LECTURE 11

CODING THEORY - II

Two things should be clear from the previous lecture. First, that we want the average length L of the message sent to be as small as we can make it (to save the use of facilities). Second, it must be a statistical theory since we cannot know the messages that are to be sent, but we can know some of the statistics by using past messages plus the inference that the future will probably be like the past. For the simplest theory, which is all we can discuss here, we will need the probabilities of the individual symbols occurring in a message. How to get these is not part of the theory, but can be obtained by inspection of past experience, or imaginative guessing about the future use of the proposed system you are designing.

Thus we want an instantaneous uniquely decodable code for a given set of input symbols, s_i, along with their probabilities, p_i. What lengths l_i should we assign, (realizing that we must obey the Kraft inequality), to attain the minimum average code length? Huffman solved this code design problem.

Huffman first showed that the following running inequalities must be true for a minimum length code. If the p_i are in descending order then the l_i must be in ascending order

$$p_1 \geq p_2 \geq \cdots \geq p_q$$

$$l_1 \leq l_2 \leq \cdots \leq l_q$$

For suppose that the p_i are in this order but that at least one pair of the l_i are not. Consider the effect of interchanging the symbols attached to the two that are not in order. Before the interchange the two terms contributed to the average code length L an amount

$$\text{before} = p_j l_j + p_m l_m$$

and after the interchange the terms would contribute

$$\text{after} = p_j l_m + p_m l_j$$

All the other terms in the sum L will be the same. The difference can be written as

$$\text{Before - after} = (p_j - p_m)(l_j - l_m)$$

One of these two terms was assumed to be negative, hence upon interchanging the two symbols we would observe a decrease in the average code length L. Thus for a minimum length code we must have the two running inequalities.

Next Huffman observed that an instantaneous decodable code has a decision tree, and every decision node should have two exits, or else it is wasted effort, hence there are two longest symbols which have the same length.

To illustrate Huffman coding we use the classic example. Let $p(s_1) = 0.4$, $p(s_2) = 0.2$, $p(s_3) = 0.2$, $p(s_4) = 0.1$, and $p(s_5) = 0.1$. We have it displayed in the attached Figure 11-1. Huffman then argued on the basis of the above that he could combine (merge) the two least frequent symbols (which must have the same length) into one symbol having the combined probability with common bits up to the last bit which is dropped, thus having one fewer code symbols. Repeating this again and again he would come down to a system with only two symbols, for which he knew how to assign a code representation, namely one symbol 0 and one symbol 1.

Now in going backwards to undo the merging steps, we would need at each stage to split the symbol that arose from the combining of two symbols, keeping the same leading bits but adding to one symbol a 0, and to the other a 1. In this way he would arrive at a minimum L code, see again Figure 11.1. For if there were another code with smaller length L′ then doing the forward steps, which changes the average code length by a fixed amount he would arrive finally at two symbols with an average code length less than 1 - which is impossible. Hence the Huffman encoding gives a code with minimum length. See Figure 11.2 for the corresponding decoding tree.

The code is not unique. In the first place at each step of the backing up process the assigning of the 0 and the 1 is an arbitrary matter to which symbol each goes. Second, if at any stage there are two symbols of the same probability then it is indifferent which is put above the other, but this can result, sometimes, in very different appearing codes - but both codes will have the same average code length.

If we put the combined terms as high as possible we get Figure 11.3 with the corresponding decoding tree Figure 11.4. The average length of the two codes is the same, but the codes, and the decoding trees are different; the first is "long" and the second is "bushy", and the second will have less variability than the first one.

We now do a second example so that you will be sure how Huffman encoding works since it is natural to want to use the shortest average code length you can when designing an encoding system. For example you may have a lot of data to put into a backup store, and encoding it into the appropriate Huffman code has been known at times to save more than half the expected storage space! Let $p(s_1) = 1/3$, $p(s_2) = 1/5$, $p(s_3) = 1/6$, $p(s_4) = 1/10$, $p(s_5) = 1/12$, $p(s_6) = 1/20$, $p(s_7) = 1/30$ and $p_8 = 1/30$. First we check that the total probability is 1. The common denominator of the fractions is 60. Hence we have the total probability

