Some Lessons Drawn from the History of the Binary Search Algorithm

R. Lesuisse
Institut d'Informatique, Rue Grandgagnage, 21, B–5000 Namur, Belgium

The paper's aim is to study the behaviour of 26 programmers who designed a binary search algorithm; for each of them, we try to point out the correct or erroneous reasoning, that yielded the algorithm. To do this, we first analyse 4 typical correct algorithms in an informal and intuitive way, discussing the probable reasons why the programmers developed the algorithms as they did; these are, namely: (1) a uniform binary search algorithm on a monotonically increasing sequence of T terms, $T = 2^n - 1, n \geq 1$; (2) a uniform binary search algorithm on a monotonically increasing sequence of T terms; (3) a binary search algorithm with 'two-way tests' on a monotonically increasing sequence of T terms, $T \geq 1$; (4) a general binary search algorithm on a monotonically increasing sequence of T terms. Then, the important errors found in the 26 published algorithms are pointed out, with an attempt at discussing why these errors were made. Three lessons can be drawn from the history of the binary search algorithm, which we believe are applicable to many other algorithms, namely (1) there is a continuous progression by programmers towards a highest possible degree of generality; (2) programs built with optimization concerns in mind have no better mean execution time than the most general ones; (3) the distribution of errors among families of algorithms is not uniform.

1. INTRODUCTION

The binary search algorithm is probably one of the most classical algorithms published in the literature. Therefore we are able to reconstruct its history,[1] to study the errors made by some designers and to try to draw some general lessons about the behaviour of programmers.

This is the scope of our paper. In section 2, we briefly give the algorithm's specifications and explain the few basic concepts underlying it; in section 3, we analyse 4 typical binary search algorithms; in section 4, the important errors found in 4 published algorithms are pointed out and discussed, and in section 5, we draw some general lessons from the history of the binary search algorithm.

2. SPECIFICATION OF THE BINARY SEARCH ALGORITHM

2.1 Search problem specification

We restrict ourselves to the search for a number x within a sequence S. This problem can be formulated as follows:

Input	Given · a sequence S of T terms · a number x
Function	find a search method, i.e. a set of rules by which we can determine . . .
Performance	. . . as safely and quickly as possible . . .
Output	. . . that —either $x \in S$; in this case, the message '$x \in S$' is generated; —or $x \notin S$; in this case, the message '$x \notin S$' is generated.

2.2 Basic idea leading to the solution

To solve this problem, one approach is to assume that the sequence S is ordered (for instance, S is a monotonically increasing sequence) and then to proceed as follows:

(1) Compare x with $S[K]$, $(1 \leq K \leq T)$
  · If $x = S[K]$, the search is successful and the message '$x \in S$' is generated;
  · If $x < S[K]$, the search proceeds within the sequence $S[1:K-1]$;
  · If $x > S[K]$, the search proceeds within the sequence $S[K+1:T]$.

(2) If the search sequence is empty, the search is unsuccessful and the message '$x \notin S$' is generated.

Assuming that each of the T terms is equally likely to be the number sought for, it is clear that an efficient search method will halve the search sequence on each iteration step. Thus a search requires about $\log_2 T$ comparisons.

2.3 Underlying concepts

From this rough idea, we can define several useful concepts for any binary search problem:

(1) *search sequence*: a set of consecutive terms of $S[1:T]$

(2) *experience*: the comparison between x and a term of the search sequence;
successful experience, if $x =$ this term
unsuccessful experience, if $x \neq$ this term

(3) *middle of a search sequence*: the term $S[K](A \leq K \leq B)$ is said to be the middle of the search sequence $S[A:B]$ if $K = \lfloor (B - A)/2 \rfloor$

(4) *binary search tree*: any binary search method within a sequence $S[1:T]$ may be seen as a binary decision tree in which the nodes are labelled with the numbers 1 to T (therefore, a binary tree is a possible *representation* of any binary search method). For

CCC–0010–4620/83/0026–0154 $05.00

instance, we can construct the binary search tree corresponding to a search within a sequence $S[1:7]$ (see Fig. 1).

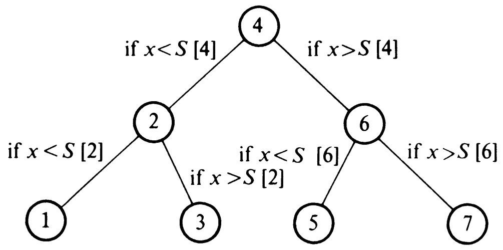


Figure 1

2.4 General binary search problem specification

The binary search problem can be formulated as follows:

Input	Given · a monotonically increasing sequence $S[1:T]$ · a number x
Function	search for x within the sequence $S[1:T]$.
Performance	it is well known that the complexity of any search, be it successful or unsuccessful is in logarithmic proportion to the sequence size.
Output	The result will be —either the message '$x \in S$' —or the message '$x \notin S$'.

