EWD239.html

On trading storage against computation time.

In present day sequential computers we can distinguish two main components, an active one (the processor) and a passive one (the store). The active component has the specific function to be fast, the passive one has the specific function to be large. The following is written under the assumption that this functional division is here to stay. (Personally I feel that it is this functional division thanks to which the reliable construction of both large and fast computers has become possible.

From the point of view of the programmer storage space and computation time are two distinct resources and I regard it as one of the responsibilities of the programmer -rather than of the system- to allocate them, i.e. to divide the load between them. It is to the consequences of this responsibility that the present report is devoted. The report is not devoted to techniques needed to estimate the various loads, i.e. to give quantitative criteria to influence the programmer's choice. It is devoted to the logical relation between the alternatives among which the programmer has the choice.

I claim that the basic relation between two alternatives is given by the following pattern.

Given a program asking regularly for the value "FUN(arg)" where "FUN" is a given function defined on the current value of one or more variables of the state space called (together) "arg".

In version A, only the current value of "arg" is stored and the value "FUN(arg)" is computed whenever needed.

In version B, an additional variable "fun" say, is introduced, whose sole purpose is to record the value FUN(arg) for the current value of arg.

Where version A has

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

version B will have

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

thereby maintaining the relation

"fun = FUN(arg)"

Where version A calls for "FUN(arg)", version B will call for the current value of "fun".

There are two possible reasons to prefer version B above version A.

1) When the value of "FUN(arg)" is more frequently requested than assignments to "arg" take place, version B requires less frequent computation time. (If necessary, the technique can be refined by the introduction of a boolean "fun up to date" indicating whether "fun = FUN(arg)" holds. Assignment to arg is the

"arg:=.....; fun up to date:= <u>false</u>";

in this way one can suppress superfluous computations of FUN(arg) as might be generated by version B.)

2) Often it is difficult to compute "FUN(arg)" from scratch for arbitrary values of "arg", but is it much easier to compute the <u>change</u> of "FUN(arg)" on account of the <u>change</u> of "arg". (often this consideration will be the very body of the algorithm, e.g. when FUN is defined in terms of a recurrence relation; see the last example of EWD238, where "j" has to be taken as one of the arguments.)

We shall now turn to an example with which we have played extensively. Consider 32 positions arranged in a circle. Please make a program generating all ways in which they can be filled with 0's and 1's (one digit per position) such that the 32 quintuples of adjoining positions present the 32 possible configurations of five binary digits. Solutions that can be mapped on each other by rotation will be regarded as equivalent, the solutions are to be given as strings of 32 digits with the five zeros leading and they have to be generated in alphabetical order.

C.Ligtmans has shown that this cyclic problem is equivalent to the following linear one. Fill a linear array of 36 positions with 0's and 1's (one digit per position) such that the 32 quintuples of adjoining positions present the 32 configurations of five binary digits. (When the linear array contains a solution the sequence formed by the first four digits will be equal to the sequence formed by the last four digits and as a result "the ring will close".) We shall tackle the linear problem.

We introduce a structured variable called "sequence" consisting of four zeros followed by k binary digits. When k=0 we call the sequence empty, when k=32 we call the sequence full; k equals the number of ways in which a quintuple of five adjoining positions can be chosen in the sequence.

Furthermore we introduce a binary variable, called "candidate", representing the value by which the algorithm seeks to extend the sequence. The operation inverse to "extend sequence with candidate" is "take candidate from sequence", by which the last digit is taken away from the sequence and is assigned to the candidate.

The algorithm maintains the sequence in such a way that the sequence will never contain two different quintuples presenting the same pattern of five digits. The question "sequence extanedable with candidate" asks whether extension would not give rise to a sequence violating the condition of different digit patterns. We can make the following two remarks

1) As k equals the number of quintuples, as all quintuples on the sequence must present different digit patterns and the number of different digit patterns equals 32, k will satisfy "$k \leq 32$".