$$(1/60)(20 + 12 + 10 + 6 + 5 + 3 + 2 + 2) = (1/60)(60) = 1$$

This second example is illustrated in Figure 11-5 where we have dropped the 60 in the denominators of the probabilities since only the relative sizes matter. What is the average code length per symbol? We compute

$$L = SUM[i=1,8;\ p_i l_i]$$

$$= (1/60)[20(2) + 12(2) + 10(3) + 6(3) + 5(3) + 4(4) + 3(4)]$$

$$= (1/60)[40 + 24 + 30 + 18 + 15 + 12 + 8 + 8] = 155/60$$

$$= 31/12 \sim 2.58...$$

For a block code of eight symbols each symbol would be of length 3 and the average would be 3, which is more than 2.58... .

Note how mechanical the process is for a machine to do. Each forward stage for a Huffman code is a repetition of the same process, combine the two lowest probabilities, place the new sum in its proper place in the array, and mark it. In the backward process, take the marked symbol and split it. These are simple programs to write for a computer hence a computer program can find the Huffman code once it is given the s_i and their probabilities p_i. Recall that in practice you want to assign an escape symbol of very small probability so you can get out of the decoding process at the end of the message. Indeed, you can write a program that will sample the data to be stored and find estimates of the probabilities (small errors make only small changes in L), find the Huffman code, do the encoding, and send first the decoding algorithm (tree) and then the encoded data, all without human interference or thought! At the decoding end you already have received the decoding tree. Thus once written as a library program, you can use it whenever you think it will be useful.

Huffman codes have even been used in some computers on the instruction part of instructions, since instructions have very different probabilities of being used. We need, therefore, to look at the gain in average code length L we can expect from Huffman encoding over simple block encoding which uses symbols all of the same length.

If all the probabilities are the same and there are exactly 2^k symbols, then an examination of the Huffman process will show that you will get a standard block code with each symbol of the same length. If you do not have exactly 2^k symbols then some symbols will be shortened, but it is difficult to say whether many will be shortened by one bit, or some may be shortened by 2 or more bits. In any case, the value of L will be the same, and not much less than that for the corresponding block code.

On the other hand, if each p_i is greater than (2/3)(sum of all the probabilities that follow except the last) then you will get a comma code, one that has one symbol of length 1 (0), one symbol of length 2, (10), etc., down to the last where at the end

you will have two symbols of the same length, (q - 1), (1111...10) and (1111...11). For this the value of L can be much less than the corresponding block code.

Rule: Huffman coding pays off when the probabilities of the symbols are very different, and does not pay off much when they are all rather equal.

When two equal probabilities arise in the Huffman process they can be put in any order, and hence the codes may be very different, though the average code length in both cases will be the same L. It is natural to ask which order you should choose when two probabilities are equal. A sensible criterion is to minimize the variance of the code so that messages of the same length in the original symbols will have pretty much the same lengths in the encoded message - you do not want a short original message to be encoded into a very long encoded message by chance. The simple rule is to put any new probability, when inserting it into the table as high as it can go. Indeed, if you put it above a symbol with a slightly higher probability you usually greatly reduce the variance and at the same time only slightly increase L; thus it is a good strategy to use.

Having done all we are going to do about source encoding (though we have by no means exhausted the topic) we turn to channel encoding where the noise is modeled. The channel, by supposition, has noise, meaning that some of the bits are changed in transmission (or storage). What can we do?

Error detection of a single error is easy. To a block of (n-1) bits we attach an n-th bit which is set so that the total n bits has an even number of 1's (an odd number if you prefer, but we will stick to an even number in the theory). It is called an <u>even (odd) parity check</u>, or more simply a <u>parity check</u>.

Thus if all the messages I send to you will have this property, then at the receiving end you can check to see if the condition is met. If the parity check is not met then you know that at least one error has happened, indeed you know that an odd number of errors has occurred. If the parity does check then either the message is correct, or else there are an even number of errors. Since it is prudent to use systems where the probability of an error in any position is low, then the probability of multiple errors must be much lower.

