Preface.

The market is already so heavily overloaded with introductory texts on computers and computer programming that one must have rather specific reasons to justify the investment of one's time and energy on the writing of yet another "Short Introduction into the Art of Programming". The sole fact that one likes to write is, all by itself, an insufficient justification. Before undertaking such a task I should therefore ask myself "Why am I going to do it?" and also "What do I expect to be the distinguishing features of this little monograph?".

There is a simple, practical reason. At my University I teach, mainly for future Mathematical Engineers, an introduction into the art of programming and my students would welcome some underlying material. Besides that, without some form of lecture notes, my colleagues have very little idea of what I am really trying to teach! The contents of this course show signs of settling down -I have now given it three times- and therefore this seems the appropriate moment to produce a document that can serve as lecture notes.

These are purely local circumstances and as far as they are concerned, a normal set of lecture notes -in Dutch, say- would do. The fact that I have not chosen this form means that I am aiming at a larger audience. Such an act is always somewhat presumptuous and the usual author's trick to save the image of his modesty is to tell that from various sides he has been urged to produce his manuscript - a trick that I could apply in this case without lying. But I don't think that I shall resort to that trick because I really believe that a larger audience than just my students can benefit from it, or even enjoy it.

The fact is that over the last years I have addressed myself to the question whether it was conceivable to increase our programming ability by an order of magnitude and what techniques (mental, arganizational or mechanical) should then ne applied in the proces of program composition.. Personally, I felt these investigations very rewarding: I gained a much deeper understanding of the nature of the difficulty of the programming task, I became much more conscious about my"programming style", which improved considerably and, in general, found myself, when programming, in much better control of what I was doing than I had ever been before. Needless to say that my teaching was heavily influenced my these experiences.

The purpose of this little monograph is to assist the programming reader in cleaning up his own thinking, to transmit to him some mental disciplines by sticking to which he can avoid making his job unnecessarily difficult. It is born out of dissatisfaction with the usual kind of programming course, which now strikes me as the type of driving lessons in which one is taught how to handle a car instead of how to use a car to reach one's intended destination. This monograph is intended as a complement to such courses: I shall try to present programming - to quote N.Wirth- "as a discipline on its own merits, as a methodology of constructive reasoning applicable to any problem capable of algorithmic solution."

I expect the distinguishing feature of this little monograph to be its incompleteness, incompleteness in many, many respects.

It will not be self-contained in the sense that I assume my readers somewhat familiar with a decent higher level programming language. (This assumption is a direct consequence of the local circumstance that my students have had a modest prior exposure to the cleaner aspects of ALGOL 60.)

For those readers who identify the programmer's competence with a thorough knowledge of the idiosyncrasies of one or more of the baroque tools into which modern programming languages and systems have degenerated, the book will also be very incomplete, because I won't describe any programming language -not even the one I use- to that level of detail. I shall use some sort of programming language, say, as "a communication language", but not for the communication of algorithms, but for the communication of ways of thinking, as a vehicle for programming style.

In yet another respect, this little monograph will be very incomplete. As said above, I shall try to present programming "as a discipline on its own merits, as a methodology of constructive reasoning, applicable to any problem capable of algorithmic solution". At present, such a methodology does not yet exist in the full sense of the word, only elements of it have become apparent, others are just lurking begind our mental horizon. This, of course, is not very satisfactory, but it is a true reflection of the current, still rather poor state of the art. It is a consolation that no piece of scholarship ever reaches the state of perfection and I am telling myself that the conviciton that there is more to come is no justification for witholding what we have got.

It will also be incomplete as a result of the choice of the examples and the choice of the considerations. By necessity, the examples will be "small" programs, while the need for a discipline only becomes vital in the case of "large" programs.

Dealing with small examples in an ad-hoc fashion gives the student not the slightest clue as how to keep the construction of a large program under his intellectual control, I hope to deal with the small examples in such a fashion, illustrating how we can avoid unmastered complexity, that methodological extrapolation to larger tasks- which, alas, fall outside the scope of this little document- is feasible. The choice of considerations is also kept to a strict minimum: we restrict ourselves to programming for purely sequential machines and when we have to consider a trade-off question, we shall usually present this in the form of a trade-off between computation time versus store requirements. In this respect the document may strike the reader as very strongly dated, perhaps even out-dated by the time it appears in print. If so, I hope that I can justify my defense, that such a reader has failed to read between the lines: it is not so much the particular trade-off question chosen that matters, as the fact that the problem has been approached in such a fashion that we have made a conceptual framework in which such specific trade-off questions can be postponed until the appropriate moment. The only thing I can do at this stage is to urge my readers to read between the lines as much as possible. (If one tries to transmit ideas or methods, one can talk about them but that alone is insufficient, one must show examples illustrating them. When lecturing, it is my sad experience that after having dealt with a specific example, I find the attention of half my audience completely usurped by this example: they have forgotten that the example was only brought to their attention to illustrate something more general. This is a sad experience and no amount of prior warning that this misunderstanding is bound to happen if they are not careful, has ever enabled me to avoid it!) To put it in another way: it is my purpose to transmit the importance of good taste and style in programming, the specific elements of style presented serve only to illustrate what benefits can be derive from "style" in general. In this respect I feel akin to the teacher of composition at a conservatory: he does not teach his pupils how to compose a particular symphony, he must help his pupils to find their own style and must explain to them what is implied by this. (It has been this analogy that made me talk about "The Art of Programming".)

There is a further class of potential readers that will find this subject matter very incompletely dealt with, viz. those who identify the programmer's task with writing programs in, say, FORTRAN of PL/1. One of my implicit morals will be that such programming languages, each in their own way, are vehicles inadequate to guide our thoughts. If FORTRAN has been called an infantile disorder, PL/1 must be classified as a fatal disease.

Although I would love to do it, it is impossible to give a true acknowledgement, listing all persons whose relevant influence on my thinking I gratefully remember or should remember. With my apologies to all persons unmentioned I would like to make a few exceptions and list in alphabetic order: my mother mrs.B.C.Dijkstra - Kluyver, R.W.Floyd, C.A.R.Hoare, P.Naur, B.Randell, D.T.Ross, N.Wirth and M.Woodger. None of the persons listed. however, should in any way be held responsible for the views expressed (with the possible exception of my mother who is in some sense responsible for my existence).

Some fundamental notions.

In this section a number of notions will be introduced, because they are fundamental to the whole activity of programming. They are so fundamental that they will not be dissected into more primitive concepts. As a result this section will be a very informal one, analogy, metaphor and natural language (poetry, if I were able!) being the only available vehicles to convey their contents and connotations.

It is not unusual -although a mistake- to consider the programmer's task to be the production of programs. (One finds terms as "software manufacturing", proposals to measure programmer productivity by the number of lines of code produced per month etc., although I have never seen the suggestion to measure composer productivity by the number of notes, monthly scribbled onhis score!) This mistake may be at the heart of the management failure which is so apparent in many large software efforts. It is a mistake, because the true subject matter of the programmer's activity is not the program he composes, but the class of possible computations that may be evoked by it, the "production" of which he delegates to the machine. It seems more fruitful to describe the programmer's activity, instead of "making a program",as "designing a class of computations". In this connection it should be borne in mind that the more relevant assertions about programs -e.g. about their correctness or their resource demands- indeed pertain to the computations, rather than to the last thing that leaves the programmer's hands, viz. the program text. It is for this reason that, when introducing the fundamental notions, I will start at the side of the computations, with the "happenings in time".

The first notion is that of an action. An action is a happening, taking place in a finite period of time and establishing a well-defined, intended net effect. In this description, we have included the requirement that the action's net effect should be "intended", thereby stressing the purposefulness. If we are interested in the action at all, it will be by virtue of our interest in its net effect.

The requirement that the action should take place in a finite period of time is most essential: it implies that we can talk about the moment T0, when the action begins, and the later moment T1, when the action ends. We assume that the net effect of the action can be described by comparing "the state at moment T0" with "the state at moment T1".

An example of an action would be a housewife peeling the potatoes for the evening dinner. The net effect is that the potatoes for the evening dinner are at moment T0 still unpeeled, say in the potato basket in the cellar, while at moment T1 they will be peeled and, say, in the pan to be cooked in.

When we dissect such a happening as a time sequence of (sub)actions, the cumulative effect of which then equals the net effect of the total happening, then we say that we regard the happening as a <u>sequential process</u>, or <u>process</u> for short.

Whenever such a dissection is permissible, we can regard the <u>same</u> happening either as an action, or as a sequential process. When our interest is confined to the net effect, to the states "before and after", then we regard it as an action. If, however, we are interested in one or moreintermediate states as well, then we regard it as a process. In the latter case the moment T0 coincides with the begin of the first subaction, the end of each subaction coincides with the begin of the next one, with the exception of the last subaction, whose end coincides with T1, the end of the whole happening.

I must stress, that whether some happening is regarded as an action or as a process is not so much an inherent property of the happening, it expresses our mood, the way in which we prefer to look at it. (Why we should want to look at it in different ways, will be left for later discussion.) Similarly, if we have chosen to regard the happening as a process, the way in which it has been dissected is also not so much an inherent property of the happening, it expresses which of its distinguishable intermediate states (for some reason or another) we wish to take into consideration.

The happening of the potato peeling housewife could for instance be described by the time-succession of the following four subactions of the housewife:

"fetches the basket from the cellar;
fetches the pan from the cupboard;
peels the potatoes;
returns the basket to the cellar" .

Here the total happening has been described as a time-succession of four subactions. In order to stress that we have given the description of a happening we have phrased it in the form of an eye-witness account. Note, that if the eye-witness did not bother to describe that the basket was fetched from the cellar before the pan was fetched from the cupboard, the first two lines would have been condensed into a single subaction "fetches the basket from the cellar and the pan from the cupboard".

We postulate that in each happening we can recognize a pattern of behaviour or pattern for short; the happening occurs when this pattern is followed. The net effect of the happening is fully determined by the pattern and (possibly) the initial state (i.e. the state at moment TO). Different happenings may follow the same pattern, if these happenings establish different net effects, the net effect must have been dependent on the initial state as well and the corresponding initial states must have been different.

How we can recognize the same pattern in different happenings falls outside the scope of this text. If we meet a friend, we can recognize his face, no matter what facial expression he shows, it may be an expression we have never seen on his face before! Somehow we manage to abstract from the current expression, we recognize the same friend. Similarly with different happenings: we recognize the same pattern, abstracting from the possibly different initial states and net effects.

We return for a moment to the housewife. On a certain day she has peeled the potatoes for the evening dinner and we have an eye-witness account of this happening. The next day, again, she peels the potatoes for the evening dinner and the second happening gives rise to an eye-witness account equal to the previous one. Can we say, without further ado "Obviously, the two accounts are equal to each other for on both occasions she has done exactly the same thing."?.

How correct or incorrect this last statement is depends on what we mean by "doing the same thing". We must be careful not to make the mistake of the journalist who, covering a marriage ceremony, told that the four bridesmaids wore the same dress. What he meant to say was that the four dresses were made from material of the same design and -apart from possible differences in size- according to the same pattern.

The two actions of the housewife are as different ~~as the d~~ from each other as the dresses are: they have, as happenings, at least a different identity, one took place yesterday, one today. As each potato can only be peeled once, the potatoes involved in the two happenings have different identities as well, the first time the basket may have been fuller than the second time, the number of potatoes peeled may differ etc.

Yet the two happenings are so similar that the same eye-witness account is accepted as adequate for both occasions and that we are willing to apply the same name to both actions (e.g. "peeling the potatoes for the evening dinner").

An *algorithm* is the *description of a pattern of behaviour*, expressed in terms of a well-understood, finite repertoire of named XXXXX (so-called "primitive") actions of which it is assumed a priori that they can be done (i.e. can be caused to happen).

In writing down an algorithm, we start by considering the happening to take place as a process, dissected into a sequence of subactions to be done in succession. If such a subaction occurs in the well-understood, finite repertoire of named actions, the algorithm refers to it by its name~~s~~. If such a subaction does not occur in the finite repertoire,the algorithm eventually refers to it by means of a subalgorithm, in which the subaction, in its turn, is regarded as a process, etc. until at the end all has been XXXXXXXXX redüced to actions from the well-understood, finite repertoire.

The notion of an algorithm, an executable precept for the establishing of a certain net effect, is very well known from daily life: knitting patterns, directions for use, recipes and musical scores are all algorithms. And if one asks in an unfamiliar town for the way to the railway station, one asks essentially for an algorithm, for the description of a pattern of behaviour which, when followed, will lead to the desired goal.

In our description of an algorithm we have stressed that the primitive actions should be executable, that they could be done. "Go to the other side of the square." is perfectly OK, "Go to hell", hpwever, is not an algorithm but a curse, because it cannot be done.

Besides that we have stressed that the repertoire should be well-understood: between the one who composed the algorithm and the one who intends to follow it, there should be no misunderstanding about this repertoire. (In this respect knitting patterns are, as a rule, excellent, recipes are of moderate quality while the instructions one gets when asking the way are usually incredibly bad!) How essential this lack of misunderstanding is may perhaps best be demonstrated by a recipe for jugger hare as it occurs in an old Dutch cookery-book; translated into English the recipe runs as follows: "One taketh a hare and prepareth jugged hare from it." The recipe is not exactly wrong, but it is hardly helpful!

Let us now contrast the eye-witness account of the potato peeling session:

> "fetches the basket from the cellar;
> fetches the pan from the cupboard;
> peels the potatoes;
> returns the basket to the cellar"

with the corresponding algorithm -the set of instructions, say, the housewife might give to a new maid-:

> "fetch the basket from the cellar;
> fetch the pan from the cupboard;
> peel the potatoes;
> return the basket to the cellar" .

Comparing the two, we may well ask what we have gained, for it seems a roundabout way of doing: describing a pattern of behaviour which, when followed, will evoke the happening, while in the eye-witness account we had an excellent way of describing the happening itself.

What have we gained? Well, nothing, as long as we restrict ourselves to algorithms that can be given -as in our example- by a concatenation of names of actions, to be done in the given order. Under that restriction an eye-witness account of the actions "as they take place" is equally good. But the behaviour of the housewife (or the maid) could be a little bit more complicated: let us suppose that after the pan has been fetched, she puts on an apron <u>if necessary</u>, i.e. when she wears a light-coloured skirt and that on one day she uses the apron while on the other day she doesn't.

On a rather abstract level -i.e. without explicit mentioning of the apron and the condition under which it is used, a uniform eye-witness account would still do (in some fashion) for both sessions, e.g.:

> "fetches the basket from the cellar;
> fetches the pan from tbe cupboard;
> takes preparation as regards to clothing;
> peels the potatoes;
> returns the basket to the cellar"

with the implcit understanding that "takes preparation as regards to clothing" covers the empty action when her skirt is not light-coloured and covers putting on an apron when her skirt is light-coloured.

If, however, we want to go into more detail and want to mention the apron explicitly, then "takes preparation as regards to clothing" has to be replaced in the eye-witness account of the one day's session by

"sees that her skirt is light-coloured and therefore puts on an apron"

and in the other day's session by

"sees that her skirt is not light-coloured and therefore omits putting on an apron" .

The trouble is, that the eye-witness account cannot contain the single sentence:

"puts on an apron if her skirt is light-coloured"

for then the audience justly asks "does she do it or not?".In other words: in that degree of detail we cannot cover the two happenings by the same eye-witness account for in that degree of detail the two happenings differ!

It is here that the potential power of the algorithm becomes apparent, for we can recognize the same pattern of behaviour in the two happenings and by describing that pattern of behaviour we give something that is applicable under both circumstances, light- as well as dark-coloured skirt. This is possible thanks to the fact that what actually happens when a certain pattern of behaviour is followed may be co-determined by the state of affairs which is current when the action begins.

We see two things: the inspection whether the skirt is light-coloured and, depending on the outcome of this inspection, the action "put on an apron" is to take place or not. In order to express this conditional execution we need in our algorithm another connective besides the semicolon. In our example of the algorithm (I refer to the instructions to the new maid) the semicolon had a double function: in the text it separates one action name from the next action name but besides that it implied for the happening a certain amount of what is technically called "sequencing control", i.e. it was meant to imply that the end moment of the preceding action should co-incide with the begin of the following action. We now need another connective, indicating whether or not the inspection should be followed by the next action in the text. We write for instance the following algorithm:

```
"fetch the basket from the cellar;
fetch the pan from the cupboard;
if skirt is light-coloured do put on an apron;
peel the potatoes;
return the basket to the cellar"   .
```

(For historical reasons the so-called conditional connective "if ... do" is split into two symbols "if" and "do", enclosing the inspection.)

The conditional connective connects two actions, the first of which must be an so-called "inspection". This inspection describes a state of affairs, which may be true or false ("false" is the technical term for "not true"). The happening which is to take place corresponding the conditional compound

"if inspection do action"

may take one of two mutually exclusive forms: either the inspection gives the result true and it is followed by the action, or the inspection delivers the result false and thereby the whole compound action has been completed. The algorithm derives its superiority over the eye-witness account from the fact that it may contain connectives (such as the conditional connective) that imply a more elaborate sequencing control than the semicolon.

We need a further connective before we can see the full superiority of the algorithm over the eye-witness account, viz. a repetitive connective.

Suppose that we want to express that "peeling the potatoes" is in itself a process that deals with one potato at a time and that, correspondingly, our primitive action is named "peel a next potato". If the number of potatoes to be peeled is a fixed constant, say always 25, then we can replace "peel the potatoes" by 25 times "peel the next potato", separated from each other by in toto 24 semicolons. But we now assume that the number of potatoes to be peeled may differ from one day to the next; yet we want to recognize in each peeling session the same pattern of behaviour. We suppose the housewife capable of looking into the pan and judging whether the amount of peeled potatoes is sufficient.

If we know a priori that in the worst case (i.e. many guests and very small potatoes) she will never have to peel more than 500 potatoes we can give a general algorithm describing the actual peeling by repeating in the text of our algorithm 500 times (separated by in toto 499 semicolons) the conditional compound:

"if number of peeled potatoes is insufficient do peel a next potato" .

