-bit and so on. There are also optimisation procedures which I won't go into here.

The algorithm is:

Start with a null (empty) string and a table consisting of the basic character set.

Pick up the next character and append to string. If string is in the table, then get the next character. If it isn't in the table, then put out the code for the previous string (which was in the table) and generate a new entry for the current string. Throw away all the characters except the last character of the current string (which becomes the current string with a single character in it) and get the next character.

Here is the analysis of our current test string:

Input	Table	Output
T	-	-
h	128 ← "Th"	"T"
i	129 ← "hi"	"h"
s	130 ← "is"	"i"
space	131 ← "s"+ space	"s"
i	132 ← space+ "i"	space
s	-	-
space	133 ← 130+ space	130

© James R. Drummond - September 1997

Input	**Table**	**Output**
a	134 ← space+ "a"	space
space	135 ← "a"+ space	"a"
t	136 ← space+ "t"	space
e	137 ← "te"	"t"
s	138 ← "es"	"e"
t	139 ← "st"	"s"
end		"t"

Notice that in this case there really isn't much compression. However the efficiency rises with length of message.

Comparing Compressions

There really isn't much to say about how to compare compressions in the general case. There is always a particular case which shows some method to be better than others. I've also restricted the examples to English ASCII text but you can compress any sequence of uniform length symbols. The only real solution is to try several methods and see which is best, but that gets tedious.

Here's a semi-serious example. A very large set of integer numbers in the range 0-1000 has been written out in tabular form with 5 character spaces per number. Here is a part of the output:

```
  365  127  333  567  134  999    0   10  670 1000  825  666
```

How would this compress the ASCII text of this output with the various schemes?

Compression Scheme	Comments	Bits/ Number
ASCII	Each number takes 5 characters for 5*7= 35bits/number. (Actually it would almost certainly be 8bits/character but never mind.)	35
Huffman	There are only 11 symbols (0-9 + space) and so these will require about 4 bits to describe the tree ($2^4 > 11 > 2^3$). So each five character sequence contains 5*4= 20bits/number	20
LZW	There are only 1000 strings (one for each possible number) and so this can be represented in 11 bits (not 10 because we need the ASCII character set as well). So each five character sequence needs 11bits	11
Binary	Straight binary needs 10 bits/number. (Unless we had to use byte boundaries in which case it would be 16)	10

In this case (highly repetitive data) LZW compression wins handsomely. Except that the straight binary output is even better. This illustrates a significant issue: In large datasets it is often better to retain them in binary format and provide "viewer" programs to convert them to readable text, than to store them as text or even as compressed text.

Another example: A random collection of lower case letters.

In ASCII this is 7bits/character

In Huffman terms we would have a tree with 26 leaves which needs 4bits/character

LZW compression would have a terrible time - no sub-strings in random text. I wouldn't like to predict the result or even if it would compress at all.

In this case Huffman wins.

© *James R. Drummond - September 1997*