For mathematical tractability we make the assumption that the channel has <u>white noise</u>, meaning that: (1) each position in the block of n bits has the same probability of an error as any other position, and (2) that the errors in various positions are uncorrelated, meaning independent. Under these hypotheses the probabilities of errors are:

$$\begin{aligned}
\text{no error} &= (1-p)^n \\
\text{one error} &= C(n,1)p(1-p)^{n-1} = [n]p(1-p)^{n-1} \\
\text{two errors} &= C(n,2)p^2(1-p^{n-2} = [n(n-1)/2]p^2(1-p)^{n-2} \\
\text{three errors} &= C(n,3)p^3(1-p^{n-3} = [n(n-1(n-2)/6]p^3(1-p)^{n-3}
\end{aligned}$$

etc.

From this if, as is usually true, p is small with respect to the block length n, (meaning the product np is small), then multiple errors are much less likely to happen than single errors. It is an engineering judgment of how long to make n for a given probability of error p. If n is small then you have a higher redundancy (the ratio of the number of bits sent to the minimum number of bits possible, n/(n-1)) than with a larger n, but if np is large then you have a low redundancy but a higher probability of an undetected error. You must make an engineering judgement on how you are going to balance these two effects.

When you find a single error you can ask for a retransmission and expect to get it right the second time, and if not then on the third time, etc. However, if the message in storage is wrong, then you will call for retransmissions until another error occurs and you will probably have two errors which will pass undetected in this scheme of single error detection. Hence the use of repeated retransmission should depend on the expected nature of the error.

Such codes have been widely used, even in the relay days. The telephone company in its central offices, and in many of the early relay computers, used a 2-out-of-5 code, meaning two and only two out of the five relays were to be "up". This code was used to represent a decimal digit, since C(5,2) = 10. If not exactly 2 relays were up then it was an error, and a repeat was used. There was also a 3-out-of-7 code in use, obviously an odd parity check code.

I first met these 2-out-of-5 codes while using the Model 5 relay computer at Bell Tel Labs, and I was impressed that not only did they help to get the right answer, but more important, in my opinion, they enabled the maintenance people to maintain the machine. Any error was caught by the machine almost in the act of its being committed, and hence pointed the maintenance people correctly rather than having them fool around with this and that part, misadjusting the good parts in their effort to find the failing part.

Going out of time sequence, but still in idea sequence, I was once asked by AT&T how to code things when humans were using an alphabet of 26 letter, ten decimal digits, plus a "space". This is typical of inventory naming, parts naming, and many other naming of things, including the naming of buildings. I knew from telephone dialing error data, as well as long experience in hand computing, that humans have a strong tendency to interchange adjacent digits, a 67 is apt to become a 76, as well as change isolated ones, (usually doubling the wrong digit, for example a 556 is likely to emerge as 566). Thus single error detecting is not enough. I got two very bright people into a conference room with me, and posed the question. Suggestion after suggestion I rejected as not good enough until one of them, Ed Gilbert, suggested a weighted code. In particular he suggested assigning the

numbers (values) 0, 1, 2, ..., 36 to the symbols 0, 1, ... , 9, A, B, ... Z, space. Next he computed not the sum of the values but if the k-th symbol has the value (labeled for convenience) s_k then for a message of n symbols we compute

$$\text{SUM}[k=1,n;\ ks_k] \text{ modulo } 37$$

"modulo" meaning divide this weighted sum by 37 and take only the remainder. To encode a message of n symbols leave the first symbol, k = 1, blank and what ever the remainder is, which is less than 37, subtract it from 37 and use the corresponding symbol as a check symbol, which is to be put in the first position. Thus the total message, with the check symbol in the first position, will have a check sum of exactly 0. When you examine the interchange of any two different symbols, as well as the change of any single symbol, you see that it will destroy the weighted parity check, modulo 37 (provided the two interchanged symbols are not exactly 37 symbols apart!). Without going into the details, it is essential that the modulus be a prime number, which 37 is.

To get such a weighted sum of the symbols (actually their values) you can avoid multiplication and use only addition and subtraction if you wish. Put the numbers in order in a column, and compute the running sum then compute the running sum of the running sum modulo 37, and then complement this with respect to 37, and you have the check symbol. As an illustration using w, x, y, z.

symbol	sum	sum of sums
w	w	w
x	w + x	2w + x
y	w + x + y	3w + 2x + y
z	w + x + y + z	4w + 3x + 2y + z = weighted check sum

At the receiving end you subtract the modulus repeatedly until you get either a 0 (correct symbol) or a negative number (wrong symbol).