Several objections can be made against this solution. There is the practical objection that it would reduce the construction of algorithms to doing lines. Furthermore we had to make the fundamental assumption that we knew in advance a maximum number. Often it is very hard to give such an upper bound a priori and if it can be given, such an upper bound is usually many times larger than the average value. And if in actual fact 25 potatoes have to be peeled, the 26th inspection "number of peeled potatoes is insufficient" -i.e. the first one to deliver the result "false"- gives fresh information, the following 474 inspections (which are prescribed by the algorithm as suggested) give no new information. Once the housewife has established that the number of peeled potatoes is no longer insufficient, she should not be forced to look into the pan another 474 times in order to convince herself!

In order to meet these objections, we introduce a repetitive connective which, again for historical reasons, is written in two parts "while ... do". Using this connective we can write the algorithm:

```
"fetch the basket from the cellar;
fetch the pan from the cupboard;
if skirt is light-coloured do put on an apron;
while number of peeled potatoes is insufficient do
                                  peel a next potato;
return the basket to the cellar" .
```

The process corresponding to

"while inspection do action"

consists of one or more executions of the conditional compound

"if inspection do action",

viz. up to including the first time that the inspection gives the result "false".

We can also describe the semantics of the repetitive connective in terms of the conditional one recursively:

```
"while inspection do action"
```

is semantically equivalent with

```
"if inspection do
     begin action; while inspection do action end" .
```

Here the symbols "begin" and "end" are used as opening and closing bracket respectively; they are a syntactical device to indicate to indicate that the conditional connective connects the inspection (from the first line) to the whole of the second line: the value delivered by the first inspection decides whether what is described on the second line (from begin until end) will be done in its entirety or will be skipped in its entirety.

Insert, [illegible] 12a

As a final exercise I shall give the algorithm describing the pattern of behaviour of a housewife who -for some obscure reason- is so conditioned that she can only peel an even number of potatoes for the evening dinner:

```
"fetch the basket from the cellar;
fetch the pan from the cupboard;
if skirt is light-coloured do put on an apron;
while number of peeled potatoes is insufficient do
   begin peel a next potato; peel a next potato end;
return the basket to the cellar".
```

This example is included to show that the same set of primitive actions allows different patterns of behaviour.

The notion of an algorithm is a very powerful one, for a single algorithm "extracts" what a possibly large number of different happenings may have in common. And it does not do so by "abstraction", i.e. by ignoring the detailed differences as would become apparent in the different eye-witness accounts, no, it can describe the single pattern of behaviour to exactly the same degree of detail (i.e. in terms of the same primitive actions). The possibly large number of different corresponding happenings is generated by the different ways of sequencing as might be controlled by the conditional, repetitive (and similar, see later) connectives.

On the one hand we have the algorithm, a finite text, a static, timeless concept; on the other hand we have the corresponding happenings that may be evoked

Note. In the above I have approached the idea of an algorithm starting my considerations from the class of happenings in which we wanted to discern the same pattern of behaviour. This led besides the semicolon as connective in the text of the algorithm to other connectives such as the conditional connective "if...do" and the repetitive connective "while...do". It is not unusual that the relation between algorithm and computations is approached from the side of the algorithm; such an approach leads already in a very early stage to syntactical considerations as a result of which the connectives are introduced in a somewhat different terminology. Instead of

"if inspection do action"

people write

"if condition do statement".

The part of the text denoted by "if condition do" is then described as "a conditional clause", which is regarded as a prefix that is attached to the "statement". The statement prefixed by the conditional clause can be called "a conditional statement". Similarly, in

"while condition do statement" ,

"while condition do" is called "a repetitive clause" and the statement is called "the repeatable statement". This terminology is so widely used that -in spite of its syntactical origin- I shall not refrain from using it whenever I see fit to do so.

by it, dynamic concepts, happenings evolving in time. The intimate relation between the two -to which we refer by the term "sequencing"- lies at the heart of the algorithmic notion. (It is to this intimate relation to which I refer whenever I stress that the programmer's true activity is "the design of classes of computations.) The notion of an algorithm is admittedly a very powerful one; before going on, however, I shall allow myself a little detour in order to indicate what "price" we have paid for its introduction.

We have stated that we restrict ourselves to happenings taking place in a finite period of time. Whenever an algorithm is followed, a happening is taking place, eventually as a time-succession of primitive actions. It is only realistic to postulate that each primitive action will take a finite period of time, unequal to zero: no action will take place "infinitely fast". This implies that we confine our attention to happenings that are taking place as a time-succession of a finite number of primitive actions.

And now we are beginning to see the price: it is very easy to write down a text that looks like an algorithm but that is not an algorithm in our sense of the word, because the effeort to follow it turns out to be a never ending task, e.g.

"while skirt is light-coloured do peel a next potato" .

When we assume that the peeling of a next potato does not influence the colour of the skirt, we have just two cases: either the skirt is not light-coloured and the only action taking place is the inspection establishing this fact, or the skirt is light-coloured and will remain so and what the pattern suggests to describe is the peeling of an infinite number of next potatoes. This is usually called "an improper algorithm".

The question, whether a text that looks like an algorithm, is indeed a proper algorithm or not, is far from trivial. As a matter of fact Alan M.Turing has proved that ~~we cannot writ~~ we cannot write an algorithm capable of inspecting any text and establishing whether it is a proper algorithm or not.

The assumption of the existence of such an algorithm leeds to a contradiction which will be sketched below. Suppose that we have such an algorithm, an inspection

"proper(L)"

which delivers the result true when the text named L is a proper algorithm and the result false when it is improper.

Consider now the following text, named L

L: "*while* proper(L) *do* whistle once"

(in which "whistle once" is assumed to be an available primitive). If we start to follow this algorithm, how many times will be whistled? The assumption that "proper(L)" delivers the result true will cause the algorithm to be improper and vice versa! The conclusion is that no algorithm for the inspection "proper" can exist. (Marvin XXXX Minsky concludes in "Computation, Finite and Infinite Machines", Prentice Hall 1967 a formal treatment of this proof with the sentence "We have only the deepest sympathy for those readers who have not encountered this type of simple yet mind-boggling argument before.")

The moral of this story is that it is an intrinsic part of the duty of everyone who professes to compose algorithms, to supply a proof that his text indeed represents a proper algorithm.

Our next fundamental notion is a *machine* (or "a computer"). A machine is a mechanism capable of causing actions to take place following a pattern of behaviour as can be described by algorithms expressed in terms of a repertoire of primitive actions that belong to this machine.

Above, we have given two algorithms for peeling potatoes, one for a natural number of potatoes and one for only even numbers of potatoes. Both algorithms have been expressed in the same repertoire of primitive actions. They were introduced in the realm of "observing happenings", the one could describe the pattern of behaviour of my left-hand neighbour, the other the one of my right-hand neighbour. Suppose now that my own wife

1) is also capable of performing those primitive actions

2) will accept from me algorithms expressed in these primitives and will follow such an algorithm obediently.

Then I can make her peel either as my left-hand neighbour or as my right-hand neighbour, depending on the algorithm I have supplied to her. Then she is an example of a machine.

A mechanism that can only do one thing (such as one of the most widely-spread automata, the toilet flusher) is not called a machine. Essential for us is the associated repertoire of actions, the ability to accept patterns of behaviour and to behave accordingly.

Machines are mechanical algorithm followers. The fact that in the last decennia increasingly powerful machines have become available to mankind is directly responsible for the increased importance of and interest in algorithms and their composition.

Note. It is a trivial matter to compose an algorithm for the fastest machine in the world, a proper algorithm in the theoretical sense of the word but somewhat impractical, as it would take the machine a million years to carry the corresponding process to completion. The claim that "the machine is capable of causing the process to take place" is then somewhat subject to doubt: in actual fact in cannot. In what follows we shalln't be bothered by the distinction between "theoretically possible" and "practically feasible". Not because we are impractical, on the contrary! The point is that in the mean time computers are so powerful that the class of practically feasible computations is by now sufficiently large -to put it mildly!- to make machines very useful and intriguing pieces of equipment, fully worthy of our attention.

We call an algorithm intended to control the behaviour of a machine, a program. In other words, we reserve the name program for those algorithms that are intended for mechanical execution. In the general notion of algorithm we have only required that the repertoire should be "well-understood", without bothering how this understanding is established: if a composer indicates "Andante" (= "going") for a composition in three-four time, he may do so, because, remarkably enough, we may expect this indication to be somehow meaningful for the player. In the case of a machine, the situation is drastically different for a machine is a finite piece of equipment which by its very construction has associated with it a finite, very well defined repertoire and whenevr it is fed with a program it shall behave exactly as prescribed.

The fact that machines are fully obedient slaves has caused complaints from many beginning programmers. Their obedience, they felt, makes programming cruelly difficult, for a trivial mistake in the program is sure to lead to entirely unintended behaviour. The programmer's inability to appeal to "common sense of the machine" has been experienced as one of its major shortcomings. The more experienced programmer learns to appreciate its servile, strict obedience: thanks to it we can instroct it to do something "uncommon"! And this is something you cannot do with a servant who "rounds off" his instructions towards the nearest probable interpretation.

In the preceding paragraphs I have introduced programming as an important activity because now we have machines that can be controlled by programs and for which we have to compose programs when we want to use them, when we want to convert them into the tool solving our problem. But this is not the whole story. A computer is a many-sided thing. For its manufacturer it is primarily a product that he can (hopefully) produce and sell with profit. For many insitutional buyersthe computer is probably primarily a status symbol. For many users it is either a source of endless worries or, as the case may be, a highly useful tool. In University surroundings, the view of the computer as a tool to be used tends to be the predominant one. And there is no denying it: in their capacity of tools the computers are changing the face of the earth (and of the moon as well!). Yet I am comvinced that we underestimate the computer's significance if we confine our appreciation of it to the aspects mentioned. They may cause shocks to the basis of our society, but I believe that in the longer run these will turn out to be but ripples on the surface of our culture. I expect a much more profound influence from the advent of the modern computer and that will in its quality of a gigantic intellectual challenge, unprecedented in the history of mankind.

The computer as piece of equipment presents us with an entirely new combination of simplicity and power, which makes the programming task a unique challenge. When the electronic engineers have done their job properly, they present to the programmer a mathematically trivial piece of equipment. Its instruction code (its "repertoire") can be described perfectly well on a modest number of pages, everything is finite and discrete, there is just no place for ~~mathematically~~ conceptually difficult mathematical notions, such as continuity, infinity, limits, irrational numbers and whatnots. The mathematical basis of programming is just very, very simple, so simple that programming should be easy: it should be easy to conceive programs, it should be easy to convince oneself that a program is correct and that the machine working under control of it will indeed produce the desired result. From its basic simplicity one derives the intuitive feeling that it should be a trivial matter to keep the happening evoked by one's programs firmly within one's intellectual grip.

But its basic simplicity is only one side of the coin: the other side presents the extreme power (both as regards capacity and speed) of currently available computers. As a result of its extreme power, both the amount of information playing a role in the computations as well as the number of operations performed in the course of a computation, escape our unaided imagination by several orders of magnitude.

Due to the limited size of our skull we are absolutely unable to visualize to any appreciable degree of detail what we are going to set in motion and programming thereby becomes an activity facing us with conceptual problems that have risen far, far above the original level of triviality.

It is the possibility to consider realistically effective solutions of any degree of sophistication, combined with our complete intellectual grip on what we are considering, which will deeply influence our ways of thinking and our appreciation of our own thought processes. It has no precedent, for whenever in the past we were faced with something powerful (a storm or an army) we never had effective control over it. (This, for a long time, used to be the definition of "powerful", viz. that we were "powerless" in the face of it!)

Programming Languages and their Implementation.

The activity of composing programs is called "programming". In the preceding section we have introduced programs as algorithms intended to control the behaviour of machines and by virtue of the actual existence of such machines, programming is a very practical activity. It is one of the youngest branches of applied mathematics (in the broad sense of the word, i.e. not confined to mathematical physics or numerical analysis), it is as important as the applications in question, it is practical in the sense that it is the programmer's intention that a machine will actually display the behaviour as prescribed by the algorithm. For that reason a conscious programmer should respect the limitations of the (finite) machine. Alternative programs causing a machine to establish the same net result and therefore in that respect equivalent, may differ greatly in what is usually called "efficiency", in the demands they make upon the machine's resources. For many years, efficiency has been used as the sole yard-stick along which to compare the relative quality of alternative programs for the same task. In the mean time programming has turned out to be so difficult, that other quality aspects have gained relative importance, such as the ease with which we can understand a program, can convince ourselves of its correctness or can modify it, etc. Yet, efficiency concerns cannot be ignored and in order to give the reader some feeling for the nature of the limitations he should respect, we shall give in this section an overall view of computing machines and the way in which the execute programs.

In this little monograph we shall confine our attention to sequential algorithms, i.e. algorithms describing what actions should happen in succession, one after the other. Such algorithms have a property for which they have been blamed (and not entirely without justification), viz. that they are often"overspecific" as regards the order in which things have to happen. If two actions, say "A" and "B" have to be dome both, a purely sequential algorithm will prescribe

either "A; B" or "B; A"

viz. action A in time followed by action B or the other way round. It will not express that the order is immaterial and possibly more serious, it will not express that the two actions are so "non-interfering" with each other that they may take place concurrently, or -to use the jargon- may be done in parallel.

For various reasons I decided to restrict my attention to purely sequential programs. The most obvious reason is to be found in the structure of the machines that are currently available or can be expected to become available in the next years. One or two decades ago, machines used to be purely sequential. In the mean time we have got equipment allowing for a limited amount of parallelism (dual processor machines, independent communication channels etc.) but such pieces of equipment are at best an aggregate of a small number of individual sequential components. In such machines the potential patallelism of activities is exploited by standard control programs (so-called "operating systems"), while the individual user still works in a strictly sequential environment. And it is to the individual user that this little monograph addresses itself.

With the advent of what is called "large scale integration" (being a term from the computer field, its acronym "LSI" is better known!) it seems to become technically feasible to build machines more like "clouds of arithmetic units" with information processing activities going on simultaneously all over the place, for shorter periods of time even undependently of each other. Programming for such machines will pose completely different trade-off problems: one will be willing to invest potentially useful computing activity before its actual usefulness has been established, all for the sake of speeding up the whole computation. But although I know that such machines may be coming, I shall not touch these problems for the following reasons. First, as far as general purpose applications are concerned, I have my doubts about the effectiveness with which such forms of parallelism can ever be exploited. Second -and that is the most important consideration- parallel programming is an order of magnitude more difficult than sequential programming. (This statement will be doubted but I have enough experience in multiprogramming to feel myself entitled to say so. The point is that with parallelism a great variety of happenings may take place under control of the same program(s). On account of undefined speed ratios a set of parallel programs is written for a partly non-deterministic machine and special care is required to ensure that, on a higher level of abstraction, its total behaviour can again be regarded as uniquely determined by the program(s).) Third, I am not over-impressed by the complaints that sequential programs specify a more stringent time-succession than logically necessary: I have often the somewhat uneasy feeling that these complaints find their origin in the mathematical tradition of the pre-computer age. In classical mathematics the notion of an algorithm has been neglected; mind you, I am not blaming the previous mathematicians for this, because before the actual existence of automatic computers, algorithms were hardly a relevant subject. But we should not close our eyes ~~for~~ to the fact that the course of history has caused mathematics to be more tuned to timeless problems, to static relations, to functional dependence. (The existence of classical mechanics does not contradict this observation: renaming the independent variable in the differential equations "k", say, instead of the usual "t" does not influence the mathematics involved.) Some of the efforts to remove the overspecification of the time-succession -they rely heavily on functional dependence- strike me as tackling the programming problem with classical concepts that have been developed for other purposes. So much for my decision to restrict my considerations to sequential machines.

To get some feeling for the demands made upon the modern automatic computer, let us focus our attention for a moment upon an average sizeable computation, for instance the computation of (a good approximation of) the inverse of a given matrix of, say, 100 by 100 elements. Such a job has two markedly quantitative aspects:

a) a vast amount of numbers is involved: posing the problem implies the specification of 10.000 numbers, the answer is also given by 10.000 numbers (each of which is, in general, a function of all 10.000 elements of the given matrix)

b) a vast amount of computation has to be done+ if it is done by elimination, the number of operations (i.e. multiplications and additions) is of the order of magnitude of 1.000.000.

The construction of machines able to cope (reliably!) with these two very different aspects of "multitude" is one of the greater triumphs of electronics. It has been achieved by applying the old and well-known principle "Divide and Rule.". In modern computers one can distinguish two vital components, each of which has the specific task to cope with one of the forms of multitude.

a) the store (called "memory" in American); this is the component able to receive, store and return vast amounts of information; its primary function is to be large, to be able to contain very much information

b) the arithmetic unit or processor; this is the component in which the actual work -adding, subtracting, multiplying, comparing, chosing etc.- is done; its primary function is to be very fast so that it may do very much in a limited period of time.

It is not the function of the arithmetic unit to be large in the sense that it should contain large amounts of information. On the contrary: while nearly all the information, relevant for the computation at large, lies "sleeping" in the store, at any moment of time only the tiny fraction actually involved in the information processing activity, is found (copied) in the arithmetic unit, which is only capable of dealing with a few numbers at a time, say the two numbers to be added and the sum formed by the act of addition. Whenever two numbers (in store) are to be added, they are transported from the store to the arithmetic unit, where the sum will be formed; once formed the sum will either be kept in the arithmetic unit for immediate further processing or it will be sent back to store for later processing. Microscopicly, the store acts as the icebox in which all information is kept which is not involved in the current activity of the arithmetic unit. If small letters indicate variables in store and R indicates a register in the arithmetic unit, the computation

"x := (a + b)*(c + d)"

might be evoked by the following sequence of instructions:

```
R:= a;
R:= R + b;
t:= R;
R:= c;
R:= R + d;
R:= t * R;
x:= R          .
```

The first instruction fetches the value of a from store into the register R, the next one increases (the contents of) R by the value of b (from store). At this stage one of the two factors to be multiplied has been computed. Before the multiplication can take place, the second factor has to have been computed as well; in a machine with a single register R for arithmetic results, this second addition implies again the use of the register R. In order to make this register available for this purpose, the third instruction sends the value of the first factor -a so-called "intermediate result"- back to store, assigning it to a variable here named "t": the first sum is sent back to store for later usage. The fourth and the fifth instruction compute the second factor the value of which is left in R, ready for multiplication by the temporarily sored value called "t". The last instruction stores the product, now formed in R, so that it can be retrieved under the name "x" for later usage.