3. HISTORY OF THE BINARY SEARCH ALGORITHM

Our analysis of 26 published binary search algorithms led us to classify these into 4 families, which we will now describe.

For each family, we will build a typical algorithm; the families are described, as much as possible, according to the chronological order of publication.

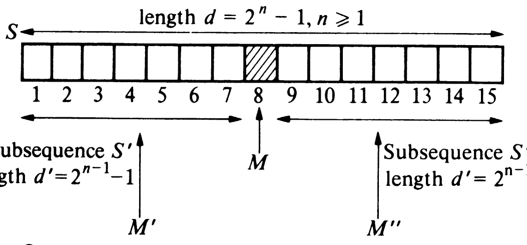
3.1 Family 1: uniform binary search algorithms on a monotonically increasing sequence of T terms, $T = 2^n - 1, n \geq 1$

3.1.1 Chronology and references. First publication: D. D. MacCracken, 1957.[2] For other members of this class, see among others, Cherton,[3] Donovan.[4]

3.1.2 Design of a typical algorithm

3.1.2.1 *Basic idea.* This family of algorithms is based on the following convention: 'any search sequence has $2^n - 1$ terms, $n \geq 1$'. This property allows us:

(1) To find a middle element $S[M]$ which divides the search sequence into 2 subsequences of *equal length* d' (see Fig. 2)



length $d = 2^n - 1, n \geq 1$
S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Subsequence S'
length $d' = 2^{n-1} - 1$
M
Subsequence S''
length $d' = 2^{n-1} - 1$
M'
M''

Figure 2

(2) To compute the middle of a subsequence from the middle and the length d of the preceding one, as follows

$$M' = M - \left\lceil \frac{\lfloor d/2 \rfloor}{2} \right\rceil = M - \left\lceil \frac{\lfloor (2^n - 1)/2 \rfloor}{2} \right\rceil$$

$$M'' = M + \left\lceil \frac{\lfloor d/2 \rfloor}{2} \right\rceil = M + \left\lceil \frac{\lfloor (2^n - 1)/2 \rfloor}{2} \right\rceil$$

Therefore, in order to search for x within the search sequence, we proceed as follows:

(1) **If** $d < 1$
then the search sequence is empty and '$x \notin S$';

(2) Compare x with the middle $S[M]$ of the search sequence;

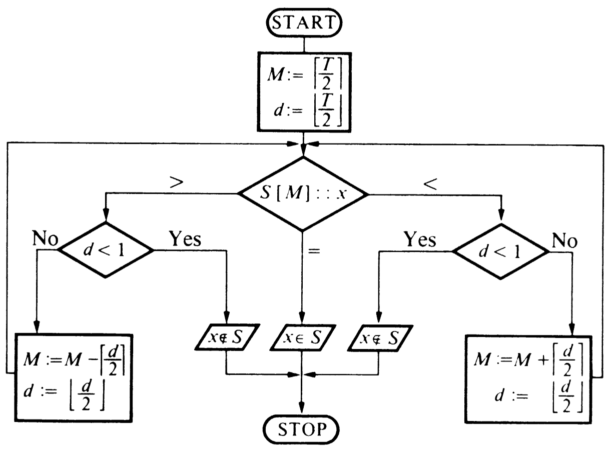
(3) · **If** $x = S[M]$
then '$x \in S$' and the search is terminated;
· **If** $x < S[M]$
then · compute $M' = M - \left\lceil \frac{\lfloor d/2 \rfloor}{2} \right\rceil$
· and recursively apply the same process to the new search sequence (*by induction on n*)

· **If** $x > S[M]$
then · compute $M'' = M + \left\lceil \frac{\lfloor d/2 \rfloor}{2} \right\rceil$
· and recursively apply the same process to the new search sequence.

3.1.2.2 *Specification of the algorithm.* The specifications of this binary search algorithm are the same as the general ones (see section 2.4), with the restriction that $T = 2^n - 1, n \geq 1$

3.1.2.3 *Representation* (see Fig. 3). The obvious representation conventions are:

(1) The search sequence $S[1:T]$ is represented by the integer array $S[1:T]$;
(2) The position of an examined term is represented by the variable M;
(3) The number x is represented by the integer variable x;
(4) The number d is represented by the integer variable d.