2) Whenever k=32 the sequence presents a solution of the linear problem and on account of the equivalence established by Ligtmans the 32 leading digits of the sequence present a solution of the circular problem.

We now give the structure of the backtracking algorithm that prints the solutions in alphabetical order

```
"set sequence empty; set candidate to zero;
repeat
     if sequence extendable with candidate then
          begin extend sequence with candidate; set candidate to zero;
               if sequence full do print solution
          end
                                        else
          begin while candidate set to one do take candidate from sequence;
               set candidate to one
          end
until sequence empty"
```

The above has been described as "the structure" of an algorithm. Can we regard it as "an algorithm"? I think we can.

The nine statements ("set sequence empty"; "set candidate to zero" etc.) have to be regarded as names of instructions of the well-understood repertoire (the primitive repertoire), "candidate" and "sequence" have to be understood as coordinates of the state space.

What do we have to assume (or: to define?) about the operations and the objects in order that this program makes sense? (I apologize for this probably impure question, for what is "making sense"? Nevertheless, I go on.)

Candidate is a two state object (the states being called "zero" and "one" respectively), these states can be set and inspected.

Sequence is a finite state object. A unique state of this object is called "empty". (Otherwise "set sequence empty" would be indeterminate.)

On a value pair seuence-candidate a boolean function "sequence extendable with candidate" is defined; if this function is true, the operation "extend sequence with candidate" is defined, giving a new sequence, and "take candidate from sequence" is then the inverse operation.

There is a boolean function "full" defined on the object sequence; its value on the empty sequence is irrelevant.

The object values that can be made by further extension of the empty sequence extended with a zero are scanned, whenever a value satisfying the criterion"full" is met (a function of) it is printed.

We can regard a set of properties as the above .when more carefully defined- as a set of axioms on objects and operations and on this level they must be sufficient to understand this program, to prove that it ends, that it produces all values satisfying "full" and all only once etc. On this same level we can make more algorithms, in this sense we have "a real machine"! E.g.

```
Set sequence empty; set candidate to zero;
if sequence extendable with candidate do
begin extend sequence with candidate;
    repeat
    if sequence extendable with candidate then
        begin extend sequence with candidate; set candidate to zero;
              if sequence full do print solution
        end
                                          else
        begin while candidate set to one do take candidate from sequence;
              take candidate from sequence;
              if  non sequence empty do extend sequence with candidate;
              set candidate to one
        end
    until sequence empty
end
```

This program scans the object values that can be made by further extension of the sequence value created by extending the empty sequence twice with a zero. My question is: to what extent can we regard the particular example of the 36 positions as a specific model, a representation? Or: can we regard the above "abstract algorithm" as the abstracted ancestor of a wide class of backtracking algorithms? And, if so, to what use?

A discussion with J.M.Rutledge, IBM Research (16th July 1968) has convinced me that the version at the bottom of EWD239 - 1 serves a purpose. Without mentioning the quintuple problem one can prove its correctness, provided
1) one gives the properties of the operations and functions defined on the objects "candidate" and "sequence"
2) one gives a sharp definitions of the required behaviour, i.e. the specification. This is very encouraging and this is what I wanted.

To my regret I have to part from this version, because it will lead me into difficulties that I should like to avoid. Let me sketch them.

The next step in my process would be the introduction of a canonical state space in terms of which all operations can be specified, particularly the functions "full" and "sequence extendable with candidate". This I can do. To make a more realistic program, I intend to define functions on this canonical state space in order that I can use it to store "current values" as sketched in EWD239 - 0.

Now one or two problems arise. I have to define a function on sequence and candidate and the result is that I should like to introduce "extend sequence with candidate and set candidate to zero", instead of "extend sequence with candidate; set candidate to zero" for, at the separating semicolon the new function is fairly meaningless. Also: in the statements

"while candidate set to one do take candidate from sequence;
set candidate to one"

we have a similar situation, not so much at the separating semivolon but in the course of the repetition. Some of the new functions are "immaterial for some time" and to impose upon the program the duty to keep these XXXXX tabulated values up to date on such a microscopic level is irrealistic.