The above example illustrates many things. It shows how

"x:= (a + b)*(c + d)"

which on one level of interest can be regarded as a single action, on closer inspection turns out to be a sequential process taking place as a time-succession of seven more primitive sub-actions ("program steps"). It also shows that at any moment of time only a tiny portion of the algorithm is in active control of what actually happens+ while the first or the second addition are performed, the fact that the two sums will have to be multiplied is still "dormant". (If the total action would have been

"x:= (a + b)/(c + d)"

the only modification necessary would have been the replacement of the sixth instruction by

"R:= t / R" ,

the first five instructions would have been insensitive to this change. That is what I menat by "dormant".)

It also demonstrates that, just as at any moment of time, only a tiny fraction of the num~~b~~erical information is involved in actual processing, also only a tine fraction of the program exercises control, viz. the instruction currently executed.

It also demonstrates that it is no good just to divide the machine into two components, store and arithmetic unit, one must also provide for a (dense) information traffic between the two: this is provided for by what connects the two together, the so-called "selection".

We have said that the store should be able to strore information, it must, for instance, be able to store "numbers", e.g. the intermediate result called "t". Obviously, these numbers cannot be kept in store like balls in an urn; when the instruction

"R:= t * R"

has to be executed, the store must return not just any number, but quite definitely it must return the value sent to it two instructions before. For that reason the numbers in store are not arranged as balls in an urn, on the contrary! Store is arranged as a number of so-called "storage cells", each capable of holding the value of one number at a time. Each storage cell is identified by its so-called "address", each time contact with the store is required -either to receive or to return information- this request is accompanied by a statement of the address of the storage cell involved. If the store is to receive information -this is called "writing into store"- the value to be stored and the address of the storage location involved (plus a "write request") are sent to store and selection respectively; as a result of the writing operation the original contents of the storage cell, which gets lost, are replaced by the new value. If the store is to return information -this is called "reading from store"- the address of the storage cell involved (plus a "read request") is sent to the selection; as a result the contents of the storage cell are returned from store (and kept in the storage cell as well for later reading if desired). As far as destruction, reception and reproduction of the information contained in a storage cell are concerned, the situation shows great analogy to a tape in a tape recorder. You can use the tape to record as many pieces of music as you want, but only one at the same time: whenever you record a new piece of music on an old tape, its previous contents are wiped out; the piece of music currently recorded, however, can be played back as many times as you wish. (To make the analogy a true one, we must restrict ourselves to pieces of music of equal duration, just matched to the length of the tape, matched to its (finite) information capacity.)

Storage cells can store information by virtue of the fact that they can be in a finite number of distinct states. In practically all computers they are composed from elementary components, each of which can be in one of two possible states. (The most common form is a little ring of ferromagnatic material that will be circularly magnetized in one of the two possible directions.) One such component can be in 2 different states (say "North" and "South"), two such components can be <u>together</u> in 4 different <u>total</u> states ("North-North", "North-South", "South-North" and "South-South"), N such components together can be in 2^N mutually different states. The number of elementary components associated with each storage cell is a characteristic constant of the machine's store and is called "the word length". If the word length is 32, the [illegible] number of different possible total states per word is 2^{32}, i.e. slightly over $4*10^9$; the arithmetic unit will associate with each state a numerical value, in terms of these numerical values a storage cell can then hold, for instance, any integer value ranging from (roughly) $-2*10^9$ to $+2*10^9$.

The finite capacity of the storage cell is something the user must be aware of: it is matched to the abilities of the arithmetic unit, i.e. if the latter deals with integer values it is geared to operations up to a certain maximum value, if it deals with (approximations of) real numbers, it is geared to dealing with them in a certain precision, maximum absolute value and precision respectively being chosen such that the numerical values to be distinguished between can be stored in one (or possibly two successive) storage cells. If greater integers or reals in higher precision have to be manipulated, special measures have to be taken which will be more expensive.

In the mean time we have explained enough about the general machine structure to mention two aspects of the "costs" involved in the execution of a program. One of them is computation time. We have seen that the arithmetic unit performs one operation after the other, and the more operations a program prescribes, the longer the total amount of time the arithmetic unit will have to spend to carry the computation to completion. The other one is storage usage. If a thousand values have to be computed in order to be added together, we may compare the following two algorithms. The first one first computes all thousand values and stores them, after which they are added, the second algorithm immediately adds each number to the partial sum as soon as it has been computed. With regard to storage usage the first algorithm is more demanding: at some stage of the computation it requires a sufficient amount of store to hold all thousand values, an amount of store which in the second algorithm remains available for other (possibly more useful) purposes.

So much for the finite capacity of each storage cell and the fact that a store contains only a finite number of such cells. Let us return to their addresses: a while ago we have hinted that each storage cell is identified by an "address" and that each reference to store takes place under control of the address of the storage cell concerned, but up till now we have not been very explicit about what an address really is. Well, this is very simple: the storage cells are numbered: 0, 1, 2, 3, 4, 5, ... up to M-1 if the store comprises Mdifferent storage cells (M between 16.000 and 1.000.000 being typical figures), and the ordinal number of each storage cell is used as "its address" (like houses in a street!). This implies that the storage cells have a natural order, viz. the order of increasing address. Given the address of a storage cell, the address of the next storage cell can be computed by adding 1 to the given address of the preceding one.

This natural oredring of storage cells is heavily exploited. If a vector, i.e. a sequence of numbers $a_0, a_1, \dots , a_n$ has to be stored, its elements can be stored in successive storage cells. If the address of the element a_0 is known, the address of element a_i can then be computed, viz. by adding (the value of) i to the address of the element a_0.

The natural order of the storage cells is also exploited in the program representation. Mind you, we have postulated that a machine could "accept" a program and that, once the program had been accepted, the machine could execute the program (i.e. cause the happening as prescribed by the program to take place). In other words, when the machine is executing the program, this program -i.e."the information describing how to behave"- must be somewhere in the machine! Where? Well, in the store, the store being specifically the machine component able to hold information. In other words, the store is used for two different purposes:

it holds the numerical information to be manipulated, but it also holds - in some other part of it- the program, i.e. the information describing what manipulations have to be performed.

To sketch briefly how this can be done, we return to our previous example where

"x:= (a + b)*(c + d)"

was decomposed into the sequence of seven instructions denoted by

```
"R:= a;
 R:= R + b;
 t:= R;
 R:= C;
 R:= R + d;
 R:= t * R;
 x:= R"
```

and the question is: by means of what conventions do we represent the above information in a store capable of holding numbers? This is achieved by a two-stage convention, one for XXX representing single instructions and one for representing a sequence.

The first convention is to chose for each instruction a unique number code. In the above notation we have denoted variables (or: the addresses of the storage cells associated with the variables and containing their current value) with small letters (a, b, c, d, t and x); but addresses are numbers and that component therefore is already numerical. Using "s" for "any address", we see that in the above example, we can distinguish instructions of four different types:

```
1)                   R:= s
2)                   R:= R + s
3)                   R:= s * R
4)                   s:= R
```

The second part of the convention associates with each type of instruction a number (e.g. the numbers 1, 2, 3 and 4 to the types shown above; it should be mentioned that in actual machines the number of instruction types is considerably larger than 4). By concatenating the digits describing the instruction type number with those giving the address we have a number code for each~~t~~ possible instruction and we assume that the word length of the storage cell is sufficient to contain such a number. The first convention as just described, reduces the problem of storing a sequence of instructions to storing a sequence of numbers. Now the second convention is that the sequence as such is represented by storing these number in successive storage cells, i.e. storage cells with successive addresses. And this completes (roughly) the trick.

(Note: This is by no means the only way to represent programs inside the machine's store; in so-called "stack machines" other conventions are chosen. The above elaboration is only shown by way of example, demonstrating the possibility.)

The dual role of the store -storage of instructions and storage of variables- implies another way in which a program can be expensive to execute: if the program text is very long, by that very fact the program text will make a heavy demand on storage capacity. If we have two alternative programs for the same job, one requiring 5000 instructions to describe it and the other requiring 10000 instructions to describe it, then -all other things being equal- the first alternative will be cheaper.

The above concludes our bird's eye view of the so-called hardware machine, i.e. the physical piece of electronic equipment as it is delivered by the manufacturer: a machine that can accept and then execute programs written as long sequences of instructions from an instruction repertoire that is specific for this particular machine (and its copies). Until the late fifties programmers indeed produced their programs as long sequences of such instructions, but when machines became faster and when more came on the market, the shortcomings of this way of working became more and more apparent.

Because the programmer expressed his program in terms of the instruction repertoire of that particular machine, he was fa~~r~~ced to tailor his program to that machine. He neede a thorough and ready knowledge of all the details of the instruction repertoire of that machine -which for the more intricate machines was no mean task- and worse: once his program was written, it could only be executed by that particular machine. Exchange of programs between institutes equipped with different machines was impossible; furthermore, whenever an institute replaced its old machine by a new and different one, all programs for the old machine became obsolete. From that point of view it was clear that tailoring one's programs so closely to the idiosyncrasies of a specific piece of hardware was not a very wise investment of the intellectual energy of one's programmers.

But even without the problem of transferring programs from one hardware machine to another, this way of programming, expressing programs as a monotonous stream of machine instructions, showed great drawbacks. One serious drawback was that this close contact between programmer and physical machine structure did not only enable the programmer to lard his program with all sorts of coding tricks, it actually invited him to do so. For many a programmer this temptation became irresistible, there even has been a time when it was generally believed that one of the most vital assets of a virtuoso programmer was that he ~~n~~e "puzzle-minded", and it was only slowly recognized that a clear and systematic mind was more essential! When "tricky programming" was en vogue, programming was not only very expensive -it took too much time- it also turned out to be too difficult to get a program correct. Looking backwards, the period of tricky programming now strikes us as a generation of programmers walking on a tight-rope, in full confidence because they were unaware of the abysmal depth beneath it! The modern competent programmer is more humble and avoids clever tricks like the plague.

It was not only the preponderance of coding tricks that made programming "in machine code" as it is called nowadays, too difficult and too risky.

Firstly, a program in machine code contains very little redundance and as a result it is very sensitive to even small writing errors -errors of the level of "spelling mistakes" or "printing errors".

Secondly, the programmer who thinks in terms of variables has to denote these variables in his program text by the addresses of the storage cells dedicated (by him) to hold their values. As a result the programmer has the burden of storage layout and all the clerical work implied by this.

Thirdly -and this is the major reason- machine code is an improper vehicle to represent "structure": it is just a single, monotonous sequence of machine instructions, incapble of expressing in a direct and useful form the structure of the algorithm. In what follows it will become abundantly clear that when we wish to compose programs in a reliable fashion, we can only do so by structuring the happenings we intend to evoke and are in urgent need of a descriptive vehicle such that in the program text itself the structure of the happenings -i.e. of the computations- can be adequately reflected.

The above shortcomings led to the design of so-called "(hihger level) programming languages". A programming language can be regarded as the machine code for a fictitious, idealized machine. Whereas the old machine codes were tailored to the needs of the hardware -i.e. the equipment electronic engineers could make- programming languages are more tailored to the intellectual needs and conceptual difficulties of the programmer who has to design the computations.

The problem now is that on the one hand we have the hardware machine A, that can be built but for which we don't like to program because it is too cumbersome, on the other hand we have "dreamed up" the fictitious machine B, for which we would love to program but which the engineers cannot build.
How do we bridge that gap?

The gap is bridged by "software": given machine A, we can make, once and for all, a program (in the machine code for machine A) which prescribes to machine A the pattern of behaviour it should follow if it is to simulate machine B. Such a program is called "software for machine A". Given hardware A, loaded with this software, we have a mechanism -partly "hard", partly "soft"- that is able to execute programs written for the fictitious machine B.

Usually this combination of hard- and software processes such a program in two stages. In the first stage (the "translation stage") the program written in the programming language B is subjected to a translation process. In this process a storage layout is decided, the necessary bookkeeping is carried out and an equivalent program -but now expressed in machine code A- is produced. In the second stage (the "execution stage") the output of the first one is interpreted by machine A as a program and the intended computation is evoked.

The standard software that goes with the machine shields the user off from idiosyncrasies of the specific machine; apart from that it invokes -behind the user's back, so to say- standard ways of dealing with the tougher properties of the hardware, such as the possible parallellism (i.e. concurrence in time) of computation proper and information transfers from and to peripheral devices and multilevel stores. Up till now we have described the hardware as if all storage cells were equally well accessible for the arithmetic unit. In practice this is seldom the case, two storage levels being quite common: primary store (ferrite cores usually) and secondary store (magnetic drums usually). The cells in primary store are the only ones that are directly and immediately accessible for the arithmetic unit; the informtaion in secondary store (which in capacity is an order of magnitude larger than primary store) is not directly accessible for the arithmetic unit, but the possibility of bulk transfers between primary and secondary store is available instead. In such machines, the software may move the information around between the two stores, all the time keeping track of where everything is to be found at any moment and trying to keep in primary store all "currently relevant" information. (This is called "the implementation of a virtual stote.)

We have mentioned the concept of the virtual store because it is related to an efficiency aspect over which the programmer has some control and in regard to which the programmer therefore has some responsibility. This is called "vagrancy". A program has a small degree of vagrance whenever for larger periods of time accesses are confined to a small, dense subset of the total amount of information; in that case the hope is justified that during that period of time, this dense subset will be kept in primary store ant that therefore the computation can go on at full speed. In computations with high vagrancy, the probability of information needed being in secondary store is much larger and the transport facilities between the storage levels then tend to become the bottle-neck. Therefore, if possible, high vagrancy should be avoided.

Variables and relations between their values.

When introducing the basic notions we have said that *different* happenings could take place following the *same* pattern of behaviour. And as the happening is fully determined by the confrontation of the pattern of behaviour with the initial state, it follows that the same pattern of behaviour can only evoke different happenings at different occasions when at these different occasions the initial states differ from each other. In this section we shall show how so-called *variables* are used for the description of the (initial and final) states. We find the typical use of variables when the same pattern of behaviour is followed repeatedly, i.e. when sequencing is repetitively controlled.

We begin with a very simple program: given two positive integer values A and B, a program has to be made that will compute (i.e. can cause a conputer to compute) the Greatest Common Divisor of A and B. Let us use the notation GCD(A, B) for that value.

(Remark. We have restricted ourselves to positive numbers in order to make life somewhat easier. Zero is divisible by any positive integer d (for $0 = 0 * d$), and there would be no objection, with $B > 0$, to

$$GCD(0, B) = B \quad .$$

But admitting zero as argument is asking for trouble, because GCD(0, 0) is clearly undefined! In order to avoid these complications, we restrict ourselves to positive arguments.)

For the sake of argument we request an algorithm in which no arithmetic operations other than addition and subtraction will be used. How do we find such an algorithm?

Well, there seem in this case two ways to attack the problem. The first one is more or less a direct application of the definition. One could construct a table of divisors of A (including 1) and a table of divisors of B (also including 1); as both A and B are finite and different from zero, both tables contain only a finite number of numbers. From these two tables of divisors one can construct a third table of common divisors, i.e. containing all the numbers occurring in both of them. This third table is non-empty (because it contains the number 1) and it is finite (because it cannot be longer than any of the original tables). From this non-empty, finite table we can select the greatest number and that will, by virtue of the way in which it has been found, be the Greatest Common Divisor.

We could do it along the lines just sketched (or at least use it as a source of inspiration). In the current example, however, there is a second way of attack necause the GCD is a well-known mathematical function, well-known meaning that a number of its <u>properties</u> are known. If we can think of so may of its properties that they <u>define</u> the GCD -i.e. that the GCD is the only function satisfying them- we might try to determine GCD(A, B) by exploiting these properties. What properties can we think of?

1) $GCD(a, b) = GCD(b, a)$

2) $GCD(a, b) = GCD(a + b, b) = GCD(a, a + b)$

3.1) if $a > b$: $GCD(a, b) = GCD(a - b, b) = GCD(a, a - b)$

3.2) if $a = b$: $GCD(a, b) = a = b$

3.3) if $a < b$: $GCD(a, b) = GCD(a, b - a) = GCD(b - a, b)$

(We can think of other properties such as

4) for $n \geq 0$ $GCD(a^n, b^n) = GCD(a, b)^n$

5) for $c > 0$ $GCD(c * a, c * b) = c * GCD(a, b)$

but they look less promising as they involve multiplication and we have to restrict ourselves to the arithmetic operations of addition and subtraction.)

The first property states that GCD is a symmetric function. The second one states that the GCD of two numbers is equal to the GCD of one of them and the sum, while the third property states this for the difference. Because we want to restrict ourselves to positive numbers, we have dealt with the cases $a < b$ and $a > b$ separately. The case $a = b$, however, is very special: it is the only case in which the value of the GCD is given directly!

Relations 2, 3.1 and 3.3 tell us that the GCD of a pair of numbers is equal to the GCD of another pair of numbers. This suggests that we use the "current state" to fix such a number pair; the algorithm can then try to change these numbers in such a way that

firstly: the GCD remains constant

secondly: until eventually the two numbers are equal and rule 3.2 can be applied.

