Towards correct programs.

The purpose of this paper is to stress what mental aids we have at our disposal in designing and understanding algorithms, to show some patterns of programming that we may hope to apply without losing our intellectual grasp on what we are doing and to stress the need that our programs (i.e. the final program and the intermediate programs leading to it) mirror as closely as possible our understanding of the problem and the algorithm solving it.

Among the mental aids available I should like to mention three explicitly:
1) Enumeration
2) Mathematical Induction
3) Abstraction.

I regard as an appeal to Enumeration the mental effort required to understand either a sequential program generating a fixed time sequence of actions or a conditional or alternative clause (the so-called "case construction" included). As one of the principle properties of the human mind I take that the appeal to enumeration should be small. In particular this means
1a) that the text describing a sequential program fragment should generate a small number of actions (i.e. that the corresponding computation can be grouped, understood as the time succession of a small number of actions)
1b) that the number of cases to be distinguished in an alternative clause should be small.

Mathematical induction is mentioned explicitly because it is the standard pattern of reasoning to understand recursive procedures and the (much more common) loops; I shall restrict myself to loops generated by some form of repetition clause.

Abstraction is regarded as the main mental tool needed for the application of mathematical induction -i.e. to form the concepts in terms of which the nett effect of the induction step can be described- and valuable in the effort to reduce the appeal to enumeration: in the case of an alternative clause it should provide the concepts in terms of which the nett effect can be described regardless the path taken.

With the above in mind I shall tackle the following problem. Given 32 cyclically arranged positions, make a program generating all ways (if any) in which these positions can be filled with zeros and ones (one digit per position) such that the 32 quintuples of five adjoining positions present the 32 different patterns of five binary digits. Fillings that only differ from each other by rotation are regarded as equivalent, all solutions have to be generated with the five zeros leading. (The last sentence reduces the number of "independent solutions" by a factor 32.)

C.Ligtmans has shown that the above cyclic problem is equivalent to the following linear problem. Given a linear array of 36 positions, make a program generating all ways (if any) in which these positions can be filled with zeros and ones (one digit per position) such that the 32 quintuples of five adjoining positions present the 32 different patterns of five binary digits. We may restrict ourselves to sequences starting with five zeros, the 32 leading digits of a solution of the linear problem present a solution of the cyclic one and vice versa. (The proof by C.Ligtmans runs as follows. Each solution of the linear problem starts with 000001... -because the pattern 00000 may occur only once-; furthermore the pattern 10000 must occur once; as the latter patterns can only be followed by 00000 or 00001 (presented already by the first two quintuples), the pattern 10000 cannot occur in the interior of the linear sequence and therefore

it must occur at the end. As a result each solution of the linear problem ends with four zeros and therefore the ring closes!) We shall solve the linear problem, imposing the further condition that the solutions (if more than one) have to be generated in alphabetical order.

The coarsest description of the program is as a single instruction
Version 0:
"do all work"

This description is completely general, but also completely useless, for it reflects none of our understanding of the problem nor anything about the structure of the algorithm. In order to be able to proceed we have to analyse the problem further.

I take for granted that, given a sequence of 36 binary digits the boolean function stating whether this sequence represents a solution is computable and that we could write an algorithm computing it. In principle we could write a program generating all 36-digit sequences with five leading zeros in alphabetical order and subjecting all these sequences to the test just mentioned, thereby selecting those satisfying the test. This gives a very unrealistic program and we shall not pursue it; we only remark that generating the trial sequences in alphabetical order will ensure that the solutions, when found, will be found in alphabetical order as well.

Remark. Our final program could be regarded as a derivation from the one sketched above, viz. by the introduction of some significant short-cuts. At present I do not feel inclined to stress this relation any further for it seems too tightly connected with this specific problem.

In our next approach we shall again generate our solutions by (generating and) scanning a larger set of sequences from which by a suitable criterion all solutions will be selected.

Let us define as "length of a sequence" the number of quintuples it contains (i.e. length = number of digits - 4). Let us call a sequence "acceptable" if no two different quintuples in it present the same digit pattern. With these definitions the solutions are a subset of the set of acceptable sequences, viz. those with length = 32.