The next goal therefore is to give more levels, in such a way that functional relationships that we want to introduce are guaranteed to hold at the semicolons of a given level! (This will be the case anyhow: the relations do not hold at the semicolons of the lavel making the adjustments.)

Rutledge has made a point: the fact that "candidate" is a twovalued variable, the "sequence" has something to do with what they call the free monoid on a binary alphabet can be concluded from the properties stated. Only "full" and "sequence extendable with candidate" pin the program down on the quintuple problem. The moral of this remark is that no flexibility is lost when we introduce the free monoid on this level, if it is only to define the desired properties of the operations.

So here we go again, this time very cautiously, for the time being not bothering ourselves too much about the relation between successive versions.

The crudest version is

version 0: "do all work" .

This, although correct, is hardly helpfull, we cannot do much with it either. Then comes

version 1: "generate all solutions"

suggesting that a number (possibly zero) of "solutions" have to be generated. The number must now be finite.

version 2: "generate all solutions in alphabetical order".

This version tells us more, it tells us that the solutions have an ordering that is called alphabetical and that they have to be generated in this order.

version 3: "generate in alphabetical order all bit sequences that are solutions "

Here we have told much more, we have said that we are looking for bitsequences in terms of which the alphabetical order can be defined. (We can take it for granted that this restricts ourselves implicitly to finite bitsequences.)

We do not know whether there are solutions at all. We do not know a first solution, also we do not recognize a last solution when we encounter it. The structure at the bottom side of EWD238 - 1 is therefore unattractive (containing "transform current sequence to next solution".

We do know, however, a criterion "acceptable" (in our example: containing no quintuples presenting the same pattern) with the following properties

1) the set of acceptable sequences is not empty and finite
2) we know a first member of the set
3) we know a virtual last member of the set (for "virtual" see below)
4) solutions are all acceptable sequences (excluding the virtual one) satisfying a further criterion "full"
5) we can transform an acceptable sequence into the next one (next in alphabetic order).

about virtual: if we are looking for the sequences with a zero leading, the virtual last one would be the first acceptable sequence with a one leading. Also, if the first acceptable sequence is the empty sequence, the virtual last one may again be the empty sequence.

We can now make the following program:

version 4:

```
set sequence to first acceptable one;
repeat if sequence full do print solution;
       transform sequence into next acceptable one
until sequence is (first or) last member
```

Remark: the first member is not subjected to the final test for the last member, therefore this test needs only to distinguish between the last member and the others, the first one excluded! There is no objection to the the first one satisfying the test as well, but also no obligation; therefore "first or" has been put between parentheses.

We also know about the property "acceptable"that

6) no extension of a sequence that is not acceptable will be acceptable.

This property enables us to implement
"trnasform sequence into next acceptable one"
-knowing that its initial value is indeed acceptable- as follows:

section 4.1: transform sequence into next acceptable one:
"extend sequence with zero;
while non acceptable do concrease sequence"

The effect of the operation "concrease" is defined in the following way: if the old sequence does not contain a zero the result is the empty sequence, if the old sequence does contain a zero, the result is a copy of the old one up to and excluding the last zero, extended with a one.

The present state is very encouraging, the important property 6 is only exploited in a next level of detail, it is here that backtracking is described, that the alphabetical order is catered for.

(Few days later.)

I must apologize, for I am afraid that I have gone too fast. I am going to modify version 4 of the previous page. The fact is that I have not expressed a changed interpretation of the current value of sequence, I have not expressed that the acceptable sequences fall into two different classes, those that are solutions and those that are not. I try to remedy this by a new version 4:

set sequence to first acceptable one;
repeat if sequence full do begin accept sequence as solution; print solution end;
transform sequence into next acceptable one
until sequence is (first or) last member

The transition from the old to the new version 4 is not striking, because "accept sequence as solution" is in all probability an empty action. (For the sake of completeness I mention that I have been hesitating between "print solution" and "print sequence"; the statement "accept sequence as solution" implies more or less -is meant to indicate- that the two formultations are equivalent.)