With the second requirement in mind, rule 2 does not look to promising: given two positive numbers, one of them can never be equal to their sum. On the other hand, given two (different!) positive numbers, one of them, viz. the smallest, can be equal to their difference. This suggests that from 3.1 and 3.3 we use:

3.1' if $a > b$: $GCD(a, b) = GCD(a - b, b)$

3.3' if $a < b$: $GCD(a, b) = GCD(a, b - a)$.

Now the moment has come to consider our first version of the program:

program 1:

```
begin integer a, b, gcd;
      a:= A; b:= B;
      while a ≠ b do
            if a > b then a:= a - b
                     else b:= b - a;
      gcd:= a;
      print(A); print(B); print(gcd)
end
```

(In this program we have used the well-known alternative connective "if...then...else". The construction

"if inspection then action1 else action2"

causes one of the two actions, either action 1 or action 2 to take place. If the inspection delivers the value true, action1 will take place (and action2 will be skipped) if the inspection delivers the value false (action1 will be skipped and) action2 will take place. We can describe the conditional connective in terms of it:

"if inspection do action" is equivalent with "if inspection then action else nothing".)

When we try to understand this program we should bear the following in mind:

> While the typical use of *variables* manifests itself with the program loop, the way to understand such a program implies looking for the relations between their values which remain *invariant* during the repetition.

In this example the invariant relation P is

P: $a > 0\ \underline{\text{and}}\ b > 0\ \underline{\text{and}}\ GCD(a, b) = GCD(A, B)$.

The relation P holds after initialization (for the $a = A$ and $b = B$; from $A > 0$ and $B > 0$, relation P then follows).

The repeatable statement will only be executed under the additional condition $a \neq b$; i.e. either $a < b$ or $a > b$. If $a > b$, then the new value of a, viz. $a - b$, will again be positive and $GCD(a, b)$ will remain unchanged on account of 3.1'; if $a < b$, then the new value of b will again be positive and $GCD(a, b)$ will remain unchanged on account of 3.3'. The invariance of relation P is therefore established.

When the loop terminates, $a = b$ is guaranteed to hold, $GCD(A, B) = GCD(a, b) = GCD(a, a)$ and on account of 3.2 the assignment "gcd:= a" will establish the net effect "gcd:= GCD(A, B)".

To complete the proof we must demonstrate that the repetition will indeed terminate. Whenever the repeatable statement is executed the largest of the two (different!) values is decreased by the value of the other which is positive; as a result

$$\max(a, b)_{T0} > \max(a, b)_{T1} \quad .$$

We also know that before the repetition $\max(a, b) = \max(A, B)$ is finite, from the invariance of relation P ($a > 0$ and $b > 0$) we conclude that

$$\max(a, b) > 0$$

will continue to hold. All values being integer, the maximum number of times the repeatable statement can be executed must be less than $\max(A, B)$ and therefore the repetition must terminate after a finite number of repetitions. And this completes the proof.

Once we have this program, it is not difficult to think of others. Reading XXX program 1 we observe that each subtraction is preceded in time by two test, first $a = b$ and then $a > b$; this seems somewhat wasteful as the truth of a $>$ b already implies the truth of a $\neq$ b. What happens in time is that a number of times a will be decreased by b, then b will be decreased a number of times by a, and so on. A program in which (in general) the number of tests will be smaller is

program 2:

```
begin integer a, b, gcd;
      a:= A; b:= B;
      while a ≠ b do
      begin while a > b do a:= a - b;
            while b > a do b:= b - a
      end ;  gcd:= a; print(A); print(B); print(gcd)
end
```

Exercise Prove the correctness of program 2.

Exercise Rewrite program 2 such that the outer repeatable statement contains only one loop instead of two.Prove its correctness.

Before going on, it is desirable to give a more formal description of the kind of theorems which we use. (In the following I shall make use of a formalism introduced by C.A.R.Hoare.)

Let P, P1, P2, ... stand for predicates stating a relation between values of variables. Let S, S1, S2, ... stand for pieces of program text, in general affecting values of variables, i.e. changing the current state. Let B, B1, B2, ... stand for either predicates stating a relation between values of variables or for pieces of program text evaluating such a predicate, i.e. delivering one of the values true of false without further affecting values of variables, i.e. without changing the current state.

Then $P1\{S\}P2$

means: "the truth of P1 immediately prior to the execution of S implies the truth of P2 immediately after that execution of S". In terms of this formalism we write down the following theorems. (Some readers would prefer to call some of them rather "axioms", but at present I don't particularly care about that difference.)

Theorem 1:

Given: P1{S1}P2
P2{S2}P3

Conclusion: P1{S1; S2}P3

(This theorem gives the semantic consequences of the semicolon as connective.)

Theorem 2:

Given: B{S}nonB

Conclusion: true{if B ~~then~~ do S}nonB

(Here "true" is the condition which is satisfied per definition, i.e. the conclusion that after the execution of the conditional statement "nonB" will hold, is independent of any assumptions about the initial state.)

Theorem 3:

Given: (P and B){S} P

Conclusion: P {if B do S} P

Theorem 4:

Given: (P1 and B){S1} P2
(P1 and non B){S2} P2

Conclusion: P1 {if B then S1 else S2} P2

Theorem 5:

Given: (P and B){S} P

Conclusion: P {while B do S}(P and non B)

Remark: This theorem only applies to the case that the repetition terminates, otherwise it is void.

Theorem 5 is one of the most useful theorems when dealing with loops. The appropriate reasoning mechanism to deal in detail with loops is mathematical induction, but often the use of Theorem 5 -which itself can only be proved by mathematical induction- avoids a direct use of mathematical induction.

We used Theorem 5 when proving the correctness of program 1. Here was

P: $a > 0$ and $b > 0$ and GCD(a, b) = GCD(A, B) and

B: $a \neq b$.

We draw attention to the fact that we could not show "P {S} P" but only "(P and B){S} P": for a and b to remain positive it was necessary to know that initially they were different. (How is this with program 2?) We also draw attention to the fact that after termination when we wished to show that a = GCD(A, B) we did not only use the mild conclusion "P {while B do S} P" but the strong conclusion "P {while B do S}(P and non B)": we needed the knowledge "a = b" in order to justify the application of 3.2.

With respect to termintation one often uses a somewhat stronger theorem, the formulation of which falls outside the strict Hoare formalism:

Theorem 6:

Given: (P and B){S} P

Conclusion: in P {while B do S} will (P and B) hold immediately after each execution of the repeatable statement which is not the the last one.

This theorem usually plays a role when deriving a contradiction from the assumption that the loop will not terminate.

* * *

There is an alternative form of repetition control which might be represented by "repeat S until B"

(other authors write "do S until B"); it is semantically equivalent to

"S; while non B do S" .

(Instead of describing its semantics in terms of the other repetitive connective, we could also have given a recursive definition in terms of itself, viz.

"S; if non B do repeat S until B" .)

The differences with the while-clause are

1) the termination condition has been inverted

2) the repeatable statement is executed at least once.

Sometimes the repeat-clause comes in really handy and gains the text in clarity when it is used. It should be used with great precaution, a precaution which is shown by the pertinent

Theorem 7:

Given: P1 {S} P2

(P2 and non B){S} P2

Conclusion: P1 {repeat S until B}(P2 and B)

Remark: This theorem only applies to the case that the repetition terminates, otherwise it is void.

The greater complexity of the assumptions about B and S which have to be verfied, reflect the additional care required in the use of the repeat-clause.

Exercise. We ~~k~~now give three tentative alternative programs (3, 4 and 5) for the computation of GCD(A, B). Discover which of them are correct and which are faulty by trying to prove their correctness. If the program is incorrect, construct an A,B-pair for which it fails.

program 3:

```
begin integer a, b, gcd;
      a:= A; b:= B;
      repeat if a > b then a:= a - b
                      else b:= b - a
      until a = b;
      gcd:= a;
      print(A); print(B); print(gcd)
end
```

program 4:

```
begin integer a, b, gcd;
      a:= A; b:= B;
      repeat while a > b do a:= a - b;
             while b > a do b:= b - a
      until a = b;
      gcd:= a;
      print(A); print(B); print(gcd)
end
```

program 5:

```
begin integer a, b, gcd, x;
      a:= A; b:= B;
      while a ≠ b do
      begin if a < b do begin x:= a; a:= b; b:= x end;
            repeat a:= a - b until a ≤ b
      end;
      gcd:= a;
      print(A); print(B); print(gcd)
end
```

(Note. If any of the above programs is correct, its occurrence in this exercise is not to be interpreted as a recommendation of its style!)

Exercise. Prove that the following program will print in addition to the greatest common divisor of A and B the smallest common multiple of A and B (being defined as their product~~m~~ divided by the greatest common divisor).

```
begin integer a, b, c, d, gcd, scm;
      a:= A; b:= B; c:= B; d:= 0;
      while a ≠ b do
      begin while a > b do begin a:= a - b; d:= d + c end;
            while b > a do begin b:= b - a; c:= c + d end
      end;
      gcd:= a; scm:= c + d;
      print(A); print(B); print(gcd); print(scm)
end
```

Hint. Follow the value of the expression $a * c + b * d$. (Courtesy Software Sciences Holdings Limited, London)

*   *   *

For a meaningful loop controlled by a while-clause we can make the following further observations. Let us consider the loop "while B do S".

Our first observation is that the execution of the repeatable statement must change the state. Suppose that it does not. If then the repeatable statement is executed once, the predicate B (that was true prior to its execution) will still be true after its execution so that the statement S will be executed still another time; then the argument can be repeated and the net result is that the repetition will not terminate. The only termination is when B is false to begin with: then the statement S will be executed zero times. This we don't call "a meaningful loop".

So, execution of statement S changes the state: let us call "s" that part of the total state that may be changed by the execution of statement S. Here "s" can be regarded as the aggregate of variables whose values can be affected by the execution of statement S. Then our seond observation is that (one or more variables of) s must be involved in the predicate B. If not, a single execution of the repeatable statement would imply infinitely many (by the same argument as used in our previous observation). Treating "s" as a generalized variable, we can express this explicitly by rewriting the loop, using two functions of the generalized argument s:

"<u>while</u> B(s) <u>do</u> s:= S(s)" .

This is the basic form of <u>every</u> repetition which is controlled by a while-clause. Reading it, it is obvious that its behaviour is undefined when the initial value of s is completely undefined. In terms of programming this leads to our third observation:

> Every repetition controlled by a while-clause requires a proper initialization of the variables involved.

(Although obvious, this rule is mentioned explicitly, because I have seen many programmers forgetting to initialize properly. The rule is so rigid that such an omission is not a mistake but a blunder.)

Let the initial state be denoted by s_0 and let the state immediately after the i-th execution of the repeatab:e statement be denoted by s_i; let the loop terminate after the k-th execution of the repeatable statement. Then the following theorems are true.

Theorem 8:

Given s_0, the remaining successive values of s are given by the recurrence relation
$$s_i = S(s_{i-1}) \qquad \text{for } 0 < i \leq k.$$

Theorem 9:

In the sequence $s_0, s_1, \ldots, s_k$ no two values (with different subscripts) are equal.

Theorem 10:

$B(s_i)$ for all i satisfying $0 \leq i < k$ and

$\underline{\text{non}}\ B(s_k)$.

Theorem 8 is fairly obvious. It is mentioned, however, because it shows that a repetition is an adequate tool for the evaluation of a recurrence relation.

Theorem 9 is mentioned because it is so obviously correct (although I see no use for it).

Theorem 10 -also called "The Linear Search Theorem"-which is also fairly obvious, is a very important one, comparable in importance to Theorem 5. It does not only assert that after termination the state s_k will be such that *non* B is valid, it asserts that in all previous states s_i with $i < k$ (if any) B did hold. This theorem is used in many searching algorithms looking for the *smallest* number for which a predicate B is false: by inspecting the numbers in the order of increasing magnitude, the *smallest* becomes the *first* found. Theorem 10 tells us that in that case a loop controlled by a while-clause is a proper vehicle. (It is not restricted to looking for a smallest number, it can be applied to any (somehow) ordered set of values of some sort.)

Remark. We have called the last three theorems "obvious". This is not meant to imply that giving a reasonably formal proof for them is trivial. I did it, but the resulting proofs were surprisingly cumbersome and boring.

Finally. in Theorems 2, 3 and 5 we have *always* assumptions including the truth of B prior to the execution of S. Clearly we are not interested in the net effect of the execution of S after an initial state in which B is false. As a matter of fact, with such initial states the net effect of an S will often be undefined. (A similar remark can be made regarding Theorem 4.) In other words: our statements S regarded as operators are often not defined on the domain of all possible states, they are "partial operators" and the predicates occurring in the clauses ensure that they will not be evoked inappropriately. This observation should temper our hope of increasing computation speed by introducing more parallellism.

Programs corresponding to recurrence relations.

Theorem 8 (EWD287 - 39) mentions successive states connected by a recurrence relation. The meaning of this theorem is twofold: it can be used to prove asserions about a given program, but also -and this, I think, is more important- it suggests us, when faced with the task of making a program, the use of a while-clause in the case of a problem that in its mathematical formulation presents itself as the evaluation of a recurrence relation. We are going to illustrate this by a number of examples.

Consider the sequence of pairs a_i, c_i given by

for i = 0 $\qquad a_0 = 1$ (1)

$\qquad c_0 = 1 - b$, with $0 < b < 2$ (i.e. $abs(c_0) < 1$)

for i > 0 $\qquad a_i = (1 + c_{i-1})* a_{i-1}$ (2)

$\qquad c_i = c_{i-1}^2$.

Then $\qquad \lim_{i \to \infty} a_i = 1/b$

Exercise. Prove the last formula. (This has nothing to do with programming, it is secondary school algebra. The clue of the proof is to be found in the relation

$$\frac{1}{1 - c_{i-1}} = \frac{1 + c_{i-1}}{1 - c_i}$$

.)

It is requested to use this recurrence relation to approximate the value of 1/b; obviously we cannot compute infinitely many elements of the sequence a_0, a_1, a_2, ... but we can accept a_k as a sufficiently close (how close?) approximation of 1/b when c_k is less than a given, small, positive tolerance named "eps". (This example is of historical interest; it is taken from the subroutine library for EDSAC 1, the world's first stored program controlled automatic computer. The order code of this computer did not comprise a divide instruction and one of the methods used with this computer to compute quotients was based on the above recurrence relation.)

Theorem 8 talks about "a prt, s, of the state space" and the loop

```
while B(s) do s:= S(s)
```

and asserts that after the initial state s_0 the states s_i after the i-th execution of the repeatable statement will satisfy

$$s_i = S(s_{i-1}) \qquad (3)$$

Our recurrence relations (2) are exactly of the form (3) if we identify the state s_i with the value pair a_i, c_i. That is, we have to introduce, to span the part s of the state space, two variables, for the purpose of this discussion called A and C, and we shall denote their values after the i-th execution of the repeatable statement by A_i and C_i respectively. We associate the state s_i (as given by the values A_i and C_i) with the value pair a_i, c_i by the relations

$$\begin{aligned} A_i &= a_i \\ C_i &= c_i \quad . \end{aligned} \qquad (4)$$

(Mind you: in the left hand sides the subscript "i" means "the value of the variable after the i-th execution of the repeatable statement", in the right hand sides the subscript "i" refers to the recurrent sequences as given by (1) and (2). It would have been usual to call the two variables "a" and "c" instead of "A" and "C", i.e. not to distinguish between the quantities defined in the mathematical formulation on the one hand and the associated variables in the program on the other hand. As this association is the very subject of this discussion, it would have been fatal not to distinguish between them.)

Within the scope of a declaration "real A, C" it is now a straightforward task to write the piece of program:

```
"A:= 1; C:= 1 - b;
 while ~~C ≥ eps do~~ abs(C) ≥ eps do
 begin A:=(1 + C)* A;
       C:= C * C
 end"   .
```

$$\boxed{\frac{A}{1-C} = \frac{1}{b}}$$

The first line has to create the proper state s_0 and does so in accordance with (4) and (1), the repeatable statement has the form, symbolically denoted by "s:= S(s)" -see the Note below- in accordance with (4) and (2), and the condition guarantees that after termination

$$(A_k =) \quad A = a_k$$

will hold with the proper value of k.

Exercise. Prove that the loop terminates.

Note. The symbolic assignment "s:= S(s)" has the form of two assignments

$$A{:=}\ (1 + C)* A;$$
$$C{:=}\ C * C; \quad .$$

With the initial condition $A = a_{i-1}$, $C = c_{i-1}$ the first assignment is equivalent to

$$A{:=}\ (1 + c_{i-1})* a_{i-1}$$

and after the first, but before the second assignment we have -on account of (2)-

$$A = a_i, \ C = c_{i-1} \quad .$$

Only after the second assignment we have the complete pair

$$A = a_i, \ C = c_i \quad .$$

Thanks to the explicit occurrence of the subscripts the order of the two relations composing (2) is immaterial, this in contrast to the two assignment statements composing the repeatable statement, whose order is vital!

Exercise. In the same EDSAC subroutine library the next scheme is used. Consider the sequence of pairs a_i, c_i , given by

for i = 0 $\qquad a_0 = b$

$\qquad\qquad\qquad c_0 = 1 - b$ with $0 < b < 2$ (i.e. $abs(c_0) < 1$)

for i > 0 $\qquad a_i = (1 + .5 * c_{i-1})* a_{i-1}$

$\qquad\qquad\qquad c_i = c^2_{i-1} *(.75 + .25 * c_{i-1}) \quad .$

$$\boxed{\frac{A^2}{1 - C} = b}$$

Then $\qquad \lim\limits_{i \to \infty} a_i = b^{.5}$

Prove the last formula and make a program using it for the approximation of the square root. The clue of the proof is to be found in the relation

$$(1 - c_{i-1})^{-.5} = (1 + .5 * c_{i-1}) * (1 - c_i)^{-.5} \quad .$$

Prove also the termintation of the repetition in the program made.