We do not know whether there are any solutions at all but we do know that the set of acceptable sequences is non-empty (e.g. "00000"); we do not have a ready made criterion to recognize "the last solution" when we encounter it, in our set of acceptable sequences, however, we can designate a "virtual last one" (viz."00001"): when that one is encountered we know that all acceptable sequences with five zeros leading have been scanned and that no further solutions will be found.

Summarizing, we know of the set of acceptable sequences
1) it is non-empty and finite
2) we know a first member ("00000")
3) we know a virtual last member ("00001")
4) we can transfor an acceptable sequence into the next acceptable sequence
5) solutions are all acceptable sequences (excluding the virtual one) satisfying the further condition "length of sequence equals 32".

The transition from considering only the set of solutions to considering the set of acceptable sequences seems to mark a step in our analysis sufficiently relevant to justify that it be expressed in our first refinement, a program in

which all instructions operate upon (a still rather abstract) object, called "sequence".

Version 1:

```
"set sequence to first acceptable one;
 repeat  if length of sequence equals 32 do
         begin accept sequence as solution;
               print solution
         end;
         transfrom sequence into next acceptable one
 until sequence is (first or) virtual last member"
```

In explaining this program, two remarks seem appropriate.
Remark 1. The final test "sequence is (first or) virtual last member" has to distinguish the virtual last member from all members excluding the first one, as the first one will not be subjected to the test. Basically there would be no objection to the first member being equal to the virtual last one (say: the empty sequence). The above freedom is expresses by putting "first or" within parentheses.
Remark 2. The statement "accept sequence as solution" may well puzzle the reader. It may turn out to be an empty statement. It is inserted to express that "at the following semicolon" the sequence is regarded as representing a solution. (One may think of it as extracting the 32 leading digits.)

The criterion "acceptable" has a further important property:
6) no extension of a sequence that is not acceptable will be acceptable.

and it is this important property that will be exploited in the refinement of "transform sequence into next acceptable one". As a direct consequence of property 6, the acceptability test need only be applied to so-called "promising sequences", where a promising sequence is defined as a one-digit extension of an acceptable sequence. This leads to our next refinement

transform sequence into next acceptable one:

```
"transform acceptable sequence into next promising one;
 while promising sequence not acceptable do
       transform promising sequence into next promising one;
 accept sequence as acceptable"
```

When we regard "transform sequence into next acceptable one" as an available primitive, the value of "sequence" as always acceptable; it is only in the interior of its dissection -viz. at the semicolons of the above refinement- that the current value of "sequence" is a promising sequence.

In view of the required alphabetical ordering "transform acceptable sequence into next promising one" forms a new sequence equal to the old one extended with a zero, while "transform promising sequence into next promising one" forms a new sequence equal to the old one up to and excluding the last zero, followed by a one, or more explicitly as in the next refinement

transform promising sequence into next promising one:

```
"while sequence ends with a one do remove final digit from sequence;
 replace final zero by a one" .
```

In the above step-wise refinements we have focussed our attention upon the sequencing of the program. Now the time has come to introduce more explicit the decisions as how the still rather abstract object called "sequence" has to be represented.

We introduce an integer k and decide that k = length of the sequence. Furthermore we introduce an integer array d[-3:33] to represent the digits with

d[-3] d[-2].....d[k] representing the sequence. (1)

Remark 1: d[-3] through d[0] will be equal to zero.
Remark 2: The maximum length of an acceptable sequence = 32; as the algorithm handles promising sequences and a promising sequence is a one-digit extension of an acceptable one, the maximum length of the sequence is 33.

The above conventions serve as a basis for the more explicit statement of the property "acceptable". We have to characterize digit patterns as presented by the quintuples contained in the current value of the sequence. I propose to characterize such a digit pattern by the integer value one gets when interpreting the digits of such a quintuple as the digits of a binary number. In other words we define the function H(i) for $1 \leq i \leq k$

$$H(i) = d[i-4]*16 + d[i-3]*8 + d[i-2]*4 + d[i-1]*2 + d[i] \quad (2)$$

The property "acceptable" means that

for $1 \leq i,j \leq k$, $i \neq j$ implies $H(i) \neq H(j)$. (3)

At each moment the function H(i) is defined on the current value of sequence for $1 \leq i \leq k$. Instead of recomputing these function values whenever we need them (i.e. in the acceptability test) we can tabulate them in an integer array h[1:33].