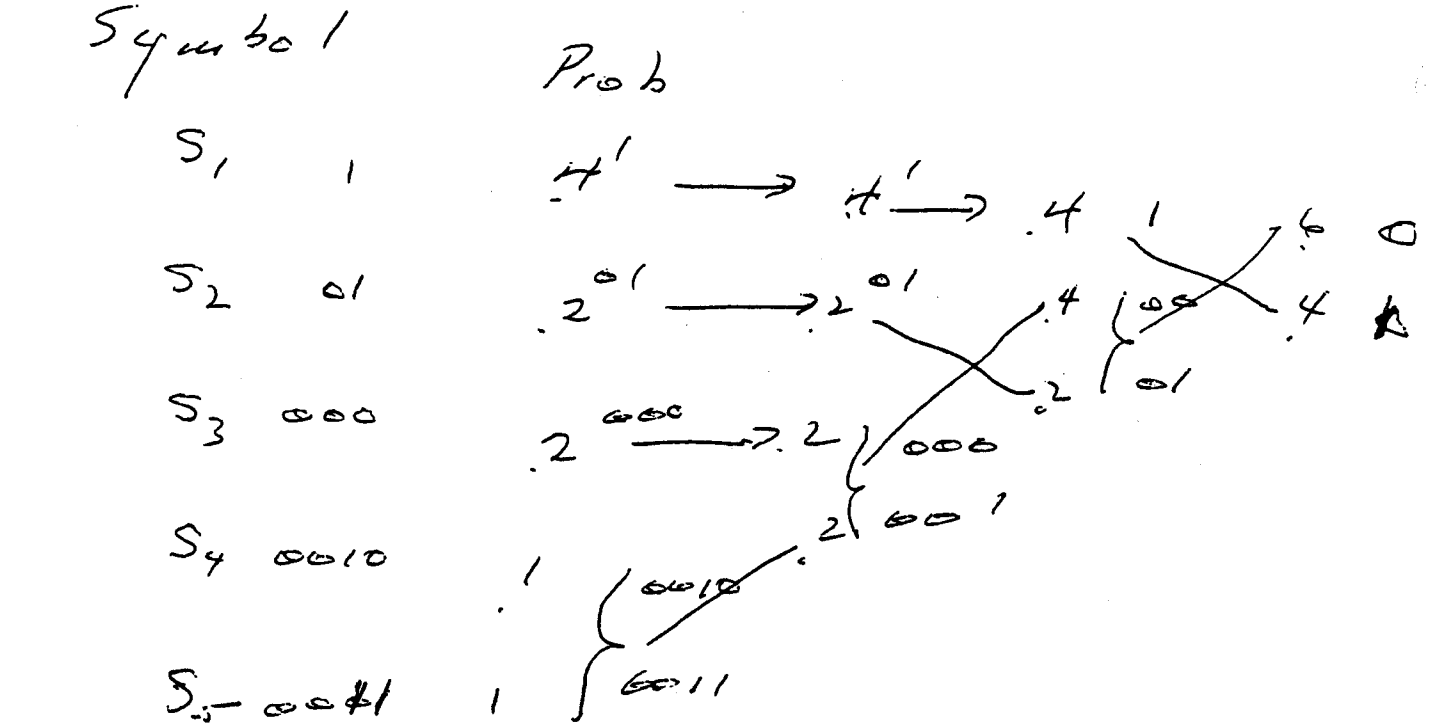
If you were to use this encoding, for example, for inventory parts names, then the first time a wrong part name came to a computer, say at transmission time, if not before (perhaps at order preparation time), the error will be caught; you will not have to wait until the order gets to supply headquarters to be later told that there is no such part or else that they have sent the wrong part! Before it leaves your location it will be caught and hence is quite easily corrected at that time. Trivial? Yes! Effective against human errors (as contrasted with the earlier white noise), yes!