START
$M := \left\lceil \frac{T}{2} \right\rceil$
$d := \left\lfloor \frac{T}{2} \right\rfloor$
$S[M] :: x$
>
<
=
No
$d < 1$
Yes
Yes
$d < 1$
No
$x \notin S$
$x \in S$
$x \notin S$
$M := M - \left\lceil \frac{d}{2} \right\rceil$
$d := \left\lfloor \frac{d}{2} \right\rfloor$
$M := M + \left\lceil \frac{d}{2} \right\rceil$
$d := \left\lfloor \frac{d}{2} \right\rfloor$
STOP

Figure 3

3.2 Family 2: uniform binary search algorithm on a monotonically increasing sequence of T terms

3.2.1 Chronology and references. First publication: I. Florès, 1965.[5] For other members of this class, see, among others, Bonnin,[6] Chandra.[7]

3.2.2 Design of a typical algorithm

3.2.2.1 *Basic idea.* We try to use the preceding method without the restriction that $T = 2^n - 1, n \geq 1$.

To achieve this, we note that, in the preceding algorithm, any search sequence has $2^n - 1$ terms, $n \geq 1$, i.e. an *odd* number of terms.

Therefore, in order to generalize the preceding method, we have two problems to solve:

(1) What is to be done when a search subsequence has an even number of terms?
(2) What is to be done when the initial search sequence has an even number of terms? (with 0 considered as an even number).

The first problem is solved as follows: suppose that after a certain number of experiences,

(1) we have a search sequence S whose length d is odd; obviously, the middle element $S[M]$ divides this search sequence into 2 subsequences (S' and S'') of equal length $d' = d''$ (see Fig. 4).

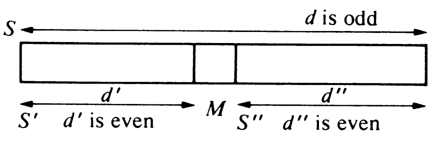


Figure 4

(2) the length $d' = d''$ of the two subsequences is even.

Then we proceed as follows:

(3) **if** $x < S[M]$
then · add one term to the right of the subsequence S'; (see Fig. 5).

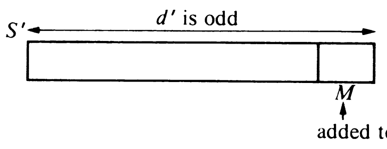


Figure 5

· **now** d' is *odd* and we may apply the preceding method to the search sequence S';

(4) **if** $x > S[M]$
then · add one term to the *left* of the subsequence S'' (see Fig. 6);

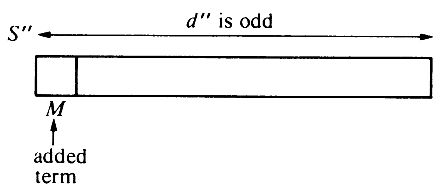


Figure 6

· now d'' is *odd* and we may apply the preceding method to the search sequence S'';

In order to solve the problem of an *initial* search sequence S with an even number of terms, we add an artificial term $S[0]$ whose value is '$-\infty$' to the *left* of S (see Fig. 7) and we may apply the preceding method to the *initial* search sequence.

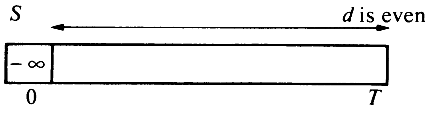


Figure 7

3.2.2.2 *Specification.* The specifications of the uniform binary search algorithm are the same as the general ones (see section 2.4).

3.2.2.3 *Representation.* According to the representation conventions described in section 3.1.2.3., the above idea leads to the same flowchart as the one described in Fig. 3. Note that 2 DIFFERENT IDEAS may lead to the SAME FLOWCHART.

3.3 Family 3: binary search algorithm with 'two-way tests' on a monotonically increasing sequence of T terms, $T \geq 1$

3.3.1 Chronology and references. First publication: H. Bottenbruch, 1962.[8] For other members of this family, see, among others, Bartee,[9] Baudoin and Meyer,[10] Baufay,[11] Dijkstra,[12] Gear,[13] Lecharlier,[14] Weinberg *et al.*[15]

3.3.2 Design of a typical algorithm

3.3.2.1 *Basic idea.* We have the following basic conventions:

(1) The search sequence contains at least 1 term.
(2) Before any experience, the positions of the first (L) and last (U) terms in the search sequence are known *explicitly*; any search sequence can therefore be written as
$$S[L:U]$$
(3) The initial search sequence $S[1:T]$ contains every further search sequence; therefore, we have:
$$1 \leq L, \quad U \leq T$$
(4) We avoid the test for equality ($x = S[M]$) until the search sequence $S[L:U]$ contains only 1 term; in this way, the algorithm is implemented with two alternatives instead of three (hence, the name 'two-way' tests), e.g. Fig. 8.

