To distinguish himself from the poor be-nighed man-in-the-street, the computer sophisti-cate is apt to refer to the beasts as "so-called giant brains" or as "lightning-fast idiots." He knows, as do we, the great gulf which separates the human brain from the general-purpose digital computer. Still, the exact dimensions of that gulf are quite unknown, and the desire to show that the hiatus between man and machine is smaller than many suspect impels both the ad-venturesome and the iconoclastic. The attempt, the successful attempt, to automate one area hitherto con-sidered an exclusively human domain constitutes my topic today.

We are all familiar, if only by hearsay, with the troubles that can beset the best of computer programs if the input to the program is not thoroughly debugged. For a program designed to test the putative behavior of, say, a proposed steam turbine, where the input consists of a scant dozen or so parameters, input debugging is hardly a problem. The situation is quite different for a data-processing operation, particularly when the input is massive, as it usually is. Three alternatives, all un-pleasant, present themselves to the supervisor of such a large-scale data-processing operation. He can build a wide variety of error-detecting features into his program, flagging all input errors for subsequent human correction, he can employ a host of human pre-editors to clean up the input, or he can hope that input errors are rare, and let it go at that.

Unhappily, there are many applications where errors are not rare, where the do-nothing solution is obviously frivolous and where, consequently, a sizeable group of humans is necessary, either as pre-editors or as on-line trouble-shooters. Nor is it always the case that the necessary human beings can be clerical types. Certain input debugging calls for sophisticated and knowledge-able practitioners. We are all hopeful-almost all, any-way—that keypunch machines and operators will sooner or later be superseded by character-reading devices and the like. There is no philosophical difficulty in conceiving of typed, printed, or handwritten characters being translated directly into computer language without any human intervention, provided, of course, that those characters were correctly typed, printed, or written to begin with. Suppose, however, that the source char-acters are incorrect. Consider the ingenuity expended in the Post Office just in recognizing all the variations of "Albuquerque." Our Russian colleagues are supposed to be far advanced in the domains of automatic translation and character-reading, but present their machines with a first edition of "Cybernetics," with all its typograph-i-cal errors, and horrible difficulties would ensue. Our choice, then, is clear. Either we admit that many im-portant data-processing applications are impossible to automate completely, or we find a way to mechanize the human capacity for making educated guesses. We be-lieve that, for some applications at least, we have found a way.

While the techniques we have developed were con-ceived with one particular application in mind, I shall describe them without reference to that application, successful as it was. The principal reason for taking this tack is to be able to present the basic, quite general, features of our method without being tripped up by the special form-fitting required by the actual problem. So, let us be general, and consider any language with which humans attempt to communicate with one another. These may be natural languages, like English or Ger-man, or artificial languages like Esperanto or certain telegraphic codes. There are all sorts of personal rea-sons for communication being difficult—ignorance, dogmatism, poor sentence structure, etc.; however, even if these factors did not exist, all sorts of nonhu-man noise would beset would-be communicators. In-formation theory makes much of "redundancy" as an aid in error-detecting and error-correcting when a noisy channel is being used. Indeed, even humans who have never heard of information theory make continual, and skillful, use of redundancy in unscrambling all sorts of garbled communications, whether the trouble be cross-talk in a telephone conversation or missing letters in a crossword puzzle. Without attempting to build a model of the brain, replete with neural nets and such, let us see if we can single out the functions performed by human redundancy-exploiters. If these functions turn out to be performable without recourse to extrasensory percep-tion or to the psychokinetic effect, our automation problem is essentially solved. There remain only the minor problems of collecting all the necessary data, carrying out a rather gruesome programming task and finding a computer fast enough and capacious enough to make our solution practicable. I shall return later to this question of practicability. At the moment, allow me to sketch the functions which, when suitably pro-grammed, allow a general-purpose computer to simulate a redundancy-exploiting, error-detecting, and error-correcting human being.

Rather than jump into a completely general and ab-stract formulation, let me use a concrete illustration. Fig. 1 shows two familiar sights, a correctly prepared mailing envelope and, below it, a somewhat sloppier ver-sion of the same thing. We shall assume at first that a

† General Electric Co., Bethesda, Md.
start off with the knowledge of the line it is on, its position on the line (left end, right end, interior) and the words which flank it on either side. With a little extra programming effort we can determine the length, i.e., the number of characters of each word, its character pattern (is it all alphabetic, all numeric, some sort of hybrid?) and, perhaps, the presence in the word of some salient feature, e.g., the colon following “ATT.:”. Indeed, we can usually determine quite easily much more information than we need for the identification of our words. Much more, that is to say, when we are dealing with a noiseless channel, and/or a communication format as simple and relatively invariable as the front of an envelope.