Exercise. In the same EDSAC subroutine library the next scheme is used. Consider the sequence of triples inc_i, s_i, x_i given by for i = 0 $\quad inc_0 = \log 2$

$\qquad s_0 = 0$

$\qquad x_i = arg \qquad$ (with $1 \leq arg < 2$)

for $i > 0$

for $x_{i-1}^2 < 2 \quad inc_i = .5 * inc_{i-1}$

$\qquad s_i = s_{i-1}$

$\qquad x_i = x_{i-1}^2$

for $x_{i-1}^2 \geq 2 \quad inc_i = .5 * inc_{i-1}$

$\qquad s_i = s_{i-1} + .5 * inc_{i-1}$

$\qquad x_i = .5 * x_{i-1}^2 \qquad .$

Then $\quad \lim_{i \to \infty} s_i = \log(arg)$

Prove this relation and make a program using it to approximate the logarithm of a value arg in the interval stated. (In this program "log 2" may be regarded as a known constant which, in fact, determines the base of the logarithm.)

The clue of the proof can be found in the invariance of the relation

$$\log(arg) = s_i + inc_i * \log(x_i) / \log 2 \quad .$$

Our next example is very simple; it is so traditional that we could call it standard. (No self-respecting programming course omits it, it is often the very first example of a loop; Peter Naur uses it in his article "Proof of algorithms by general snapshots", 1966, BIT, 6, pp 310-316.)

Given a sequence of values

$$a[1],\ a[2],\ a[3],\ \ldots,\ a[N] \qquad \text{(with } N \geq 1\text{)}$$

and a variable called "max". Make a piece of program assigning to the variable named "max" the maximum value occurring in the sequence. (As $N \geq 1$, the sequence is not empty and therefore the task makes sense; it is not required that any two values in the sequence differ from each other, the maximum value sought may occur more than once in the sequence.) If he welcomes the experience the reader is invited to try to make this piece of program himself before reading on.

How do we define the maximum value occurring in a sequence of length N for general $N \geq 1$? If we call "maximum_k" the maximum value occurring among the first k elements $a[1], \ldots , a[k]$

1) the ansewer sought is maximum_N

2) the values maximum_k are given

for $k = 1$ by the base: $\text{maximum}_1 = a[1]$ (5)
-appealing to the knowledge that the maximum element in a sequence of length 1 must be the only element occurring in the sequence

for $k > 1$ by the recurrence relation: $\text{maximum}_k = \text{MAX}(\text{maximum}_{k-1}, a[k])$ (6)
-assuming the knowledge of the function MAX of two arguments.

The recurrence relation (6) presents us with an additional difficulty because it is <u>not</u> of the form

$$s_i = S(s_{i-1})$$

because -via "a[k]"- the value k occurs at the right hand side not exclusively in the subscript "k-1". To overcome this we use a trick that might be called a method. If we call n_k the k-th natural number, then $n_k = k$; the numbers n_k the obvious recurrence relation $n_k = 1 + n_{k-1}$. We can now rewrite the definition for the sequence of values maximum_k in the form of a definition for the pairs $(n_k, \text{maximum}_k)$:

for $k = 1$ $n_1 = 1$ (7)
$\text{maximum}_1 = a[1]$

for $k > 1$ $n_k = 1 + n_{k-1}$ (8)
$\text{maximum}_k = \text{MAX}(\text{maximum}_{k-1}, a[1 + n_{k-1}])$

and now the recurrence relations are of the form $s_i = S(s_{i-1})$, the only -trivial- difference being that in Theorem 8 we started with $i = 0$ and here with $k = 1$. The trick we called a method shows that we need a second (integer) variable; call it "m". Our state s_i will associate (with $k = i + 1$)

$$\text{max}_i = \text{maximum}_k$$

$$m_i = n_k \quad .$$

The piece of program now becomes:

```
max:= a[1]; m:= 1;
while m < N do
begin m:= m + 1;
      max:= MAX(max, a[m])
end  .
```

Again the order of the two assignment statements is essential.

We have given the above piece of reasoning and the explicit reference to the recurrence relation of Theorem 8 because it shows a mechanism leading the the conclusion that the part s of the state space on which the repetition operates needs to comprise an additional variable. Even a moderately trained programmer draws this conclusion "intuitively" and from now onwards I shall assume my reader equipped with that intuition. Then -and only then!- there is a much shorter path of reasoning that leads to the program we found. It does not consider -"statically" so to speak- the sequence of values s_0, s_1,... in the sense that it bothers about the values of the subscript i in s_i. It appeals directly to Theorems 5 and 6 and works in terms of assertions valid (before or after) any execution of the repeatable statement. The price to be paid for this is the duty to prove termination separately.

Given the base

$k = 1 \qquad \text{maximum}_1 = a[1]$

and the step

$1 < k \leq N \qquad \text{maximum}_k = \text{MAX}(\text{maximum}_{k-1}, a[k])$

the programmer "intuitively" introduces two variables which he calls "maximum" and "k" for short and the relation to be kept invariant is

P: $\qquad 1 \leq k \leq N$ <u>and</u> $\text{maximum} = \text{maximum}_k$.

(Here any use of "maximum" and "k" stands for the current value of the variable thus named, while "maximum_k" stands for the value as given by the recurrence relation. This double use of the same names is tricky but programmers do it. I too.)

The program then consists of two parts: establishing relation P in accordance with the base and repeatedly increasing k under invariance of relation P, i.e. in accordance with the step.

The initialization

"maximum:= a[1]; k:=1"

establishes P (with k = 1), the repetition

```
"while k < N do
 begin k:= k + 1;
       maximum:= MAX(maximum, a[k])
 end"
```

causes the repeatable statement to be executed under the combined relation "$B\ \underline{\text{and}}\ P$" i.e.

$$k < N\ \underline{\text{and}}\ 1 \leq k \leq N\ \underline{\text{and}}\ \text{maximum} = \text{maximum}_k$$

which reduces to

$$1 \leq k < N\ \underline{\text{and}}\ \text{maximum} = \text{maximum}_k \qquad (9)$$

In order to show that the execution of the repeatable statement under the initial condition (9) leaves relation P valid it is desirable to distinguish between the values before and after its execution; now it would be confusing to do so with subscripts (why?), therefore we distinguish the values after execution by primes.

Initially we have relation (9); after the assignment $k := k + 1$ we have the relation $k' = k + 1$ and from the first part of (9), i.e. $1 \leq k < N$, follows $2 \leq k' \leq N$, which implies

$$1 \leq k' \leq N \quad . \qquad (10)$$

The second assignment now becomes effectively $\text{maximum} := \text{MAX}(\text{maximum}_k, a[k'])$, resulting in the relation

$$\text{maximum}' = \text{maximum}_{k'} \quad .$$

Relations (10) and (11) combine to a replica of P, but now for the primed quantities.

Termination follows from the fact that each execution of the repeatable statement involves an effective increase of the integer values variable k. After termination we have, according to Theorem 5, $P\ \underline{\text{and}}\ \underline{\text{non}}\ B$, i.e.

$$1 \leq k \leq N\ \underline{\text{and}}\ \text{maximum} = \text{maximum}_k\ \underline{\text{and}}\ \underline{\text{non}}\ k < N \quad ;$$

from the first and the last term we conclude $k = N$ and then from the middle part

$$\text{maximum} = \text{maximum}_k$$

which concludes the proof.

Exercise. Make a program effectively assigning "prod:= X * Y" with integer X and Y, satisfying $X \geq 0$, $Y \geq 0$

a) using only addition and subtraction

b) using in addition the boolean function "odd(x)", doubling and halving of a number. (The so-called Egyptian multiplication.)

Exercise. Make a program effectively assigning "rem:= REM(X,Y)" with integer X and Y, $X \geq 0$, $Y > 0$, where the function REM(X,Y) is the remainder after division of X by Y

a) using only addition and subtraction

b) using in addition doubling and halving of a number. Modify both programs such thatm in addition" "quot:= QUOT(X,Y)" will effectively take place. (The so-called Chinese division.)

We conclude this section by an example of the (standard) circumstance in which a recurrence relation should not be translated blindly into a loop. Given two sequences of values

$$x[1], x[2], \ldots , x[N] \quad \text{and}$$
$$y[1], y[2], \ldots , y[N] \qquad \text{with } N \geq 0 \;;$$

make a program assigning to the voolean variable named "eq" the value true if $x[i] = y[i]$ for all i satisfying $1 \leq i \leq N$ and the value false if in that range a value of i exists such that $x[i] \neq y[i]$. (The sequences may be empty, in that case "eq" should get the value true.)

How do we define equality of sequences of length N for general N? Again by means of a recurrence relation.

Let eq_i mean "no difference has been observed among the first i pairs"; then the sequence of values eq_i is given by

for $i = 0$ $\qquad eq_0 = \underline{\text{true}}$

for $i > 0$ $\qquad eq_i = eq_{i-1} \;\underline{\text{and}}\; x[i] = y[i]$

The net effect of the program to be made should be

$$eq:= eq_N \qquad .$$

A blind translation into initialization followed by repetition would lead to

```
eq:= true; i:= 0;
while i < N do begin i:= i + 1; eq:= (eq and x[i] = y[i]) end   .
```

Although the above program is correct, the following program, besides being equally correct, is on the average much more efficient:

```
eq:= true; i:= 0;
while i < N and eq do begin i:= i + 1; eq:= (x[i] = y[i]) end
```

because it terminates the repetition as soon as a difference has been found.

Exercise. Prove the correctness of the second program.

A first example of step-wise program composition.

The little examples dealt with so far are not representative for the programs that have to be made: they are several orders of magnitude too small. A trained programmer "sees" them at a glance, he can think about them without pencil and paper. They are of the size of a paragraph, while we have to deal with programs of the size of pages, chapters or books and eventually -to quote A.Perlis- with programs that no longer fit into a single programmer's skull! The composition of such very large programs falls outside the scope of this little monograph, but the very least thing we can do is to show the reader how to organize his thoughts when composing, say, page-size programs. If in the following the reader thinks that I am too careful, he should bear the chapter-size programs in mind. (If he is still inconvinced he should study a one-page program by a messy programmer: he will then discover that even a single page has room enough for a disgusting and intellectually unhealthy amount of unmastered complexity!)

Here we go. Consider sequences composed of 1's, 2's and 3's which contain only pairs of different adjoining non-empty subsequences. Examples of good sequences are

1
12
12312
3212321 .

Examples of bad sequences are

11
12323131
32132132132 .

In all probability the list of good sequences is infinite. I tell you that there is as least one good sequence of length 100 (i.e. consisting of 100 digits) and it is asked to make a program generating in alphabetic order the beginning of this list up tp and including the first sequence of length 100. (Alphabetic order under the convention that the 1 precedes the 2 and the 2 precedes the 3; the criterion can be translated into a numerical one by putting a decimal point in front of the sequence and then interpreting the sequence as a decimal fraction. With that convention alphabetic order is the order of increasing magnitude.)

I have used this example extensively in oral examinations. After some initial skirmishes, most students discovered for themselves

1) that a wrong sequence could never be made into a good one by extending it, i.e. all good sequences are either a one-digit sequence of a one-digit extension of a good sequence

2) if sequence B is a good one-digit extension of sequence A, sequence A precedes sequence B in the alphabetic order, i.e. a good sequence is followed by all its possible extensions

3) the alphabetic order requires that the good sequence A will first be followed by the extensions starting with a 1 (if any), then by those starting with a 2 (if any) and then by those starting with a 3 (if any).

These observations lead to the following rule:

a good sequence should be printed and be extended with a 1 as a next trial sequence; from a bad sequence terminal 3's (if any) should be removed and the final digit (now ≠ 3) should be increased by 1, giving the next trial sequence.

The beginning of the list to be generated is

1

12

121

1213

12131

121312

1213121

1213123

....

by searching the following list of trial sequences (omitting the ones marked by †)

1

† 11

12

121

† 1211

† 1212

1213

12131

† 121311

121312

```
      1213121
†     12131211
†     12131212
†     12131213
†     1213122
      1213123
      .......
```

Many of them suggested a program of the following structure.

program 1:

```
SET SEQUENCE TO ONE AND LENGTH TO ONE;
repeat if GOOD then begin PRINT; EXTEND WITH ONE end
                else ADJUST
until length > 100
```

in which the primitive "EXTEND WITH ONE" extends the given sequence with a 1 and the primitive "ADJUST" increases the last digit by 1 after removal of terminal 3's if any. (For the operation "ADJUST" to be defined it is necessary that after removal of terminal 3's the remaining sequence won't be empty; this follows from the fact that the list to be produced is known to contain a sequence of length = 100)

To a program along the lines sketched, a number of objections can be made. One objection is that at the beginning of the execution of the repeatable statement its length will be ≤ 100 and furthermore we know that the operation "ADJUST" will never increase its length; nevertheless each adjustment is followed in time by a test on the length and the first objection is therefore that these tests are superfluous. A more severe objection is to be found in the tortuous reasoning required to establish that the end condition is OK. Instead of stopping when for the first time a solution of length 100 has been printed, it will stop when the first trial sequence of length > 100 has been generated. It is clear that the above program will never produce a solution larger than 100 because such a long trial sequence will never be subjected to the test "GOOD". To show, however, that it will stop after the production of the <u>first</u> solution of length = 100 is much harder.

A much nicer program is based upon the observation that we can regard the empty sequence as a virtual solution which does not need to be printed.

program 2:

```
      SET SEQUENCE EMPTY AND LENGTH TO ZERO;
      repeat EXTEND WITH ONE;
             while non GOOD do ADJUST;
             PRINT
      until length = 100     .
```

The objections raised are no longer valid. The true reason, however, why the above program is so much more attractive, is to be found in the observation that we can mentally combine the first two statements of the repeatable statement into a single one. The above version is a refinement of the more abstract

program 3:

```
      SET SEQUENCE EMPTY AND LENGTH TO ZERO;
      repeat TRANSFORM SEQUENCE INTO NEXT SOLUTION;
             PRINT
      until length = 100.
```

(Note. In programs 1, 2 and 3 the (outer) repetition could also have been controlled by a while-clause.)

This is much more preferable. Observe that here, in program 3, we have a level of description from which the trial sequences have disappeared! It is a level of description which can be understood in terms of solutions only. By distinguishing, i.e. by mentally isolating the operator "TRANSFORM SEQUENCE INTO NEXT SOLUTION" and postponing its refinement, we have separated the task of formulating the correct criterion for termination from how the transition from one solution to the next will be performed via a number of trial sequences which may be rejected. Remembering the limited size of the programmer's skull, this separation is a vital achievement, as it enables us to deal with one thing at a time.

To show that all this is not just idle playing with words we shall proceed from program 3 as our starting point, refining from there onwards. By way of surprise we shall arrive at a refinement differing from program 2, again without essentially changing the algorithm. (Although I had used this example extensively in examinations, the next version only occurred to me when writing this very text! This is just to show, that such abstract programs are vital stepping stones in our process of constructive reasoning!)

To find this refinement we do a step backwards, asking ourselves what enables us the transition from program 1 to program 2. It was the introduction of the empty sequence as "virtual solution". In program 1, the first solution was given, the others were generated, in program 2 and 3, all solutions to be printed are generated by the same operator "TRANSFORM SEQUENCE INTO NEXT SOLUTION".

When refining this operator the trial sequences have to be generated and in program 2 we find that the criterion "GOOD" has to be applied to trial sequences generated in two different ways, either by "EXTEND WITH ONE" or by "ADJUST". Can we clean up our refinement by insisting that all trial sequences to be tested are generated by the same operator? Yes we can by slightly changing the extension operator and slightly generalizing the operator "ADJUST", as in the following refinement.

```
TRANSFORM SEQUENCE INTO NEXT SOLUTION:
      EXTEND WITH ZERO;
      repeat ADJUST until GOOD  .
```

Here "GOOD" stands for a rather complicated function; an alternative form uses the boolean variable "good" and leaves us with the task of refining the operator "SET GOOD".

```
TRANSFORM SEQUENCE INTO NEXT SOLUTION:
boolean good;
      EXTEND WITH ZERO;
      repeat ADJUST; SET GOOD until good     .
```

(Note. In the above refinement the repetition is not to be controlled by a while-clause, Why?)

Now the time has come to make a decision on the representation of "sequence". It has a property "length", now satisfying the inequalities $0 \leq \text{length} \leq 100$ and is an ordered sequence of length digits. The appropriate vehicle for representing this sequence is (part of) a linear array of integer variables. We suggest to declare an <u>integer</u> <u>array</u> d[1:100] and that at any moment the sequence will be represented by

$$d[1],\ d[2],\ \ldots\ ,\ d[\text{length}] \quad .$$

We draw attention to the fact that this is a well-motivated decision. An alternative representation would have been

```
d[101 - length], d[102 - length], ... , d[100]
```

but with the latter convention all operations changing the length of the sequence would imply moving the values upwards or downwards, whereas with the the suggested representation, the values being kept can "stay where they are". When the chosen convention is made to apply to all sequences manipulated (i.e. to solutions as well as to trial sequences) the following four refinements are fairly obvious. (As far as they are concerned, the chosen representation is certainly adequate.)

```
SET SEQUENCE EMPTY AND LENGTH TO ZERO:
    length:= 0

EXTEND WITH ZERO:
    length:= length + 1; d[length]:= 0

ADJUST:
    while d[length] = 3 do length:= length - 1;
    d[length]:= d[length] + 1

PRINT:
    i:= 0; repeat i:= i + 1; printdigit(d[i]) until i = length; newline
```

where we have assumed the availability of the primitives "printdigit" and "newline" for the transition to the beginning of the next line of output. The only refinement which still can cause headaches is the operator "SET GOOD".