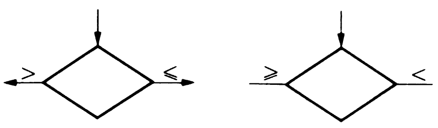


Figure 8

Using these conventions, we solve the search problem by induction on the set $\mathscr{D}[1:T]$ of intervals I such that $I \subseteq [1:T]$; this set is ordered by the set inclusion relation and the smallest elements of ($\mathscr{D}[1:T]$, $\subseteq$) are the intervals of length 1.

Suppose that, after a certain number of experiences, the initial search sequence has been reduced to the search sequence $S[L:U]$, $1 \leq L \leq U \leq T$ (Fig. 9).

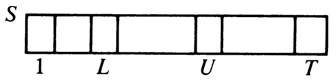


Figure 9

The problem to be solved is then: '$x \in S[L:U]$?'. In order to answer this question, we proceed as follows:

```
(1) If $L = U$
    then · the search sequence has only 1 term;
         · therefore if $x = S[L]$
               then '$x \in S$' and the search is terminated;
               else '$x \notin S$' and the search is terminated;
(2) If $L < U$
    then · compute the position $M$ of the middle of the search sequence
               $M := \left\lfloor \frac{L+U}{2} \right\rfloor$
         · compare $x$ with $S[M]$;
         · if $x \leq S[M]$
           then $x$ has no chance of being in the sequence $S[M+1:U]$; therefore, we recursively apply the same process to the interval $[L:M]$
         · If $x > S[M]$
           then $x$ has no chance of being in the sequence $S[L:M]$; therefore, we recursively apply the same process to the interval $[M+1:U]$
```

In order to solve the initial problem (search for x within $S[1:T]$)

- let $L := 1$, $U := T$;
- we apply the same process to the interval $[L:U]$ and the problem is solved.

3.3.2.2 *Specification.* The specifications of the binary search algorithm with 'two-way tests' are the same as the general ones (see section 2.4), with the restriction that $T \geq 1$.

3.3.2.3 *Representation.* According to the following obvious representation conventions:

- the position L (resp. U) of the first (resp. last) term of a search sequence is represented by the integer variable L (resp. U);
- for the representation of the search sequence S, the position M of an examined term and the number x search for, see section 3.1.2.3.

the above idea leads to the flowchart in Fig. 10.

3.4 Family 4: general binary search algorithm on a monotonically increasing sequence of T terms

3.4.1 Chronology and references. First publication: D. E. Knuth, 1963.[1] For other members of this family, see, among others, Aho *et al.*,[16] Berztiss,[17] Burge,[18] De Angelo and Jorgensen,[19] Gear,[13] Kelly and Mac Gowan,[20] Leeds and Weinberg,[21] Leroy,[22] Price,[23] Stone and Siewiorek,[24] Wirth.[25]

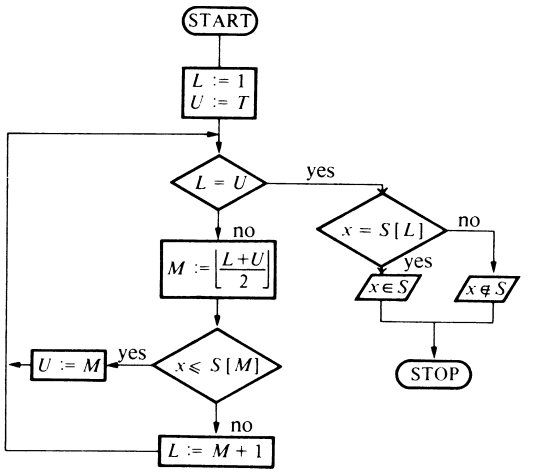


Figure 10

3.4.2 Design of a typical algorithm

3.4.2.1 *Basic idea.* We have the following basic conventions:

(1) the search sequence may be empty;
(2) the two other basic conventions are the same as conventions (2) and (3) of the preceding algorithm see section 3.3.2.

Using these conventions, we solve the search problem by induction on the set $\mathscr{E}[1:T]$ of intervals I such that $I \subseteq [1:T]$; this set is ordered by the set inclusion relation and the smallest elements of ($\mathscr{E}[1:T]$, $\subseteq$) are the intervals of length 0.

Suppose that after a certain number of experiences, the initial search sequence $S[1:T]$ has been reduced to the search sequence $S[L:U]$, $1 \leq L$ and $U \leq T$ (see Fig. 11).