Of course, whether this information is adequate, overly complete, or inadequate depends on how we use it. At this point in the identification process, the machine must turn to its accumulated store of factual knowledge, a store which is compiled by a subsidiary program in advance of production running. This store consists of lists and tables of probabilities, and provides the data which, in conjunction with the specific information for each envelope, allow each word to be identified with a high probability of correctness. Our basic technique here is the use of Bayes Factors as instruments for weighing evidence. Fig. 2 gives the essentials of this technique.

For each class of words that can occur in the specific type of communication in question—mail envelopes, in our example—an a priori probability is given for the occurrence of a representative (or two, or n) of that class. This probability, like all the others we use in this process, is derived from frequency counts on sufficiently large samples of the data to be processed. Also for each class, we provide the probabilities that, for example, a specific representative of that class will have length 3, or 4, or 5, or that the class representative will be found at the beginning, or the end, of a line. In brief, for every piece of information we scan each envelope for, we have a corresponding set of probability distributions, one set for each class of expected words.

In the identification phase of our program, we consider one actual word at a time, testing that word against the hypotheses that it is a representative of expected class A, B, etc. Eq. (1) in Fig. 2 gives the skeleton of such a test. Here we are testing the hypothesis that the word “Smith” is a representative of the “zone-number” class. Our frequency counting is supposed to have informed us that the a priori probability that any word on our envelope is in the zone-number class is 0.017. We first test our hypothesis by using the empirically-determined fact that “Smith” has length 5. This gives us our second term on the right side of (1), i.e., the Bayes Factor for the “length event.” The product of the Bayes Factor and the a priori probability is the a posteriori probability that “Smith” is a zone-number. Not very surprisingly, this is a small number. We now compare this number with two thresholds. If the a
posteriori probability exceeds the acceptance threshold, we accept the hypothesized identification and turn our attention to the next actual word; if the probability falls below the rejection threshold, we reject the hypothesis, and test the actual word against the next expected word class. Finally, if our probability falls between the two thresholds, we test the same hypothesis against the next event, using our a posteriori probability as the new a priori probability. In our example in Fig. 2 we have been generously low with our rejection threshold, so that it is necessary to go to (2) where we test the hypothesis, “Smith = zone-number,” against the character pattern, and allow the low probability of a zone-number consisting exclusively of letters, to push our hypothesis into limbo. If we were scanning French addresses, with zone-numbers given in Roman numerals, the Bayes Factor in (2) would be very different.

After scrutinizing all the actual words on the envelope in this manner, we may find that certain words are still unidentified. In this case, we iterate through our process once again. However, certain features of the process will have changed. Suppose that we have identified two different zone-numbers in the first pass. Since we expect to find no further zone-numbers, we no longer test any of our undecided actual words against the hypothesis that they are zone-numbers. This not only reduces our processing time—it also changes the a priori probabilities of the remaining word classes, and affects the numbers entering into all the Bayes Factors. Another change in the second pass is that new evidence can be used to give rise to Bayes Factors. A word identified as a zone-number in the first pass provides strong evidence that the word to its left is a city name. Clearly, the topological relationships subsisting between words cannot be utilized until some words have been identified.

If successive identification passes still leave a residuum of unidentified actual words, as might happen if, for example, two or more words were run together, thus appearing to the machine as one word, there are subsidiary tricks that can be played. Due to time limitations, I shall have to leave these tricks to your imagination, and move on to the recognition phase.

In the simplest case, all actual words will have been correctly identified and, if the words are all correctly spelled and correctly ingested by our character-reader, recognition will consist of little more than finding the exact match in the proper list, a list determined by the identification of the word. It is possible to make even this simple process simpler or, at least, faster. To search a list of all the cities in the United States can be time-consuming, particularly if the list must be transferred from tape to core memory. However, if the corresponding state has previously been recognized, then a much reduced list of cities can be inputted and searched. Suppose further that the corresponding zone-number has been recognized as “25.” Then we need consider only those cities in the given state which have at least 25 zones.