The transition is more marked when I give the new section 4.1 on top of this page. In section 4 the sequence is always "acceptable", in section 4.1, where we exploit the important property 6 of "acceptability" we consider for the first time sequences that are not guaranteed acceptable. I call them "doubtful": a sequence is called doubtful when it is a one-digit extension of an acceptable sequence, property 6 tells us that when looking for acceptable sequences, we can confine our attention (the machines attention!) to doubtful sequences.

With this definition of doubtful, section 4.1, new version, would be something like

[illegible]
"transform acceptable sequence into next doubtful one;
while doubtful sequence non acceptable do
transform doubtful sequence into next doubtful one;
accept sequence as acceptable"

The purpose of this new version is twofold. On the one hand it expresses that only doubtful sequences are subjected to the test "acceptable", on the other handit allows us to introduce different representations for doubtful and acceptable sequences. The rules for doubtful sequences hold "at the semicolons" of this level. The first statement makes from an [illegible] acceptable sequence a doubtful one, the last one makes from a doubtful sequence an acceptable one. This allows us to introduce for acceptable sequences a representation unfit for arbitrary doubtful ones. I shall return to this later.

Now I think that the time has come (experience makes me expect that I shall regret it within a few pages!) to introduce what I have called "the canonical state space" i.e. the variables that I need to pin the program down on the specific problem, in this case the quintuple problem.

We introduce the integer k, satisfying $0 \leq k$
and the array d, to represent the value of

sequence: "0 0 0 0 d[0] ... d[k]" .

Putting the zeros also in the array d and using the fact that k will satisfy $k \leq 32$ we can declare
array d[-4: 32].

In terms of these variables we can now give the bodies of the instructions used in the middle of EWD239 - 5

set sequence to first acceptable one:
"d[-4]:= d[-3]:= d[-2]:= d[-1]:= d[0]:= 0; k:= 0"

sequence full:
"k = 31"

accept sequence as solution:
" " (i.e. empty statement)

print solution:
"print the 32 leading elements of the array d" (I refuse here to go into more detail)

transform sequence into next acceptable one , see below

sequence if (first or) last member:
"d[0] = 1" (only satisfied by the last member) or
"k = 0" (satisfied by first and last member).

For "transform sequence into next acceptable one" we have a more detailed version on the bottom of page EWD239 - 5, in which the criterion "acceptable"is mentioned. In order to define it, we define upon the sequence k+1 function values
h(i) ="binary value of bit sequence d[i-4]...d[i]" for $0 \leq i \leq k$
and a sequence is called "acceptable" if all the k+1 values h(i) are different (this expresses the requirement that no two different quintuples present the same bit pattern).

At the semicolons of the detailing of "transform sequence into next acceptable one" the sequence is called "doubtful", i.e. a one digit extension of an acceptable sequence. As only doubtful sequences are subjected to the test acceptable it is sufficient to compare "h(k)" with the "h(i)" for $i < k$.

The next step in detailing now gives:

transform acceptable sequence into next doubtful one:
"k:= k + 1; d[k]:= 0"

doubtful sequence <u>non</u> acceptable:
"for all i, $0 < i < k$ holds: $h(i) \neq h(k)$"

[illegible]
transform doubtful sequence into next doubtful one:
"while d[k] = 1 do k:= k - 1; d[k]:= 1"
(As a result of first and last member, the minimum value of k generated by this operation will be = 0; the final assignment replaces a 0 by a 1.)

[illegible]
accept sequence as acceptable:
" " (also empty statement).