Indeed, you see such a code on your books these days with their ISBN numbers. It is the same code except that they use only 10 decimal digits, and 10, not being a prime number, they had to introduce an 11-th symbol, labeled X, which might at times arise in the parity check - indeed, about every 11-th book you have will have an X for the parity check number as the final sym-

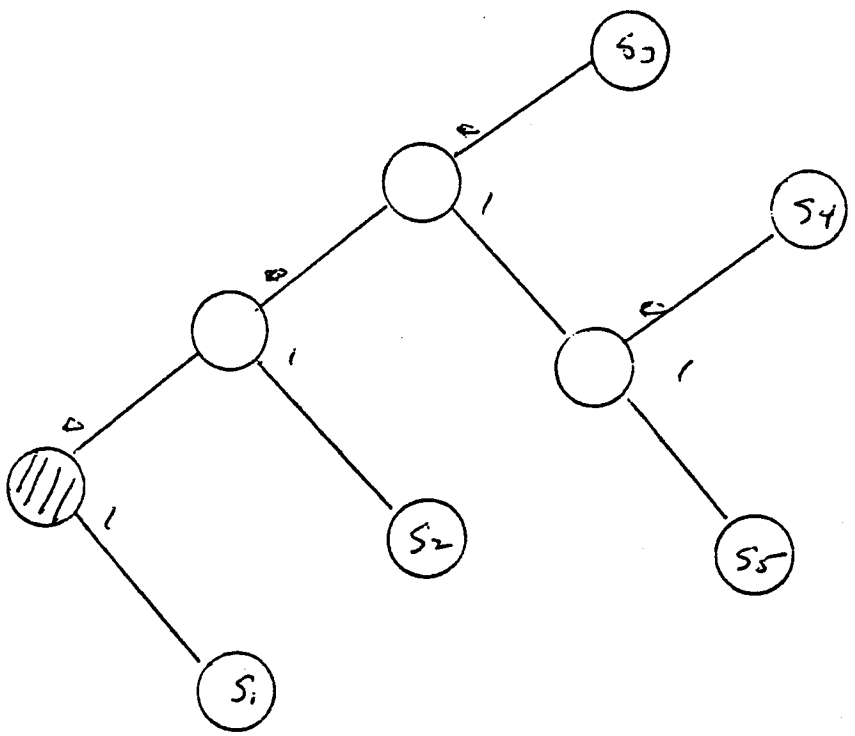
bol of its ISBN number. The dashes are merely for decorative effect and are not used in the code at all. Check it for yourself on your text books. Many other large organizations could use such codes to good effect, if they wanted to make the effort.

I have repeatedly indicated that I believe the future will be increasingly concerned with information in the form of symbols, and less concerned with material things, hence the theory of encoding (representing) information in convenient codes is a non-trivial topic. The above material gave a simple error detecting code for machine-like situations, as well as a weighted code for human use. They are but two examples of what coding theory can contribute to an organization in places where machine and human errors can occur.

When you think about the man-machine interface one of the things you would like is to have the human make comparatively few key strokes - Huffman encoding in a disguise! Evidently, given the probabilites of you making the various branches in the program menus, you can design a way of minimizing your total key strokes if you wish. Thus the same set of menus can be adjusted to the work habits of different people rather than presenting the same face to all. In a broader sense than this, "automatic programming" in the higher level languages is an attempt to achieve something like Huffman encoding so that for the problems you want to solve require comparatively few key strokes are needed, and the ones you do not want are the others.