If we are bound to get a direct match whether we scan a big list or a little list, this process of list reduction is of secondary value only. It is when a direct match is not forthcoming that this technique assumes greater importance. In the absence of a direct match, we are constrained to use brute force techniques of a more or less sophisticated nature. If we are fortunate enough to reduce a list down to one entry, then we can avoid brute force completely. Failing this, we can expect two advantages to accrue to the use of a reduced list, in general. First, for the same elapsed time, we can employ more brute force techniques per list entry; second, we can at least hope that, by reducing our initial list, we will ex­punge spurious candidates to which our brute force techniques might give scores equal to, or even greater than, the score of the correct candidate. For example, Fig. 3 gives one horrible example, often quoted in this connection. Recognizing (a) as being either “New York” or “Newark” is an awful job. A non-brute-force technique, such as list reduction, which removes the false entry from consideration is a welcome way of cutting this Gordian knot.

Again for lack of time, I must give the actual brute-force techniques a very hasty treatment. Let me mention just two techniques of the many available. To match a word which has had two letters transposed [as

\[
P(H|E_0) = \frac{P(E_0|H)P(H)}{P(E_0)} = \frac{P(E_0|H)}{P(E_0)} \frac{P(H)}{P(E_0)}
\]

where

\[ H = \text{hypothesis that "Smith" is a "zone-number"} \]
\[ E_0 = \text{the event that the length of "Smith" is "5"} \]
\[ E_{\text{a}} = \text{the event that the pattern of "Smith" is "all alphabetic"} \]

Fig. 2—Hypothesis testing.
in (b) in Fig. 3], against the original word, we look for list entries with the same letter composition as our actual word, i.e., entries with the same number of A's, B's, C's, etc. Scanning these for a single transposition is relatively easy.

A second technique is useful when a letter or two (or more) has been erroneously dropped from, or added to, a word. (c) in Fig. 3 is due to a stuttering typist who repeated the first letter of the word. Two words run together provide further examples of this kind of noise. What we try here is a direct match of our actual word with a proper subset of our list entries, and vice versa.

![Fig. 3—Typical typographical errors.](image)

If no amount of brute force seems to work, and certain words just cannot be recognized, we can either give up gracefully at this juncture or we can admit, even more gracefully, that one of our educated guesses might have been wrong. If we choose the latter alternative, we have the messy job of deciding whether we went haywire in the recognition phase, or all the way back in the identification phase. In either case, it is still necessary to find a likely spot for picking up the dropped stitch without causing the entire garment to unravel. Sometimes, indeed, we are left with the original ball of wool. These, however, are almost always the cases which stump human editors.

This ability to iterate back, and back, and back, can of course lead to excessive use of computer time. It does have its advantages though. It means that a bad guess is not an irrevocable misstep. It also means that various parameters, the identification acceptance and rejection thresholds, for example, are not nearly as critical as they would be in a one-through process. Since these are among the hardest parameters to estimate accurately, any diminution of their sensitivity is a positive gain.

At this point, I should like to restate our major techniques in somewhat folksier terms than "Bayes Factors" and "list reduction." In our identification phase, we attempt to use the constraints imposed by the format, mailing envelopes in our example, plus the constraints of the language itself, the length and character patterns of the expected word classes, to provide a rudimentary form of pattern recognition. We then use our *a priori* knowledge of word statistics and interword relationships to find the most probable matching of the actual words to the expected word classes. Human beings presumably use rank orderings of hypotheses, modified by intuition, to perform such matchings. Since machines lack intuition and since we have not yet developed a calculus of rank orderings, we use the paraphernalia of Bayes Factors to accomplish the same task.

Without undue stretching of the terms, we might say that, in the identification phase, we exploit the syntactic constraints on the language we are processing, whereas in the recognition phase, by our use of the list reduction technique, we exploit the semantic constraints. We might subsume both types of constraint under that much-abused word, redundancy. As for a folksy term for our brute-force techniques, the most accurate that occurs to me is "knowledgeable cynicism." We expect errors to be made and, usually, we have some information as to the kinds and sources of error, as well as their frequencies of occurrence. If we know that typists frequently hit a key next to the one they should hit, we store the keyboard pattern in our program; if we know that our character-reader frequently confuses "o" with "c," that, too, goes into our dossier.

A final word now as to the applicability and practicability of our techniques. What with tape searches and Bayes Factor computations, processing time may, but need not always, be excessive. The preparation of all the lists required in the recognition phase is a painful task. With a relatively stagnant language, this list-making can be a one-shot ordeal; with a volatile language requiring frequent updating of the lists, the pain might be unbearable. What has been said about lists also applies to the preparation of the probability tables for the identification phase. A final pause-giving consideration is the amount of redundancy in the language to be processed, particularly when the processor cannot establish the language, which he sometimes can do. Our private feelings are that a language sufficiently low in redundancy to be unintelligible to a machine will also be unintelligible to a man. I won't press a point which trods so heavily on anthropocentric toes.