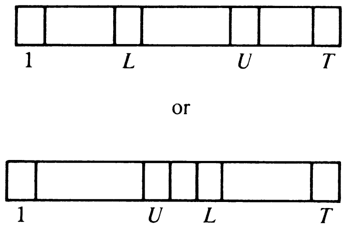


Figure 11

The problem to be solved is then the following one: '$x \in S[L:U]$'?

In order to answer this question, we proceed now as follows:

```
(1) If $L > U$
    then $x \notin S$; obviously the search sequence is empty;
(2) If $L < U$
    then · compute the position $M$ of the middle of the search sequence:
               $M := \left\lfloor \frac{L+U}{2} \right\rfloor$ or $M := \left\lceil \frac{L+U}{2} \right\rceil$
```

· compare x with $S[M]$
· **if** $x = S[M]$
  then $x \in S$ and the search is terminated;
· **if** $x < S[M]$
  then x has no chance of being in the sequence $S[M:U]$; we therefore apply the method described above to solve the problem '$x \in S[L:M-1]$?'
· **if** $x > S[M]$
  then x has no chance of being in the sequence $S[L:M]$; we therefore apply the method described above to solve the problem '$x \in S[M+1:U]$?'

In order to solve the initial problem (search for x within $S[1:T]$), we set L to 1 and U to T and apply the method to the interval $[L:U]$; the problem is therefore solved.

3.4.2.2 *Specification.* The specifications of the general binary search algorithm have already been described, see section 2.4.

3.4.2.3 *Representation.* According to the representation conventions described in section 3.3.2.3., the above idea leads us to the flowchart of Fig. 12.

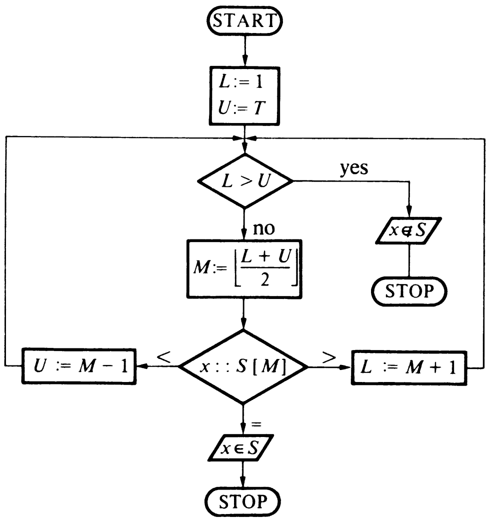


Figure 12

4. ERROR ANALYSIS

Four among the twenty-six examined algorithms contain errors, namely the algorithms of Donovan,[4] Flores,[5] Bonnin[6] and De Angelo and Jorgensen.[19]

We will first briefly describe the errors and then comment on their causes.

4.1 Incorrect algorithms

4.1.1 Donovan's algorithm. Donovan's program belongs to family 1. The following error has been found in the Assembler program.

```
L      5, LAST        Set table size (2^N * 14 bytes)
SRL    5, 1           Divide by 2 by shifting
LR     6, 5           Copy into register 6
```

```
LOOP       SRL   6, 1              Divide table size in half again
           LA    4, SYMTBL(5)      Set address of table entry
           CLC   0 (8, 4) SYMBOLCompare with symbol
           BE    FOUND             Symbols match, entry found
           BH    TOOHIGH           SYMTBL entry > symbol
TOOLOW     AR    5, 6              Move higher in table
           B     TESTEND
TOOHIGH    SR    5, 6              Move lower in table
TESTEND    LTR   6, 6              Test if remaining size is 0
           BNZ   LOOP              No, look at next entry
NOTFOUND (Symbol not found)
           .
           .
           .
FOUND      (Symbol found)
```

We recall the semantics of the LTR Assembler instruction: 'LTR R1, R2:LTR moves (R2) to R1. In doing so, it sets the condition code to indicate whether the number moved is positive, negative or zero. If R1 and R2 are the same register, LTR simply sets the condition code to indicate whether (R1) is positive, negative or zero'.

The shaded instruction is therefore erroneous; the search should be stopped as soon as register 6 has a value less than 14, in this case, value 7 by application of the binary shift instruction instead of the value 0 as assumed by Donovan. As soon as the register 6 contains a value less than 14, it means that the table is empty, since Donovan assumed that all elements are 14 bytes long (see comment of the first instruction of Donovan's algorithm above).

4.1.2 Flores algorithm. Flores' algorithm belongs to family 2; however, his conventions are somewhat different.

Flores assumes *implicitly* that any search sequence must have an *odd* number of terms.