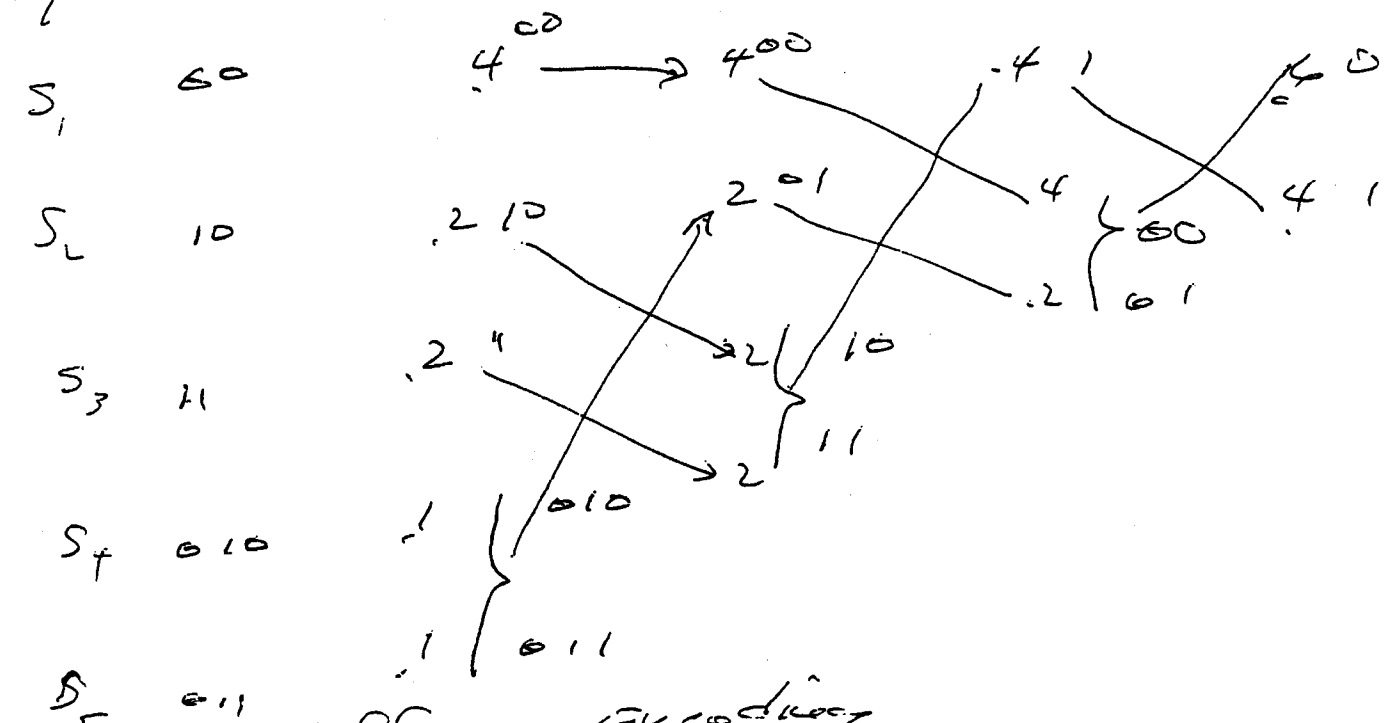


Huffman Encoding
Fig 11.1

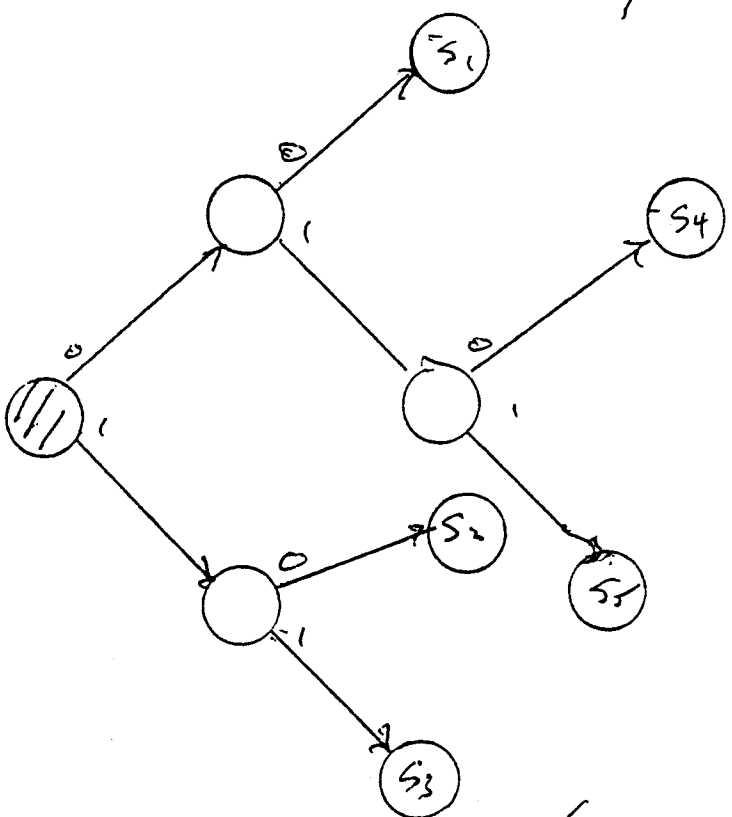


Decoding Tree

Fig 11.2



Huffman Encoding
Fig. 11.3