In summary, then, we feel our techniques can be useful in some massive data-processing applications, in automating post offices, in translating natural languages, where every second word in the source language has several correlates in the target language, and we know our techniques have worked at least once. I won't ask that you take this on faith, though I'd appreciate it if you would.
A Memory of 314 Million Bits Capacity with Fast and Direct Access—Its Systems and Economic Considerations

N. BISHOP† AND A. I. DUMEY‡

The graphic arts laboratory of Time Inc. in Springdale, Conn., is a laboratory largely devoted to improving the arts of printing and papermaking. Of equal importance to a publisher of weekly news magazines is a large and well-serviced list of subscribers. This may serve to explain why there exists today in Springdale a completed and working engineering prototype of a large direct-access memory. This equipment was developed by the laboratories to prove the technical and economic feasibility of storing and processing large basic record files in memories providing direct access facilities to individual records. Since our program of systems study, equipment development, and economics study has indicated such feasibility, a review of our findings may well open new avenues of approach to the solution of present and expected future problems in the expanding field of automatic data processing.

The data-processing systems designer is constantly striving to strike an optimum balance between the costs of data storage, computer time, job requirements of input and output, and manpower. So fast is the art progressing that today's best balance for customer A may be quite different from that for customer B a year later, even though A and B present identical job requirements.

The concept of large data-storage capacity coupled with direct access to individual records has long intrigued the systems man. His interest is usually diminished when he starts to consider the cost of storing large basic files in such a manner with the presently available memories providing such direct-access facilities. Either the storage cost per bit is too high or the access time is too high to keep up with the daily work load. Serial storage provides high capacity and low cost per bit, but it does not provide for the often essential requirement of direct access to individual records. There appears to be a definite gap in the systems man's bag of tricks and the need to cover high activity against the file, as in the case of short-term airline reservations.

The engineering prototype of our memory has a total information bit capacity of 314,500,000. This capacity is distributed in 262,144, \(2^{18}\), separate storage locations within the memory, each location providing a message capacity of 1200 bits. The operation of the memory is under the control of three basic commands derived from peripheral equipment. The first command consists of an 18-bit parallel-fed address, followed by a seek record signal. The second is a process command calling for a read, write, or erase cycle. The third is a release-record command, which restores the memory to readiness for operation on the next series of commands.

The maximum time required to execute a seek record command is 3.87 seconds and the minimum is 0.61 second. Each process cycle, read, write, or erase, takes 0.45 second. On completion of the last process cycle, 0.6 second is required to restore the memory to a condition of readiness for the next series of commands.

These are the basic figures. Here are some examples for manufacture and maintenance. Excessive travel distances from record to processing station result either in immoderate acceleration and torque requirements or excessive access times, if acceleration and torque requirements are kept within bounds. Too many records per access mechanism, with consequent increase in traffic per unit, can overburden the designer with access-speed requirements. The key to an approach which will meet systems and economic requirements is expressed in one word— moderation. A combination of moderate packing factors, moderate travel distances, moderate accelerations, and moderate average access times can result, and has resulted, in a fast and direct access memory with a record capacity sufficient for economical large basic file application. Let us now discuss its operational characteristics.

The significant characteristics of a direct access memory to a systems man are those which answer the questions: "How much data can I store?" "How many entries can I process per working day?" "What is the best available estimate of the cost per bit of storage plus provision for direct access?"

As far as the bit capacity of presently available direct-access equipment is concerned, the upper bound seems to be a few tens of millions; and, on the cost side, the lower bound seems to be some ten bits for the penny. Access times in such memories vary from a second down to a few milliseconds, the latter figure reflecting the need to cover high activity against the file, as in the case of short-term airline reservations.

The engineering prototype of our memory has a total information bit capacity of 314,500,000. This capacity is distributed in 262,144, \(2^{18}\), separate storage locations within the memory, each location providing a message capacity of 1200 bits. The operation of the memory is under the control of three basic commands derived from peripheral equipment. The first command consists of an 18-bit parallel-fed address, followed by a seek record signal. The second is a process command calling for a read, write, or erase cycle. The third is a release-record command, which restores the memory to readiness for operation on the next series of commands.

The maximum time required to execute a seek record command is 3.87 seconds and the minimum is 0.61 second. Each process cycle, read, write, or erase, takes 0.45 second. On completion of the last process cycle, 0.6 second is required to restore the memory to a condition of readiness for the next series of commands.

These are the basic figures. Here are some examples

† Time Inc., Springdale Labs., Div., Springdale, Conn.
‡ Systems Consultant, Roslyn Heights, N. Y.