To investigate an arbitrary sequence is indeed a gruesome task, but it becomes much easier if we exploit the circumstance that the only sequences subjected to the test are trial sequences and each trial sequence is a one-digit extension of an (earlier) good sequence. As a result it can only violate the condition if its terminal element is included in one of the subsequences, i.e. it has to be rejected as bad if there exists an m (satisfying $0 < 2 * m \leq$ length) such that the pair of adjoining "msequences"

```
d[length - 2*m + 1] ... d[length - m]     and
d[length - m + 1] ... d[length]
```

are equal. Assuming the availability of the operator needed to compare the msequences of this pair (for arbitrary, given m) our first refinement of "SET GOOD" is

```
SET GOOD:
      integer m;
      good:= true; m:= 1;
      while 2 * m ≤ length and good do
      begin GIVE GOOD THE MEANING THAT THE MSEQUENCES DIFFER;
            m:= m + 1
      end
```

or (probably) better

```
SET GOOD:
      integer m, mbound;
      good:= true; m:= 1; mbound:= length div 2;
      while m ≤ mbound and good do
      begin GIVE GOOD THE MEANING THAT THE MSEQUENCES DIFFER;
            m:= m + 1
      end
```

Here the operator div is the integer divide, rounding the quotient to the nearest integer towards zero. The double condition for continuing the repetition expresses that the investigation can be stopped as soon as an equal pair has been found, as this is sufficient to establish its being bad. We have seen this construction at the end of the previous section, see EWD287 - 49.

Question. An alternative form would have been

```
      integer m;
      good:= true; m:= length div 2;
      while m > 0 and good do
      begin GIVE GOOD THE MEANING THAT THE MSEQUENCES DIFFER;
            m:= m - 1
      end  .
```

Why do we propose to do the investigation in the order of increasing m?

Finally we refine the comparison of two msequences

```
GIVE GOOD THE MEANING THAT THE MSEQUENCES DIFFER:
      integer firstend, k;
      firstend:= length - m; k:= 0;
      repeat good:= (d[firstend - k] ≠ d[length - k]);
             k:= k + 1
      until k = m or good
```

again (see EWD287 - 49) expressing that the comparison of two msequences can be terminated as soon as their difference somewhere has been established.

Collecting the declarations and inserting the refinements -keeping their names as labels for explicative purposes. we arrive at the complete program as shown on EWD287 - 58. The succesive levels of detail have been indicated by systematic use of indentation.

Exercise.

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, restricting ourselves to sequences starting with five zeros. C.Ligtmans has shown that any such solution must end with four zeros. His argument is as follows. Each solution must start with 000001... -because the pattern 00000 may only occur once. Somewhere in the sequence the pattern 10000 must occur once; as this pattern can only be followed by a 0 or a 1, its "following"quintuple must be either 00000 or 00001, presented already by the first two quintuples. As a result the pattern 10000 cannot occur in the interior of the linear sequence and therefore it must be the rightmost pattern. From Ligtmans' observation it follows that we can close the ring, making the last four zeros overlap with the four initial ones. The 32 different patterns are then arranged in a cycle of 32 positions. The patterns have to be generated in alphabetic order.

Discussion and some hints.

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 alphabetic 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 alphabetic order will ensure that the solutions, when found, will be found in alphabetic order as well.

The program to be made could be regarded as a derivation from the ridiculous one sketched above, viz. by the introduction of some significant short-cuts. At present we shall not stress this relation any further.

```
begin integer array d[1:100]; boolean good; integer length, i, m, mbound, k, firstend;
    SET SEQUENCE EMPTY AND LENGTH TO ZERO:
        length:= 0;
    repeat TRANSFORM SEQUENCE INTO NEXT SOLUTION:
            EXTEND WITH ZERO:
                length:= length + 1; d[length]:= 0;
            repeat ADJUST:
                    while d[length] = 3 do length:= length - 1;
                    d[length]:= d[length] + 1;
                 SET GOOD:
                    good:= true; m:= 1; mbound:= length div 2;
                    while m ≤ mbound and good do
                    begin GIVE GOOD THE MEANING THAT THE MSEQUENCES DIFFER:
                            firstend:= length - m; k:= 0;
                            repeat good:= (d[firstend - k] ≠ d[length - k]);
                                k:= k + 1
                            until k = m or good;
                        m:= m + 1
                    end
            until good;
         PRINT:
            i:= 0; repeat i:= i + 1; printdigit(d[i]) until i = length; newline
    until length = 100
end
```

Instead of generating all 36-digit sequences and selecting from this set, we aim at generating only a much smaller set which is guaranteed to contain all solutions. 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 leading zeros 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 transform an acceptable sequence into the next acceptable sequence
5) solutions are all acceptable sequences (excluding the virtual one) satisfying the further condition length = 32.
6) no extension of a sequence that is not acceptable will be acceptable.

It is this latter property which makes this problem mathematicaaly speaking very much similar to the previous one.

Hint. The test for acceptability can be speeded up considerably by tabulating which quintuples are to be found in the sequence.

Remark. This problem is difficult and it will take you hours to produce a beautiful program. But these hours will be extremely well-spent.

The shortest spanning subtree of a graph.

I have chosen the following example for a variety of reasons. Firstly, although the final program is rather short, the solution is far from trivial. Secondly, our true subject matter is rather "structure" than straightforward numerical material and as a result the decisions taken to represent the information (using numerical values) are more manifest. Finally it presents us with a type of decisions of strategy which are typical.

Two points can be connected by one point-to-point connection, three points can be connected with each other by two point-to-point connections, in general N points can be fully interconnected by N-1 point-to-point connections. Such a set of interconnections is called a "tree", the point-to-point connections that constitute the tree are called "its branches". Cayley has been the first to prove that the number of possible trees between N points equals N^{N-2}.

We now assume that for each possible branch the length has been given. Defining the length of a tree as the sum of the length of its branches, we can ask for the shortest tree between those N points. (For the time being we assume that the given lengths are such that the shortest tree is unique. From our analysis it will follow that no two branches of equal length is a sufficient condition for this assumption.)

Note. The points don't need to lie in a Euclidean plane, the given distances don't need to satisfy the triangle inequality.

An apparently straightforward solution would generate all trees between the N points, compute their lengths and select the shortest one, but Cayley's theorem shows that that would become very expensive as N increases. The following theorem enables us to find the shortest tree with considerably less work. Given a subtree of the shortest tree, then the shortest branch that can be found between one of the points touched by this tree and one of the points not touched by this subtree will be part of the shortest tree between all N points.

This theorem is easily proved. Colour the branches of the subtree and all points connected by it red; colour all the remaining points blue and colour all branches leading from a red point to a blue one violet. The theorem asserts that the shortest violet branch is part of the shortest tree as well. Call this shortest violet branch V and assume that it is not part of the shortest tree T; we shall then construct a tree T' which is shorter than T, thus arriving at a contradiction. Add to the tree T the violet branch V; in the resulting graph the violet branch V must be contained in a closed loop. As this violet branch connects a red point with a blue one, it is clear that, going around the loop we must find at least one other violet branch in this loop. Call this V' and remove V'. The resulting graph has again N-1 branches, it interconnects all N points (we have removed a branch from a loop) and therefore it is a tree connecting all N points. Call it T'. From $T' = T + V - V'$ follows:

$$\text{length}(T') = \text{length}(T) + \text{length}(V) - \text{length}(V') \quad .$$

As V was the shortest violet branch, we have

$$\text{length}(V) < \text{length}(V'),$$

so that

$$\text{length}(T') < \text{length}(T)$$

i.e. the tree T cannot have been the shortest one.

The above theorem tells us that a red subtree of the shortest tree T can be extended with a point and the branch leading to it, viz. the shortest violet branch and the blue point it leads to can be coloured red. As a result, if we can find a red subtree to start with, we can let it grow with a branch at a time. But it is very easy to start with a red subtree, viz. the red subtree consisting of a single point (any point will do) and no branches. Starting from that subtree we can let it grow to the sortest tree T in N-1 steps, each step adding a new red branch and a new red point to it. We can represent the framework of this algorithm as follows:

```
COLOUR ONE POINT RED AND THE REMAINING ONES BLUE;
while NUMBER OF RED POINTS < N do
begin SELECT SHORTEST VIOLET BRANCH;
      COLOUR IT AND ITS BLUE ENDPOINT RED
end     .
```

As it stands, the main task will be "SELECT SHORTEST VIOLET BRANCH", because the number of violet branches may be quite large, viz. k *(N - k) where k = NUMBER OF RED POINTS. If "SELECT SHORTEST VIOLET BRANCH" were an isolated operation, there is not much that could be done about it; in the above program, however, the operation has to be performed N-1 times in succession and the successive sets of violet branches are strongly related: they are the branches between red and blue points and each time only one point changes its colour. We would like to exploit this with the aim of reducing the set of branches from which each time the shortest branch should be selected: we are looking for a useful subset of the violet branches. We still don't know if such a really useful subset exists, but let us assume for the moment that it can be found and let us call it the "ultraviolet" ones. If such a set exists (each time) it is only helpful provided that we have a cheap way of constructing this subset and our only hope is to be found in the past history of the computation, for instance the set of ultraviolet branches used the preceding time. This suggest a program of the following structure:

```
COLOUR ONE POINT RED AND THE REMAINING ONES BLUE;
CONSTRUCT THE SET OF ULTRAVIOLET BRANCHES;
while NUMBER OF RED POINTS < N do
begin SELECT SHORTEST ULTRAVIOLET BRANCH;
      COLOUR IT AND ITS BLUE ENDPOINT RED;
      ADJUST THE SET OF ULTRAVIOLET BRANCHES
end
```

where the set of ultraviolet branches should be defined in such a way that

1) it is guaranteed to contain the shortest violet branch

2) the set of ultraviolet branches is in general much smaller than the set of the just violet ones

3) the operation "ADJUST THE SET OF ULTRAVIOLET BRANCHES" is cheap -otherwise the profit we are trying to gain is lost.

Can we find such a definition of the concept "ultraviolet"? Well, for lack of further knowledge I can only suggest that we try.

Considering that the set of violet branches leading from the k red points to the N-k blue ones has k *(N-k) members and observing criterion 1, two obvious possible subsets present themselves immediately:

1) Make for each red point the shortest violet branch ending in it ultraviolet. In this case the set of ultraviolet branches has k members.

2) Make for each blue point the shortest violet branch ending in it ultraviolet. In this case the set of ultraviolet branches has N-k members.

Our aim is to keep the ultraviolet subset small, but from their size we won't get a clue: with the first choice the sizes will run from 1 through N-1, with the second choice it will be the other way round. So, if there is any chance of deciding we must find it int the price of the operator "ADJUST THE SET OF ULTRAVIOLET BRANCHES".

Without trying the various adjustments of the ultraviolet sets, there is one observation which suggests a preference for the second choice. In the first choice k ultraviolet branches may lead from the red tree to the same blue point; then we know a priori that at most one of them will be coloured red, while with the second choice each blue point is connected in exactly one way to the red tree (the sum of the number of red and ultraviolet branches is then constantly = N-1) and it is possible that all ultraviolet branches at a certain moment will eventually be coloured red -in which case the adjustment operator was empty but for the removal of the one made red. Therefore, let us try the second choice for the criterion ultraviolet. (The initial value of this set are the N-1 branches leading from the one red point to the remaining N-1 blue ones, so that presents no problem.)

Consider now that we have a red subtree R and that from the corresponding set of ultraviolet branches (according to the second choice -I ~~am~~ shall no longer repeat~~ing~~ that qualification) the shortest branch leading to the blue point P and the blue point P have been coloured red. The number of ultraviolet branches has been decreased by 1 as it should. Are they the good ones? For each blue point they represent the shortest connection to the red tree R, they should represent the shortest possible connection to the new red tree R + P. But this is for each blue point B a simple comparison: if the branch PB is shorter than the ultraviolet branch connecting B to R the latter is to be replaced by the branch PB -its colour is washed away and PB is made ultraviolet instead- otherwise it is maintained, as the growth of the red tree R with the point P did not provide a shorter way of connecting B with the red tree. As a result the adjustment operator -which has to deal with N-k blue points- is in price a linear function of N and K (and not quadratic as k *(N-k)) and the introduction of this concept of ultraviolet is indeed giving us the gain we were hoping for.

Exercise. Convince yourself that the rejected choice of the concept "ultraviolet" is not so helpful.

Let us try to represent our algorithm in its current stage of refinement:

```
COLOUR ONE POINT RED AND THE REMAINING ONES BLUE;
CONSTRUCT THE SET OF ULTRAVIOLET BRANCHES;
while NUMBER OF RED POINTS < N do
begin SELECT SHORTEST ULTRAVIOLET BRANCH AND CALL ITS BLUE ENDPOINT P;
      COLOUR IT AND POINT P RED;
      ADJUST FOR EACH BLUE POINT B BY COMPARING WITH THE BRANCH BP
end   .
```

By now the time has come to bother about the representation of the information involved. We assume that N points being numbered from 1 through N, we assume the length of the branches being given by a two-dimensional array

real array distance[1:N, 1:N] ,

such that for $1 \leq i,j \leq N$

distance[i,j] = distance[j,i] = length of branch connecting the points i and j.

The answer asked is a tree of N-1 branches, each branch being identified by the numbers of its endpoints; the answer is an (unordered) set of (unordered) pairs of numbers. We can represent them by two arrays

integer array from, to[1:N-1]

where for each h satisfying $1 \leq h \leq N-1$ the pair "from[h], to[h]" gives (the numbers of) the endpoints of the h-th branch. In our final solution the branches will be numbered (by h), the only order that makes sense is the order in which they have been coloured red. The observation made earlier that the total number of branches (red and ultraviolet together) to be manipulated remains constant suggests that we store them in the same array:

if k = NUMBER OF RED POINTS

from[h], to[h] will be red for $1 \leq h < k$

from[h], to[h] will be ultraviolet for $k \leq h < N$.

The ultraviolet branches will be represented leading from a red point to a blue one. The array "length" has been introduced in order to reduce the number of subscriptions to be carried out:

length[h] = distance[from[h], to[h]]

will hold (and will be restored immediately when temporarily invalid).

Point N is chosen as the initial point to be coloured red. SELECT SHORTEST ULTRAVIOLET BRANCH is a straightforward search for a minimum value (see EWD287 - 46). COLOUR IT AND POINT P RED is done by an interchange in the three arrays (if necessary), followed by an increase of k. In ADJUST FOR EACH BLUE POINT B BY COMPARING WITH THE BRANCH BP, h scans the violet branches, to[h] will scann the blue points and len is used to store the length of branch BP. The final program is given on the next page.

Exercise.

Let distance[i,j] be the distance from point i to point j *in that direction*. As we may have one-way traffic, the relation

distance[i,j] $\neq$ distance[j,i]

is then permissible. Make a program finding in the graph the shortest path leading from point I to point J. This is a difficult exercise, therefore it is worth trying!

```
begin integer array from, to [1:N-1]; real array length [1:N-1]; real len, minlen; integer k, h, minh, p;
      COLOUR ONE POINT RED AND THE REMAINING ONES BLUE:
            k:= 1;
      CONSTRUCT THE SET OF ULTRAVIOLET BRANCHES:
            h:= 1; while h < N do begin from[h]:= N; to[h]:= h; length[h]:= distance[N,h]; h:= h + 1 end;
      while k < N do
      begin SELECT SHORTEST ULTRAVIOLET BRANCH AND CALL ITS BLUE ENDPOINT P:
                  minh:= k; minlen:= length[k]; h:= k + 1;
                  while h < N do begin len:= length[h]; if len < minlen do begin minlen:= len; minh:= h end; h:= h + 1 end;
                  p:= to[minh];
            COLOUR IT AND POINT P RED:
                  if k ≠ minh do begin h:= from[k]; from[k]:= from[minh]; from[minh]:= h;
                                       h:= to[k]; to[k]:= to[minh]; to[minh]:= h;
                                       len:= length[k]; length[k]:= length[minh]; length[minh]:= len
                                 end;
                  k:= k + 1;
            ADJUST FOR EACH BLUE POINT B BY COMPARING WITH THE BRANCH BP:
                  h:= k;
                  while h < N do
                  begin len:= distance[p, to[h]]; if len < length[h] do begin length[h]:= len; from[h]:= p end; h:= h + 1 end
      end;
      h:= 1; while h < N do begin print(from[h]); print(to[h]); newline; h:= h + 1 end
end
```

The towers of Hanoi.

Given three pegs, numbered 1, 2 and 3 and a set of N disks ($N \geq 1$) of decreasing diameter and a hole in their centre. At the beginning the N disks are all on peg nr.1 in order of decreasing diameter, i.e. the largest disk is at the bottom, the smallest is at the top side of the first peg. It is requested to move this tower from the first peg to the third one in a number of "moves", where a "move" is moving one disk from the top of a peg to the top of another peg with the restriction that never a larger disk may be placed on top of a smaller one. Hereby the second peg may be used as auxiliary "store" for pegs which are "in the way".

Now, if we can solve this game for any two pegs for $N = N_0$, we can also solve it for $N = N_0+1$. Call

```
movetower(m, A, B, C)
```

the set of moves that transports a tower of m disks from peg A (if necessary via peg B) to peg C. The individual moves are of the form

```
movedisk(A, B)    .
```

The set of moves

```
movetower(N0+ 1, A, B, C)
```

is the same as that of the successive moves of

```
movetower(N0, A, C , B)
movedisk(A, C)
movetower(N0, B, A, C)          .
```

In words: first a tower of N_0 disks is moved from A to B, using C as auxiliary peg, then the N_0+ 1st disk is moved directly from A to C and finally the tower of N_0, that has been placed temporarily on peg B is moved to its final destination C, using A as auxiliary peg. As the tower consists of the N_0 smallest disks, the presence of larger disks will not cause violation of the condition of decreasing disk diameter. Moving a one-disk tower ($N_0= 1$) presents no difficulty and as a result the puzzle has been solved. The question posed to us, however, is to make a program generating the diskmoves in the order in which they successively have to take place.

Note. It is not realistic to require the execution of this program for large values of N because the total number of moves required = $2^N - 1$.