Suppose that the search sequence has 13 terms ($S[1:13]$). In order to search for x within such a sequence, Flores proceeds as follows:

(1) x is compared with the middle, $S[7]$, of the search sequence;
(2) · **if** $x = S[7]$
  then '$x \in S$' and the search is terminated;
  · **if** $x < S[7]$
  then the new search sequence is $S[1:6]$; this search sequence has no odd number of terms. Therefore, Flores adds one term to its *left* (see Fig. 13) (the typical algorithm of family 2 adds this term to its *right*); the middle of this new search sequence is $S[3]$
  · The case $x > S[7]$ is symmetrical.

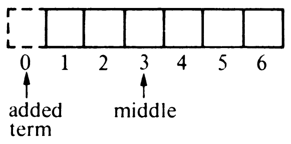


Figure 13

So, we note that the position of the middle of the new search sequence is obtained by adding or subtracting some quantity d' to (from) the position of the middle of the old search sequence; this quantity d' may be computed from the length d of the old search sequence as follows:

$$d' = \left\lceil \frac{\lfloor d/2 \rfloor + 1}{2} \right\rceil$$

Using this idea, the flowchart of Flores' algorithm looks like Fig. 14.

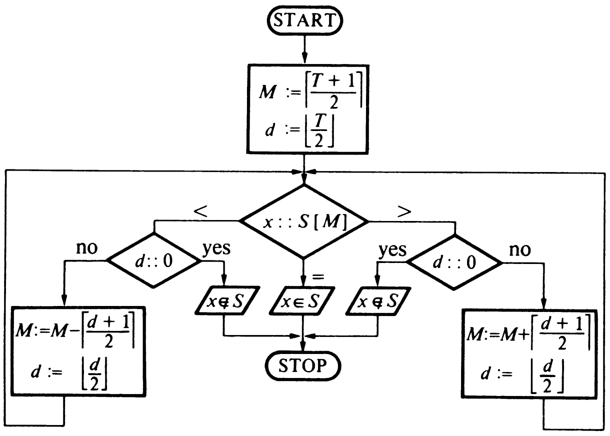

START
$M := \lceil \frac{T+1}{2} \rceil$
$d := \lfloor \frac{T}{2} \rfloor$
$x :: S[M]$
<
>
no
$d :: 0$
yes
yes
$d :: 0$
no
=
$x \notin S$
$x \in S$
$x \notin S$
$M := M - \lceil \frac{d+1}{2} \rceil$
$d := \lfloor \frac{d}{2} \rfloor$
STOP
$M := M + \lceil \frac{d+1}{2} \rceil$
$d := \lfloor \frac{d}{2} \rfloor$

Figure 14

In this algorithm, the search for a number x may fall *outside* the initial search sequence.

In order to visualize this, we build the binary tree corresponding to this algorithm, for an initial search sequence of 8 terms, i.e. $S[1:8]$ (Fig. 15).

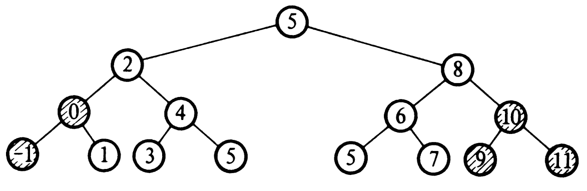


Figure 15

We observe that the shaded nodes are 'impossible', since the initial search sequence $S[1:8]$ does not contain the terms $S[-1]$, $S[0]$, $S[9]$, $S[10]$, $S[11]$; thus, the search falls outside the initial search sequence.

In order to smooth away these difficulties, Flores makes two proposals:

(i) the initial search sequence contains a term $S[0]$, with conventional value '$-\infty$'

(ii) ... and a term $S[10]$ with conventional value '$+\infty$'

These naïve proposals work only for the case $T = 8$ but are not adequate in the general case.

4.1.3 Bonnin's algorithm. Bonnin's algorithm belongs to family 2. He arbitrarily assumes that the initial search sequence S contains 500 terms.

His flowchart looks like Fig. 16.

Owing to the absence of good documentation, the author's reasoning is hard to reconstruct.

So, in order to point out some of the errors, we build the binary tree corresponding to his algorithm for an initial search sequence $S[1:8]$ (Fig. 17).