Now at last we get to the subject of this report, trading storage against computation time. We have two alternatives, either we compute the functions h(i) when we need them, or we tabulate them. Computing them when we need them presents itself as a further detailing of "doubtful sequence non acceptable", we shall investigate the consequences of tabulating them.

For this purpose we introduce the
array H[0:32] and postulate that
$H[i] = h(i)$ will hold for $0 \leq i \leq k$
but we have to state on which levels this will hold. It will certainly hold for the [illegible] outer level (the one of the acceptable sequences, middle of page EWD239 - 5) I decide that it will also hold for the level of the doubtful sequences.

It implies an addition to a number of bodies
set sequence to first acceptable one:
is extended with "H[0]:= 0"

transform acceptable sequence into next doubtful one:
is extended with "H[k]:= (2 * H[k-1]) mod 32"
(at this moment "k > 1" is guaranteed to hold)

transform doubtful sequence into next doubtful one:
is extended with "H[k]:= H[k] + 1"

In the above three extensions the particular choice for the quintuple pattern characterizing function h(i) has found its deposit.

In terms of the tabulated values H[i] the analysis for "doubtful sequence non acceptable" would still require scanning. We can repeat the trick by tabulating how often a value occurs in the sequence H[0]....H[k].

This now can be done in two ways. We can introduce the
integer array times[0:31] such that for all j
times[j] = the number of times that the value j occurs among H[0]...H[k]

Let us make this to hold for both acceptable and doubtful sequences. The function is defined on H and k, modifications are to be expected by the three extensions just given and by modifications of k.

The extension of "set sequence to first acceptable one" is extended with
"times[0]:= 1; other values of times set to 0"

The extension of "trnaform acceptable sequence into next doubtful one" is extended with
"times[H[k]]:= times[H[k]] + 1"

And the new version of "transform doubtful sequence into next doubtful one" becomes

"while d[k] do begin times[H[k]]:= times[H[k]] - 1; k:= k - 1 end;
d[k]:= 1; times[H[k]]:= times[H[k]] - 1; H[k]:= H[k] + 1; times[H[k]]:=
times[H[k]] + 1"

The test "doubtful sequence non acceptable" is now simply

"times[H[k]] > 1".

The introduction of the array times is somewhat awkward. One of the nasty things is that its values are restricted to 0 and 1 and only "times[H[k]]" can ever get the value = 2. And this is to be modified immediately.
Additions and subtractions from these elements are usually just setting to 1 or 0.

Here we can use an alternative solution, we do not introduce the array times, but the boolean array IN.

In the outer level (acceptable sequences)

IN[j] means "the value j occurs (once!) in the sequence H[0]...H[k]"

In the inner level (doubtful sequences)

IN[j] means "the value j occurs (once!) in the sequence H[0]...H[k-1]" .

We now review all instructions.

set sequence to first acceptable one:
"d[-4]:= d[-3]:= d[-2]:= d[-1]:= d[0]:= 0; k:= 0;
H[0]:= 0; IN[0]:= true; remaining elements of IN become false"

sequence full:
"k = 31" (unchanged)

accept sequence as solution:
" " (unchanged)

print solution (unchanged)

sequence is (first or) last member (unchanged)

transfor acceptable sequence into next doubtful one
"k:= k + 1; d[k]:= 0; H[k]:= (2 * H[k-1]) mod 32"
(As the sequence changes from acceptable to doubtful, IN is left unchanged)

doubtful sequence non acceptable:
"IN[H[k]]"

transform doubtful sequence into next doubtful one:
"while d[k] = 1 do begin k:= k - 1; IN[H[k]]:= false end;
d[k]:= 1; H[k]:= H[k] + 1"
(Compare this with the top lines of this page!)

And finally "accept sequence as acceptable" that was empty until now, gets the form IN[H[k]]:= true, on account of the changing definition.

By now I trust that my reader will have lost the various versions and his way through them; this was somewhat intentional. It shows the need for a clerical aid, a hierarchical assembler or possibly .... computer assistance!