The fact that a move with $N > 1$ is decomposed into three "smaller" moves, suggests that we keep a list of moves to be done. If the first one to be done is simple, we do it, otherwise we replace it by its decomposition and reconsider our obligations. In both cases, when we have a list of k moves, only the first to be done needs consideration and while we process it, the remaining K-1 moves remain as standing obligations. This suggests that we introduce

$$\text{move}_k,\quad \text{move}_{k-1},\ \ldots\ldots,\ \text{move}_2,\ \text{move } 1$$

to be done in the order of decreasing subscript, in the order from left to right. If move_k is simple, it is done, leaving

$$\text{move}_{k'=k-1},\ldots\ldots,\ \text{move}_2,\ \text{move}_1$$

(indicating with k' the new value of k, the length of the list of obligations) otherwise move_k is <u>replaced</u> by three others, leaving

$$\text{move}'_{k'=k+2},\ \text{move}'_{k'-1=k+1},\ \text{move}'_{k'-2=k},\ \text{move}_{k'-3=k-1},\ldots\ldots\text{move}_2,\ \text{move}_1\ .$$

In both transformations the lower (i.e. later) k-1 moves have been unaffected.

A move is given by four parameters, say

n	=	number of disks in the tower to be moved
from	=	number of source peg
via	=	number of auxiliary peg
to	=	number of destination peg.

We can store these moves in four arrays <u>integer</u> <u>array</u> n, from, via, to [1: 2*N-1]. (Verify that the list of obligations is never longer than 2*N-1 moves.)

The non-simple move (with $N > 1$), given in tabular form by

	n=	from=	via =	to =
k:	N	A	B	C

is to be replaced by the triple

```
begin integer k; integer array n, from, via, to [1:2*N-1];
    n[1]:= N; from[1]:= 1; via[1]:= 2; to[1]:= 3; k:= 1;
    repeat if n[k] = 1 then
        begin movedisk(from[k], to[k]);
            k:= k - 1
        end
                else
        begin n[k+2]:= n[k] - 1; from[k+2]:=from[k]; via[k+2]:= to[k]; to[k+2]:= via[k];
            n[k+1]:= 1; from[k+1]:= from[k]; to[k+1]:= to[k];
            n[k]:= n[k+2]; from[k]:= to[k+2]; via[k]:= from[k+2]; to[k]:= via[k+2];
            k:= k + 2
        end
    until k = 0
end
```

	n' =	from' =	via' =	to' =
k'- 2 = k:	N-1	B	A	C
k'-1 = k+2:	1	A	(B)	C
k' = k+2:	N-1	A	C	B

in which the top line replaces the original one, while the next two lines are new. (In the middle line the element "via" has been put between brackets, as it is immaterial; the program on page EWD287 - 69 leaves that value unaffected.)

Remark. In the program we have not described the details ov the operation "movedisk(A, B)". If it prints the number pair A,B, the solution will be printed; if the machine is coupled to a mechanical hand which really moves pegs, the machine will play the game!

The reader is strongly invited to follow the above program himself for small value of N (say: 4) so that he sees how the value of k goes up and down.

The reader is also invited to check carefully the "shunting" of the values N(-1), A, B and C when a non-simple move is decomposed into three simpler ones. This check is a painful process, so painful that everyone who has done it, will only be too willing to admit that the above program is rather ugly. The above program has been included with the aim of making him more appreciative of the elegance of the so-called "recursive solution" which now follows.

```
begin procedure movetower(integer value m, A, B, C);
      begin if m = 1 then movedisk(A, C)
                     else
            begin movetower(m-1, A, C, B);
                  movedisk(A, C)
                  movetower(m-1, B, A, C)
            end
      end;
      movetower(N, 1, 2, 3)
end
```

It introduces an operator named "movetower" with four (integer valued) parameters, moving a tower of length "m" from "A" via "B" to "C". In terms if this operator the final program collapses into a single statement as given in the last line, viz. "movetower (N, 1, 2, 3)". All that is given in front of it (lines 2 to 8) describes this operator in terms of a little program, little, because the operator is allowed to invoke itself. The definition of the operator -the so-called "procedure body"- follows completely our original analysis of the game (page EWD287 - 67). Recursive procedures -i.e. procedures that are allowed to invoke themselves- are such a powerful tool in programming that we shall give some more examples of them.

Remark. Some ot the more old-fashioned programming languages do not cater for recursion. Programming courses based on such programming languages often contain many exercises which are only difficult because the recursive solution is denied to the student.

The problem of the eight queens.

It is requested to make a program generating all configurations of eight queens on a chess board of 8 * 8 fields, such that no queen can take any of the others. This means that in the configurations sought no two queens may be on the same row, on the same column or on the same diagonal.

We don't have an operator generating all these configurations, this operator is exactly what we have to make. Now the (very general!) way to attack such a problem is as follows. Call the set of configurations to be generated A; look for a greater set B of configurations with the following properties
1) set A is a subset of set B;
2) given an element of set B, it is not too difficult to decide whether it belongs to set A as well;
3) we can make an operator generating the elements of set B.

With the aid of the generator (3) for the elements of set B, the elements of set B can then be generated in turn, they will be subjected to the decision criterion (2) which decides whether they have to be skipped or handed over, thus generating elements of set A. Thanks to (1) this algorithm will produce all elements of set A.

Three remarks are in order.
1) If the whole approach makes sense, set B is not identical to set A and as it must contain set A as a (true) subset, it must be larger. Nevertheless, it is advised to choose it "as small as possible": the more elements it has, the more elements of it have to be rejected on account of the decision criterion (2).
2) We should look for a decision criterion that is cheap to apply, at least the discovery that an element of B does not belong to A should (on the average) be cheap.
3) The assumption is that the generation of the elements of set B is easier than a direct generation of the elements of set A. If, nevertheless, the generation of the elements of set B still presents difficulties we repeat our pattern of thinking, re-apply the trick and look for a still larger set C of configurations that contain B as subset, etc.

Above we have sketched a very general approach, applicable to may, very different problems. Faced with a particular problem, i.e. faced with a specific set A, the problem of course is, what to select for our set B.

In a moment of optimism one could think that this is an easy matter, as we might consider the following technique. We list all the mutually independent conditions that our elements of set A must satisfy and omit one of them. Sometimes this works but as a general technique this is too naive; if we want to see its shortcomings, we only need to apply it blindly to the problem of the eight queens. We can characterize our solutions by the conditions:

1) there are 8 queens on the board

2) no two of the queens can take one another.

Omitting either of them gives for the set B the alternatives

B1: all configurations with N queens on the board such that no two queens can take one another

B2: all configurations of 8 queens on the board.

But both sets are so ludicrously huge that they lead to utterly impractical algorithms. We have to be smarter. How?

Well, at this stage of our considerations, being slightly "at a loss" we are not so much concerned with the efficiency of our final program but rather with the efficiency of our own thought processes! So, if we decide to make a list of the properties of solutions, in the hope of finding a useful clue, this is a rather undirected search, we should not invest too much mental energy in such a search, that is: for a start we should restrict ourselves to their obvious properties. Let us go ahead.

a) No row may contain more than one queen, 8 queens are to be placed and the chess board has exactly 8 rows. As a result we conclude that each row will contain exactly one queen.

b) Similarly we conclude that each column will contain exactly one queen.

c) There are fifteen "upward" diagonals, each of them containing at most one queen, i.e. 8 uoward diagonals contain exactly one queen and 7 upward diagonals are empty.

d) Similarly we conclude that 8 downward diagonals are occupied by one queen and 7 are empty.

e) Given any non-empty configuration of queens such that no two of them can take one another, removal of any of these queens will result in a configuration sharing that property.

Now the last one is a very important property: in our earlier terminology it tells us something about any non-empty configuration from set B1. Conversely it tells us that each non-empty configuration from B1 can be generated (in N different ways!) by extending a configuration from B1 with N-1 queens by another queen. We have rejected B1 because it was too large, but maybe we can find a suitable subset of it, such that each non-empty configuration of the subset is a one-queen extension of only one other configuration from the subset. This "extension property" suggests that we are willing to consider configurations with less than 8 queens and that we would like to form new configurations by adding a queen to an existing configuration - a relatively simple operation presumably. Well, this draws our attention immediately to the generation of the elements of the (still mysterious) set B. For instance: in what order? And this again raises a question to which, as yet, we have not paid the slightest attention: in what order are we to generate the solutions, i.e. the elements of set A? Can we make a reasonable suggestion in the hope of deriving a clue from it?

Prior to that we should ask ourselves: how do we characterize solutions once we have them? To characterize a solution we must give the positions of 8 queens. The queens themselves are unordered, but the rows and the columns are not: we may assume them to be numbered from 0 through 7. Thanks to property a), which tells us that each row contains exactly one queen, we can order the queens according to the number of the row they occupy. Then each configuration of 8 queens can be given by the value of the integer array x[0:7], where

x[i] = the number of the column occupied by the queen in the i-th row.

Each solution is then "an 8-digit word" (x[0]...x[7]) and the only sensible order in which to generate these words that I can think of is the alphabetic order.

(Note. As a consequence we open the way to algorithms in which rows and columns are treated differently. At first sight this is surprising, because the original problem is completely symmetrical in rows and columns. We should be glad: to consider asymmetric algorithms is exactly what the above considerations have taught us!)

Returning to the alphabetic order: now we are approaching familiar grounds. If the elements of set A are to be generated in alphabetic order and then have to be generated by selecting them from a larger set B, then the standard technique is generating the elements of set B in alphabetic order as well and to produce the elements of the subset in the order in which they occur in set B.

First we have to generate all solutions with $x[0] = 0$, then all with $x[0] = 1$ etc.; the solutions with $x[0] = 0$ have to be generated first those with $x[1] = 0$ (if any), then those with $x[1] = 1$ (if any), then those with $x[1] = 2$ (if any) etc. In other words: the queen of row 0 is placed in column 0 -say: the field in the top left corner- and remains there until all elements of A (and B) with queen 0 in that position have been generated and only then is she moved one filed to the right to the next column. For each position of queen 0, queen 1 will walk from left to right in row 1 -skipping the fields that are covered by queen 0-; for each combined position of the first two queens, queen 2 walks along row 2 from left to right, skipping all fields covered by the preceding queens, etc.

But now we have found set B! It is indeed a subset of B1: set B consists of

> all configurations with one queen in each of the first N rows, such that no two queens can take one another.

Having established our choice for the set B, we find ourselves immediately faced with the task of generating its elements in alphabetic order. We could try to do this via an operator "GENERATE NEXT ELEMENT OF B" with a program of the form

```
INITIALIZE EMPTY BOARD;
repeat GENERATE NEXT ELEMENT OF B;
       if N = 8 do PRINT CONFIGURATION
until B exhausted
```

but this is not too attractive for the following two reasons.

Firstly, we don't have a ready-made criterion to recognize the last element of B when we meet it and in all probability we have to generalize the operator "GENERATE NEXT ELEMENT OF B" in such a way that it will produce the indication "B EXHAUSTED" -for instance in the form of the empty board- when it is applied to the last "true" element of B. Secondly, it is not too obvious how to make the operator "GENERATE NEXT ELEMENT OF B": the number of queens on the board may remain constant, it may increase and it may decrease.

So that is not too attractive. What can we do about it? As long as we regard the sequence of configurations from set B as a single sequence, not subdivided into a succession of subsequences, the corresponding program structure will be a single loop as in the program just sketched. If we are looking for an alternative program structure, we must therefore ask ourselves: "How can we group the sequence of configurations from set B into a succession of subsequences?".

Realizing that the sequence of configurations from set B have to be generated in alphabetic order and thinking of the main subdivision in a dictionary -viz. by first letter- the first grouping is obvious: by position of queen 0.

Generating all elements of set B -for the moment we forget about the printing of the elements that belong to the subset A as well- the presents itself in the first instance as

```
h:= 0;
repeat SET QUEEN ON FIELD H;
       GENERATE ALL CONFIGURATIONS WITH QUEEN 0 FIXED;
       REMOVE QUEEN;
       h:= h + 1
until h = 8    ,
```

where the operations SET QUEEN and REMOVE QUEEN pertain to row zero, i.e. the first free row or the last occupied row respectively.

But now the question repeats itself: how do we group all configurations with queen 0 fixed? We have already given the answer: in order of increasing column position of queen 1, i.e.

```
h1:= 0;
repeat if FIELD H1 FREE do
       begin SET QUEEN ON FIELD H1;
             GENERATE ALL CONFIGURATIONS WITH FIRST 2 QUEENS FIXED;
             REMOVE QUEEN
       end;
       h1:= h1 + 1
until h1 = 8
```

where, again, FIELD FREE and SET QUEEN pertain to the first free row and REMOVE QUEEN pertains to the last occupied row.

For "GENERATE ALL FIELDS WITH FIRST 2 QUEENS FIXED" we could write a similar piece of program and so on: inserting them inside each other would result into a correct program with some eight nested loops, but they would all be very, very similar. To do so has two disadvantages:

1) it takes a cumbersome amount of writing

2) it gives a program solving the problem for a chess board of 8 * 8 fields, but to solve the same puzzle for a board of, say, 10 * 10 fileds would require a new (still longer) program. We would like to avoid this by exploiting the similarity of the loops.

The we have to answer two questions

1) can we make the loops exactly identical?

2) can we profit from their similarity?

The two exceptional cycles are the outermost one and the innermost one. The outermost one is different because it does not test, whether the next field is free. There is however no objection to inserting this test: as it is only applied when the board is empty it is guaranteed to give the value true and we can give the outermost cycle the same form by inserting the conditional clause

```
        if FIELD H FREE do.
```

The innermost cycle is exceptional in the sense that as soon as 8 queens have been placed on the board, there is no point in generating all configurations with those queens fixed, because we have a full board. Instead the configuration has to be printed, because we have found an element of set B that is also an element of set A. We can map the innermost cycle and the embracing seven ones upon each other by replacing the line "GENERATE" by

```
        if BOARD FULL then PRINT CONFIGURATION
                      else GENERATE ALL CONFIGURATIONS EXTENDING THE CURRENT ONE.
```

By now the only difference between the eight cycles is that each cycle has to have "its private h". By the time that we have reached this stage, we can give an affirmative answer to the second question. The sequencing through the eight nested loops can be provoked with the aid of a recursive procedure "generate" say which describes the cycle once. Using it, the program itself collapses into

```
    "INITIALIZE EMPTY BOARD;
     generate"
```

while "generate" is defined recursively as follows:

```
procedure generate;
begin integer h;
      h:= 0;
      repeat if FIELD H FREE do
             begin SET QUEEN ON FIELD H;
                   if BOARD FULL then PRINT CONFIGURATION
                                 else generate;
                   REMOVE QUEEN
             end;
             h:= h + 1
      until h = 8
end    .
```

Each activation of "generate" will introduce its private local variable h, this catering for h, h1, h2, ... that we would need when writing 8 nested loops inside each other. FIELD H FREE and SET QUEEN ON FIELD H again refer to the first free row, the operation REMOVE QUEEN to the last occupied one.

Our program -although correct to this level of detail - is not yet complete, i.e. it has not been refined up to the standard degree of detail that is required by our programming language. In our next refinement we should decide upon the conventions according to which we represent the configurations on the board. We have already decided more or less that we shall use the

```
          integer array x[0:7]
```

giving in order the column numbers occupied by the queens. We need a separate convention to represent the number of queens on the board. Let us introduce integer n, such that

n =   the number of queens on the board

and for $0 \leq i < n$: $x[i]$ = the number of the column occupied by the queen in the i-th row.

The array x and the scalar n are together sufficient to fix any configuration of the set B and those will be the only ones on the chess board. As a result we have no logical need for more variables; yet we shall introduce a few more because from a practical point of view we can make good use of them. The problem is that with only the above material the analysis whether a given field in the next free row is unciverd is rather painful and time-consuming. Here we can look for a standard technique, called "trading storage space versus computation time". The pattern of this technique is as follows.

In its most simple form we are faced with a computation that regularly needs the value of "FUN(arg)" where "FUN" is a given, computable function defined on the current value of one or more stored variables, collectively called "arg". In version 1 of a program only the value of arg is stored and the value of FUN(arg) is computed whenever needed. In version 2, an additional variable, "fun" say, is introduced whose sole purpose is to record the value of "FUN(arg)" corresponding to the current value of "arg".

Where version 1 has

```
        arg:=...            (i.e. assignment to arg)
```

version 2 will have (effectively)

```
        arg:=...; fun:= FUN(arg)  ,
```

thereby maintaining the validity of the relation

$$fun = FUN(arg) \quad .$$

As a result of the validity of this relation, wherever version 1 calls for the evaluation of FUN(arg), version 2 will call for the current value of the variable "fun".

The introduction of such redundant additional tabulated material is one of the programmer's most powerful ways to improve the efficiency of a program. Of course we need our ingenuity for its invention!

Quite often the situation is not as simple as that and we come now to the second reason for introducing such a variable "fun". Often it is very unattractive to compute FUN(arg) from scratch for arbitrary values of arg while it is much easier to compute how the value of FUN(arg) changes when the value of arg is changed. In that case the adjustment of the value of "fun" is more intimately linked with the nature of the functional dependence and the history of the variable "arg" than is suggested by

```
        arg:=...; fun:= FUN(arg)    .
```

After this interlude on program optimization via trading storage space versus computation time, we return to our eight queens. The role of "arg" is played by the configuration on the board, but this value is not changed wildly, oh no, the only thing we do with it is adding or removing a queen. And we are looking for additional tables that will assist us in the decision whether a field is free, tables such that they can be kept up to date easily when a queen is added to or removed from the configuration.

How? Well, we might think about a boolean array of 8 * 8, indicating for each field whether it is free or not. If we do this for the full board, adding a queen implies dealing with 29 fields, removing a queen, however, is then a painful process because it does not follo that all fields no longer covered by <u>her</u> are indeed free: they might be covered by other queens. There is a remedy for this, viz. associating with each field not a boolean variable but an integer counter, counting the number of queens covering the field. Addind a queen means increasing 29 counters by 1, removing a queen means decreasing 29 counters by 1 and a field is free when its counter is zero. We could do it that way, but the question is whether this is not overdoing it: 29 adjustments is quite a lot.