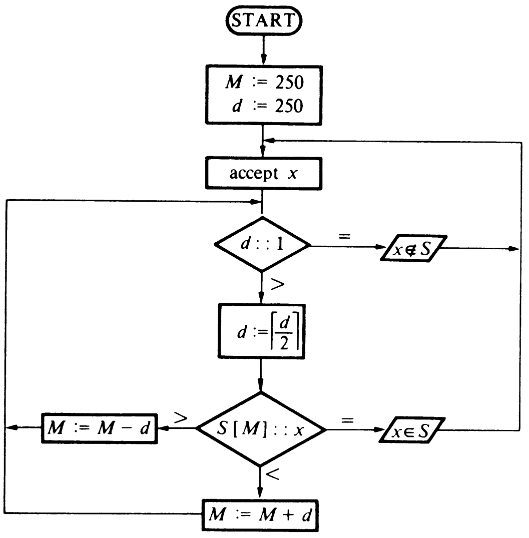


Figure 16

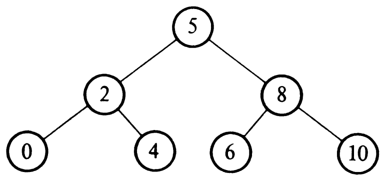


Figure 17

Note that:

(1) The algorithm never terminates: no action is provided when there are no more x to be input (misuse of the COBOL verb 'accept').

(2) The search for a number x may fall outside the search sequence $S[1:T]$.

(3) The algorithm takes only search sequences with at least 2 terms into account; so, if we consider the above binary tree (Fig. 17), we note that the terms $S[1]$, $S[3]$, $S[5]$, $S[7]$ could not be tested; therefore a search may be unsuccessful, even though x effectively belongs to the search sequence.

4.1.4 De Angelo's algorithm. De Angelo's algorithm belongs to family 4; this algorithm makes successful search; it looks like Fig. 18.

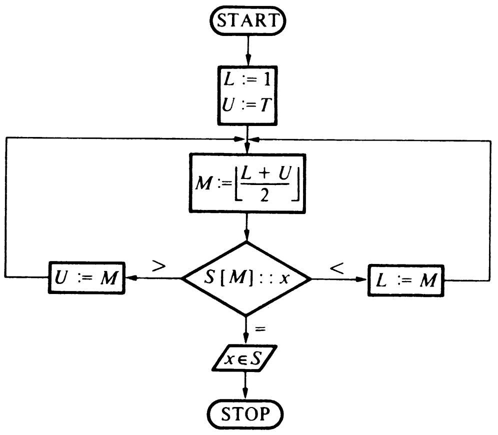


Figure 18

In order to point out the error, let us build the binary tree corresponding to this algorithm, assuming again that the initial search sequence has 8 terms: the values of the bounds L and U of the search sequence (before the computation of the middle M) have been attached to the edges (Fig. 19).

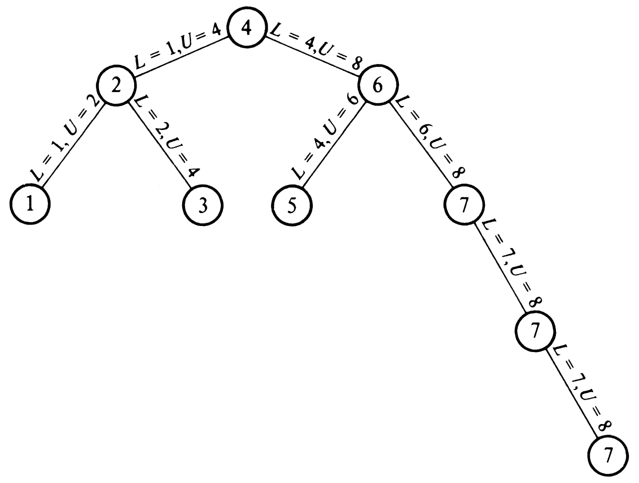


Figure 19

The search thus never gives the expected result, when $x = S[8]$, and more generally when $x = S[T]$, and the algorithm may not terminate.

The following correct initialization for L and U seems to be necessary:

$$L := 0; \qquad U := T + 1$$

4.3 Discussion

Among the 6 errors detected, 4 are design errors (i.e. errors contained within the solution idea itself) while 2 are coding errors (i.e. errors made during the representation of this idea by means of some programming language).

4.3.1 Design errors. Among the design errors, we have:

(a) two undefined searches[5,6] falling outside the initial sequence;
(b) one non-terminating loop[19] in the case when the searched element is the last element of the initial sequence;
(c) one unsuccessful search[6] even if the searched element belongs to the original sequence.

All these errors have a common point; they never appear at the first experience, but rather towards the end of the search.

This suggests that their designers have built the algorithm by *mental simulation of its execution*, from the first comparison to the last one.

This is the way many inexperienced programmers work. Typically, they will write an algorithm like Fig. 20.