Decoding tree
Fig. 11.4

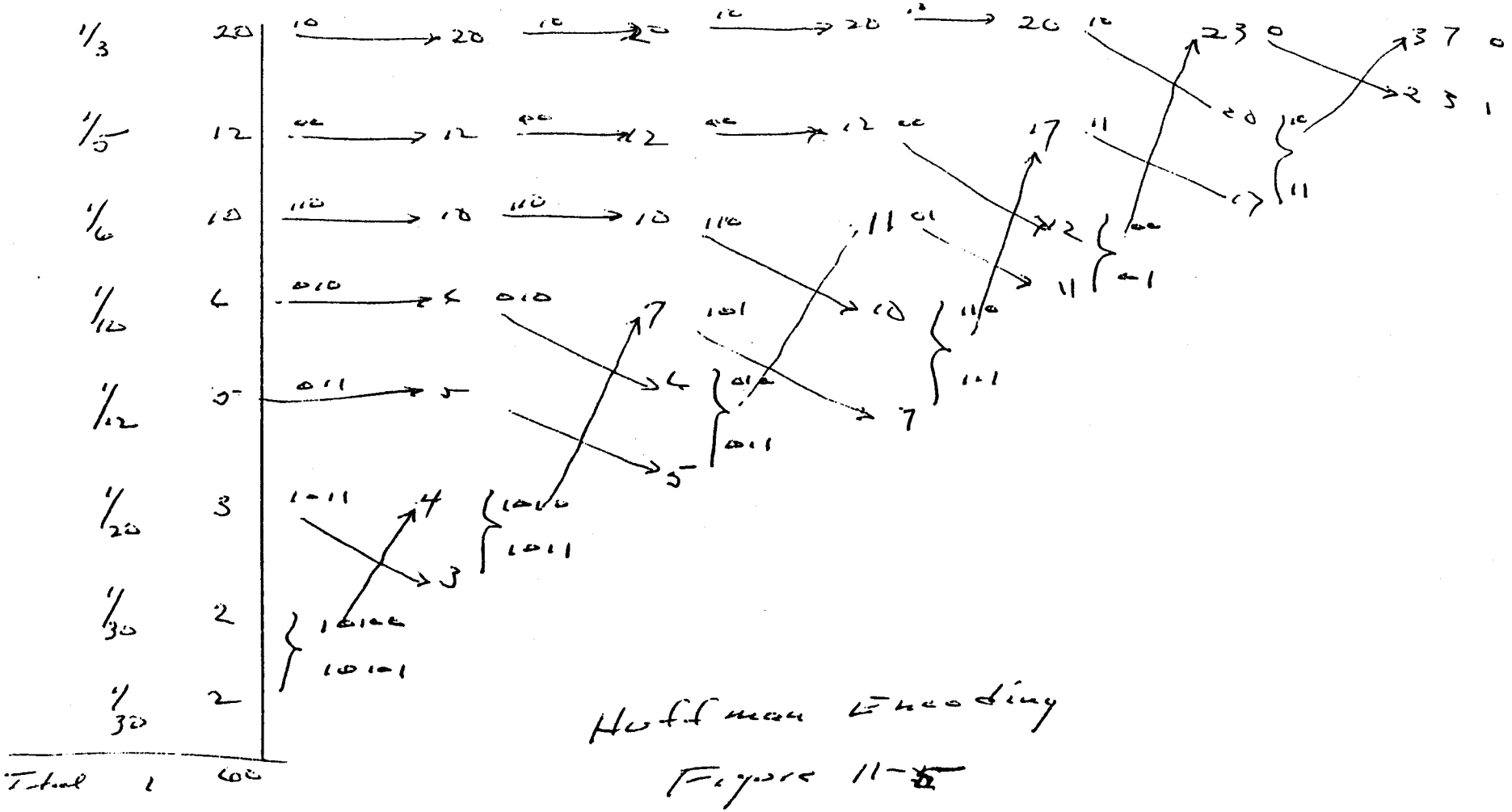

1/3
1/5
1/6
1/10
1/12
1/20
1/30
1/30
Total 1
20
12
10
4
3
2
2
60
Huffman Encoding

Figure 11-