Each field, in the freedom of which we are interested, covers a row (which is free by defenition, so we need not bother about that), covers one of 8 columns (one which still should be empty), covers one of the 15 upward diagonals (one which should be empty) and one of the 15 downward diagonals(one which should be empty). This suggests that we should keep track of

1) the columns that are free

2) the upward diagonals that are free

3) the downward diagonals that are free.

As each column or diagonal is covered only once we don't need a counter for each, a boolean is sufficient. For the columns we introduce a

```
        boolean array col[0;7]
```

where "col[i]" means that the i-th column is still free.

How do we identify the diagonals? Well, along an upward diagonal the difference between row number and column number is constant, along a downward diagonal their sum. As a result difference and sum respectively are the easiest index by which to distinguish the diagonals and we introduce therefore

```
boolean array up[-7:+7], down[0:14]
```

to keep track of which diagonals are free.

The question whether field[n,h] is free becomes

```
col[h] and up[n-h] and down[n+h]    ,
```

setting and removing a queen both imply adjustment of three booleans, one in each array.

Without the tabulated material, REMOVE QUEEN would only consist of "n:= n - 1", now we would like to know her column number as well, i.e. we replace it by REMOVE QUEEN FROM FIELD H". In the final program, the variable "k" is introduced for general counting purposes, statements and expressions are labelled for explicative purposes.

```
begin integer n,k; integer array x[0:7]; boolean array col[0:7], up[-7:+7], down[0:14];
    procedure generate;
    begin integer h; h := 0;
        repeat if FIELD H FREE: (col[h] and up[n-h] and down[n+h]) do
             begin SET QUEEN ON FIELD H:
                     x[n]:= h; col[h]:= false; up[n-h]:= false; down[n+h]:= false; n:= n + 1;
                 if BOARD FULL: (n = 8) then
                 begin PRINT CONFIGURATION:
                         k:= 0; repeat print(x[k]); k:= k + 1 until k = 8; newline
                 end
                                 else generate;
                 REMOVE QUEEN FROM FIELD H:
                     n:= n - 1; down[n + h]:= true; up[n-h]:= true; col[h]:= true
             end;
             h:= h + 1
        until h = 8
    end;
    INITIALIZE EMPTY BOARD:
    n:= 0;
    k:= 0; repeat col[k]:= true; k:= k + 1 until k = 8;
    k:= 0; repeat up[k-7]:= true; down[k]:= true; k:= k + 1 until k = 15;
    generate
end
```

This completes the treatment of our problem; the program, incidentally, generates 92 configurations.

By way of conclusion I would like to make up the bill: the final solution is not very important (at least not more important than the problem of the eight queens). The importance of this section is to be found in the methods on which our final program relies and the way in which we have found them.

1) The final algorithm mebodies a very general technique, so general that it has a well-established name: it is called "backtracking". The configurations of set B can be thought of as placed at the nodes of a hierarchical tree, each node containing configuration C supporting the subtree with all the nodes with configurations that have C as a true sub-configuration. At the root of the tree we have the empty configuration (from which 8 different branches emanate), at each next level we find configurations with one queen more and at the top nodes (the leaves) we find the 92 solutions. The backtracking algorithm generates and scans the nodes of this tree in a systematic manner. I recommend the reader to become thoroughly familiar with the idea of backtracking, because its can be applied when faced with a great number of at first sight very different problems. (It is only when you <u>recognize</u> that they all ask for a solution by means of backtracking that the problems become boringly similar to each other.)

2) If the only thing the student learns from this section is his becoming familiar with backtracking, he has learned something, but it was my intention to teach him more: we showed all considerations which together can lead to the <u>discovery</u> of our method, this time to backtracking. But it is my firm conviction that, when faced with a different problem to be solved by a different method, the latter may be discovered by a very similar analysis of ends and means.

3) The finla program contained a recursive procedure. But backtracking is by no means the only algorithmic pattern that is conveniently coded with the aid of recursion. The main point was the collection of considerations leading to the <u>discovery</u> that in this case recursion was an appropriate tool.

4) The major part of our analysis has been carried out before we had decided how (and how redundantly) a configuration would be represented inside the machine.

It is true that such considerations only bear fruit when, finally, a convenient representation for configurations can be found. Yet it is essential not to bother about the representation before that becomes crucial. There is a tendency among programmers to decide the (detailed) representation conventions first and then to think about the algorithm in terms of this specific representation but that is putting the cart before the horse. It implies that any later revision of the representation convention implies that all thinking about the algorithm has to be redone; it fails to give due recognition to the fact that the only point in manipulating (such groups of) variables is that they stand for something else, configurations in our case.

5) The trading of storage space versus computation time is more than a trick that is useful in this particular program. It is exemplar for many of the choices a producing programmer has to take; he will work more consciously and more reliably when he recognizes them as such.

Exercise. Write two programs generating for $N > 0$ all $N!$ permutation of the numbers 1 through N, one with and one without recursion, and establish for both programs their correctness.

Exercise. For $) < N < M$ generate all integer solutions of the equations in $c[1]$ through $c[N]$ such that

1) $c[1] \geq 0$

2) $c[i] \geq c[i-1]$ for $1 < i \leq N$

3) $c[1] + \ldots + c[N] = M$.

Again, write two programs, one without and one with recursion and establish their correctness.

Exercise. Write a program solvong the problem of page EWD287 - 57, but now using recursion.

A rearranging routine.

The following example has been inspired by work of C.A.R.Hoare (Algorithm 64, C.A.C.M.).

The original problem was to rearrange the values of the elements of a given array $A[1:N]$ and a given value of f $(1 \leq f \leq N)$ such that after rearrangement

$$\begin{array}{ll} \text{for } 1 \leq k < f & A[k] \leq A[f] \\ \text{for } f < k \leq N & A[k] \geq A[f] \quad . \end{array} \qquad (1)$$

As a result of this rearrangement $A[f]$ equals the f-th value in order of non-decreasing magnitude. We call the array rearranged satisfying (1) "split around f"; we call the final value of $A[f]$ "the splitting value". When the array has been split it is divided into two halves, the one half -the "left-hand" half, say- containing the "small" values and the other half -the "right-hand" half, say- containing the large elements, with the splitting value sandwiched in between. The overall function of the algorithm is to move small values to the left and large values to the right. The difficulty is that for given f the final value of $A[f]$, i.e. our criterion "small/large", is unknown to start with.

Hoare's invention is the following. Select some, rather arbitrary criterion "small/large"; by moving small elements to the left and large elements to the right, a split will be established somewhere, around some position s. If s happens to turn out = f, the original problem is solved. The kernel of Hoare's invention is the observation that ~~otherwise~~ in the other cases the original problem ~~is now~~ can be reduced to the same problem, but now applied to one of the halves, viz. to the left-hand half if f lies to the left of the split and to the right-hand half is f lies to the right of the split.

Note. An alternative approach would be to sort the array completely: after that $A[f]$ will equal the f-th value in the order of non-decreasing magnitude. But this can be expected to be rather expensive, for then we have established relations (1) for all values of f. As a matter of fact we will arrive at a rearranging routine which itself can be used for complete sorting, on account of the fact that, when a split around s has been established, $A[s]$ has the value it is going to have in the completely sorted array, and that -because all elements to the left of it are $\leq$ A[s] and those to the right of it are $\geq$ A[s]- completely sorting it could now be performed by sorting thereafter the two parts ~~to the left and to the right of A[s] respectively seperately.~~ independently.

We now focus our attention ~~to~~ on the rearranging routine which is to cause a split in the array section

A[m] ... A[n]   with $1 \leq m \leq n \leq N$.

When we try to make such a routine we are immediately faced with the choice of our criterion "small/large". One of the ways is to select an arbitrary element from the section, to call all elements larger than it "large", all elements smaller than it "small" and all elements equal to it either "large" or "small", just what is most convenient (in the final ~~XXX~~ arrangement they may occur everywhere, either at the split or at ~~any~~ either of its two sides). Let us therefore postpone the choice in this discussion for a moment, as there is a chance that we can use our freedom ~~at~~ to some advantage.

We are going to select one of the values in the array as the "splitting value"; having chosen this value, its final position - i.e. the position of the split- is unknown before the algorithm starts; it is defined when the algorithm has been executed, in other words it is determined by the evolution of the computation. This suggests that we build up the collection of the small values, starting at the left-hand end, and that of the large values at the right-hand end, and continue doing so until the two collections meet somewhere in the middle. To be more precise, we introduce two pointers, "i" and "j" say, whose initial values will be "m" and "n" respectively, and rearrange values in such a fashion that, when we call the splitting value V, we ensure that

A[k] $\leq$ V   for $m \leq k < i$ and
A[k] $\geq$ V   for $j < k \leq n$.

Having chosen the splitting value, the algorithm will have the duty ~~to~~ of building up the collections of small and large values repectively from the outside inwards. The algorithm can start scanning, at the left-hand end say, until a large element is encountered. If ~~so~~ this occurs, this value has to be removed from the collection of small values, i.e. it has to be added to the collection of large elements. It is, as a matter of fact, the first element ~~of which~~ whose "bigness" the algorithm has established ~~its "largeness"~~: as we have decided to build up the collections from the outside inwards, this "large" value has to be assigned to A[n]. As we would like this postion in the array to be "free" -i.e. available to receive this first large value- the original value of A[n] can be taken out of the array and can be chosen as the splitting value V.

That is, we initialize i = m and j = n and "take out" A[n] -by assigning it to the variable V- thereby initializing the situation where scanning starts at element A[i], while "j" points to the "hole" just made. When the upward scan (under control of increasing "i") finds a large element, i.e. when for the first time A[i] > V, this value is placed in the hole, now leaving the hole at the place pointed to by "i". From then onwards a downward scan (under control of decreasing "j") can ~~start~~ operate until a small element has been encountered which will be placed in the hole at position "i", leaving the hole at the place pointed to by "j". Such upward and downward scans have to succeed each other alternately until i = j, i.e. until both point to the hole at the position around which the split has been effectuated. Finally the hole receives the value V which had been taken out at the beginning.

The above sketch gives an informal description of the essential features of the algorithm, it settles by no means the structure of the sequential program that will embody it.

I have tried a program in which the core consists of the program part for the upward scan followed by the program part for the downward scan. The first part consists of a loop with "i:= i + 1" in the repeatable statement; the second part consists of a loop with "j:= j - 1" in the repeatable statement. The two parts together then form the repeatable statement of an ~~outward~~ outer loop. This program became very ugly and messy, the reason being that termination may occur either because the upward scan or because the downward scan is on the verge of scanning the hole. The reasoning needed to establish ~~the~~ that the program did properly terminate became tortuous.

On account of that experience I have tried the alternative approach, one loop in which a single execution of the repeatable statement decreases the difference "j - i" -i.e. the length of the unscanned array section- by 1, by doing a step of the appropriate scan.

The decision to control the steps of *both* scans by the *same* repeatable statement calls for the introduction of another variable; as we have to distinguish between only two cases, a boolean variable suffices, "up" say, with the meaning:

up = *true* means:

the algorithm is in the state of upward scanning and j points to the hole

up = *false* means:

the algorithm is in the state of downward scanning and i points to the hole.

The initialization has to be extended with the assignment "up := *true*"; after the initialization the program continues with

"*while* i < j *do* perform the appropriate step" .

In the course of the action "perform the appropriate step", the value of "up" has to change its value whenever the hole is filled and the scanning direction has to reverse. Without any further detours I arrived at the following procedure:

```
integer procedure split(real array a, integer value m, n);
begin integer i, j; real V; boolean up;
      i:= m; j:= n; V:= a[j]; up:= true;
      while i < j do
      begin if up then
               if a[i] > V do begin a[j]:= a[i]; up:= false end
                   else
               if V > a[j] do begin a[i]:= a[j]; up:= true end;
            if up then i:= i + 1 else j:= j - 1
      end;
      a[j]:= V; split:= j
end
```

In its applications we shall only call the procedure "split" with m < n; as it stands it also caters for the case m = n.

Exercise. Show that in a version of split that only needs to cater for m < n, its internal repetition could have been controlled by a *repeat until* clause as well.

Note. At the price of a larger number of subscriptions to be performed, the text of the procedure can be shortened by not introducing the separate variable

V, but by storing this value "in the hole" i.e.

V = *if* up *then* a[j] *else* a[i] . .

As a result the splitting value zigzags to its final position. With the above convention the tests "a[i] > V" and "V > a[j]" both become "a[i] > a[j]", the assignments XX "a[j]:= a[i]" and "a[i]:= a[j]" become both the interchange

"W:= a[i]; a[i]:= a[j]; a[j]:= W"

and the assignments "up:= *false*" and "up:= *true*" can both be represented by

"up:= *non* up" .

The above observations allow us to condense the procedure text into

```
integer procedure split(real array a, integer value m, n);
begin integer i, j; real W; boolean up;
      i:= m; j:= n; up:= true;
      while i < j do
      begin if a[i] > a[j] do
                begin W:= a[i]; a[i]:= a[j]; a[j]:= W; up:= non up end;
            if up then i:= i + 1 else j:= j - 1
      end;
      split:= j
end   .
```

We now return to our original problem: given an array A[1:N] and a value f ($1 \leq f \leq N$), rearrange the elements in such a way that

for $1 \leq i < f$ $\quad A[i] \leq A[f]$ and
for $f < i \leq N$ $\quad A[i] \geq A[f]$;

as a result A[f] will equal the f-th element in the order of non-decreasing magnitude.

The idea is to apply the operator "split" first to the original array from 1 through N. The operator establishes the split somewhere, position s say. If the position of the split coincides with f (f = s), we have reached our goal, otherwise the operator "split" is applied to one of the halves, viz. to the left-hand half when f < s and to the right-hand half when f > s etc.

For this purpose we introduce variables p and q, satisfying

$$1 \leq p \leq f \leq q \leq N$$

such that A[p] .... A[q]

will be the section of the array to which the split will be applied, as this section is certain to contain the (future) value of A[f].

If the split is found to the right of f (i.e. $f < s$) the operator has to be applied to the left-hand half, i.e. q has to be reset to $s - 1$ and p can be left unchanged. In the case $f > s$, p has to be reset to $s + 1$ and q can be left unchanged. We thus arrive at the routine

```
integer p, q, s; p:= 1; q:= N;
repeat s:= split(A, p, q);
       if f < s do q:= s - 1;
       if f > s do p:= s + 1
until f = s .
```

(Note. In the case N = 1 this routine will call split with m = n.)

We may wish to improve upon this program: it is rather superfluous to call the operator "split" with p = q: if the section consists of a single element no (significant) rearrangement can take place; the split will be around its single element and both halves will be empty. The relation $p < q$ gives us therefore another necessary criterion for continuation, and we can look to see whether we can make it the sole criterion for continuation. Because we want to stick to $p \leq f \leq q$, the termination via the luck of hitting f with the split, i.e. f = s, has to generate p = f = q. The following program would achieve that result.

```
integer p, q, s; p:= 1; q:= N;
while p < q do
begin s:= split(A, p, q);
      if f = s then begin p:= f; q:= f end
                else if f < s then q:= s - 1 else p:= s + 1
end
```

From the above program text it is obvious that the operator "split" will only be applied to sections of the array containing at least two elements.

A more intricate use of the operator "split" is in complete sorting of the array, observing that after application of the operator "split" at least one element (viz. A[s]) has reached its final destination, while all other elements, although not necessarily in their final position, will be in the correct half, so that complete sorting then consists of sorting both halves independently.

The naive approach is a recursive

```
procedure sort(real array a, integer value p, q);
begin integer s;
      s:= split(a, p, q);
      if p < s - 1 do sort(a, p, s - 1);
      if s + 1 < q do sort(a, s + 1, p)
end
```

such that the call

sort(A, 1, N)

will sort the entire array. Again it has been exploited that sorting an array section is only a necessary operation if the section contains at least two elements. (The routine "sort" may be called with a section of only one element, but will not generate such calls itself.)

We have called the above procedure naive and we have done so for the following reasons. The operator "split" may divide the section offered to it into two very unequal parts -e.g. when originally the rightmost element had a near maximum value-; as a result the maximum dynamic depth of recursive calls may grow proportionally to N, the length of the array section. As recursive calls require an amount of storage space proportional to the dynamic depth, the given program may turn out to be prohibitively demanding in its storage requirements. This would lead to the conclusion that recursive sorting is ~~un~~impractical, but for the fact that a slight rearrangement of the procedure "sort" ensures that the maximum dynamic depth will not exceed $\log_2 N$. In view of the existence of such a sorting procedure we call the previous one "naive".

We can guarantee that a sorting routine will not generate a dynamic depth exceeding $\log_2 N$, if whenever it has called "split", it will only prescribe a recursive call on itself for the sorting of the smallest of the two halves. (In the case that the two halves are of equal length, the choice is immaterial.) Applying "sort" recursively to the smallest half only will leave the other half unsorted, but this can be remedied by repeatedly applying this only half-effective sorting effort to the still unsorted section. In the body of "sort", two integers "pu" and "qu" are introduced, pointing to the left- and right-hand end of the still unsorted section.

```
procedure sort(real array a, integer value p, q);
begin integer s, pu, qu;
      pu:= p; qu:= q;
      while pu > qu do
      begin s:= split(a, pu, qu);
            if qu - s < s - pu then
               begin if s + 1 < qu do sort(a, s - 1, qu); qu:= s - 1 end
                                 else
               begin if pu < s - 1 do sort(a, pu, s - 1); pu:= s + 1 end
      end
end
```

Again, sort may be called with a section of a single element, but will not generate such calls itself.

Exercise. Prove that termination of the loop is guaranteed to take place with pu = qu.

Note. If, to start with, the elements of array A are ordered according to non-decreasing magnitude, excessive depth of recursive calls has been prevented, but the algorithm remains time-consuming (proportional to N^2). This has given rise to refinements of the procedure "split": instead of blindly taking the right-most element of the array section as splitting value, some sort of small search for a probably better approximation of the median value can be inserted at the beginning of "split"; this element can be interchanged with the rightmost element and thereafter split can continue as described.