Our convention is that for all sequence values will hold

$h[i] = H(i)$ for $1 \leq i \leq k$. (4)

This convention implies that modifications of the value of sequence in general includes updating of the array h in order to maintain relation (4).

The acceptability test is now -analogous to (3)- that for $1 \leq i,j \leq k$ $i \neq j$ implies $h[i] \neq h[j]$.

Next we exploit the fact that the only sequences to be subjected to the acceptability test are promising sequences, i.e. one-digit extensions of an acceptable sequence. For a promising sequence we can conclude that it is acceptable if and only if

$h[i] \neq h[k]$ for $1 \leq i < k$ (5)

i.e. when the last quintuple presents a pattern different from all the preceding ones.

This, again, would imply scanning, but we can repeat the trick and tabulate whether a certain digit pattern already occurs. The most elegant way is to introduce a boolean array in[0:31] where for $0 \leq m \leq 31$ in[m] means:

for an acceptable sequence: m occurs among h[1]...h[k]
for a promising sequence: m occurs among h[1]...h[k-1] (6)

Note. When the sequence is acceptable, each pattern can only be presented once and a boolean variable is sufficient to record whether it occurs. In a promising sequence h[k] may equal h[i] for $i < k$; therefore convention 6 distinguishes between acceptable and promising sequences.

We now give the final version of the program. What were names of primitives now occur as labels (either of statements or of expressions).

```
begin integer k; integer array d[-3:33];
      integer array h[1:33]; boolean array in[0:31];
      set sequence to first acceptable one:
      begin d[-3]:= d[-2]:= d[-1]:= d[0]:= d[1]:= 0; k:= 1;
            h[1]:= 0; in[0]:= true;
            begin integer m; m:= 1;
                  repeat in[m]:= false; m:= m + 1 until m = 32
            end
      end;
      repeat  if length of sequence equals 32: (k = 32) do
            begin accept sequence as solution: new line carriage return;
                  print solution:
                  begin integer m; m:= 0;
                        repeat print(d[m-3]); m := m + 1 until m = 32
                  end
            end;
            transform sequence into next acceptable one:
            begin transform acceptable sequence into next promising one:
                  begin k:= k + 1; d[k]:= 0;
                        h[k]:= 2 * h[k-1] - 32 * d[k - 4]
                  end;
                  while promising sequence non acceptable: (in[h[k]]) do
                        transform promising sequence into next acceptable one:
                        begin while sequence ends with a one: (d[k] = 1) do
                              remove final digit from sequence:
                              begin k:= k - 1; in[h[k]]:= false end;
                              replace final zero by a one:
                              begin d[k]:= 1; h[k]:= h[k] + 1 end
                        end;
                  accept sequence as acceptable: in[h[k]]:= true
            end
      until sequence is virtual last member: (k = 1)
end
```

In explanation:
1) "accept sequence as solution", that could have been an empty statement has been given the meaning of transition to a new line
2) "accept sequence as acceptable" has got contents, due to convention (6) where distinction is made between promising and acceptable sequences.

Concluding Rekarks.

We have shown successive program versions, leading from the original problem statement to the final program. In our final program, the merging of these successive versions has been done by hand and the more abstract versions have been reduced to comment in the form of labels.

For large programs this merging process itself becomes a major data processing task and I expect the growth of interactive program composition techniques in which the service of computers will be enlisted for the benefit of this process.

Furthermore: at present the more abstract versions are only reflected as explanatory comment, inserted for human understanding. The origin of this is that we want at present a program formulated at a constant semantic level, viz. the level of the programming language. For the more abstract versions we have at present during run time no mechanical use. In future I expect the more abstract

versions to be an integral part of the program.

Finally, we have considered only a single line of programs leading from the problem statement to a working program expressed at the desired semantic level. In future I expect that this single line will be extended to a more or less tree-structured class of programs in which is also room for alternatives, thus structurely tying together program composition and program modification.

Edsger W.Dijkstra
Department of Mathematics
Technological University
P.O.Box 513
EINDHOVEN
The Netherlands

(This paper has been produced in relation to a talk given at the University of Grenoble in December 1967.)