If we build the corresponding binary tree, for an initial search sequence with 14 terms (Fig. 21), we observe that only the left path of this binary tree corresponds to a binary search. This reflects the following kind of mental simulation:

(i) first, solve the particular case where the search would be unsuccessful because $x < S[1]$;

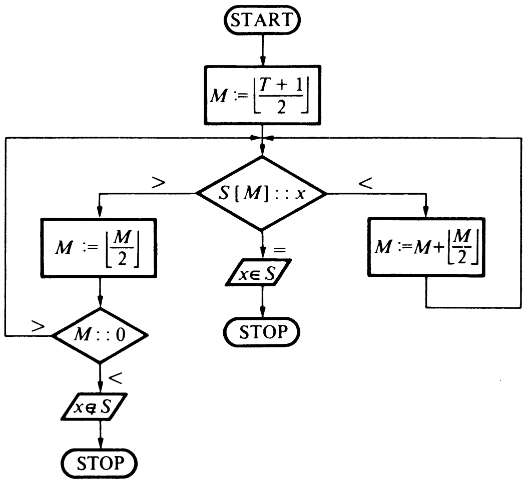

START
$M := \left\lfloor \frac{T+1}{2} \right\rfloor$
$S[M] :: x$
>
<
=
$M := \left\lfloor \frac{M}{2} \right\rfloor$
$M := M + \left\lfloor \frac{M}{2} \right\rfloor$
$x \in S$
STOP
$M :: 0$
>
<
$x \notin S$
STOP

Figure 20

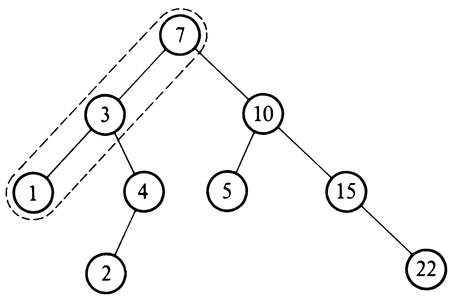


Figure 21

(ii) then, try to adjust this solution to a more general case.

Design by experimentation on particular cases is of course very often subject to failure; this is particularly true for problems having natural solutions based on inductive arguments.

4.3.2 Coding errors. Two coding errors[4,6] were detected among the 26 algorithms considered. Both errors were made in the description of actions to be executed when certain exceptional events occur (cf. the misuse of the Assembler instruction LTR and of the COBOL verb ACCEPT).

This kind of error depends heavily on programming language facilities for explicit exception handling.

5. SOME LESSONS TO BE DRAWN FROM THE HISTORY OF THE BINARY SEARCH ALGORITHM

We feel that the following discussion might be applicable to many other algorithm designs.

5.1 The history of binary search algorithms shows a continuous progression by programmers towards a higher degree of generality

We may observe a difference between the behaviour of the programmers who built typical algorithms 1, 2 and 3, and the behaviour of the programmer who built typical algorithm 4.

5.1.1 Behavioural analysis of the programmers who built typical algorithms 1, 2 and 3. In order to solve the problem, the designers of typical algorithms 1, 2 and 3

(a) proceeded by enumerating cases and first solving a particular case;
(b) imposed certain representation constraints directly, in order to optimize the execution time of their algorithm.

5.1.1.1 *Behavioural analysis of the programmer who built typical algorithm* 1.

(a) This programmer makes a distinction between the case where the search sequence contains $2^n - 1$ terms, $n \geq 1$ (this particular case being easy to solve) and the case where the search sequence does not contain $2^n - 1$ terms, $n \geq 1$; the latter case is solved by reducing the problem to the preceding one, e.g. by adding artificial terms on the right of the search sequence.
(b) Furthermore, the algorithm's representation is immediately taken into account; namely, when the search sequence contains exactly $2^n - 1$ terms, $n \geq 1$, it is possible to use binary shift instructions instead of arithmetic instructions such as division (see Gear's remark[13]); in doing so, the programmer believes his program has the best possible execution time.

5.1.1.2 *Behavioural analysis of the programmer who built typical algorithm* 2.

(a) This programmer makes a distinction between the case where the search sequence contains an *odd* number of terms and the case where the search sequence contains an *even* number of terms; again, the latter case is solved by reducing the problem to the preceding one, e.g. by adding one term on the left (right) of the search sequence.
(b) Binary shift instructions are used again.

5.1.1.3 *Behavioural analysis of the programmer who built typical algorithm* 3.

(a) This programmer makes a distinction between the case where the search sequence contains only 1 term (the problem being then easy to solve) and the case where the search sequence contains more than 1 term; again, the latter case is solved by reducing the problem to the preceding one.
(b) Representation details are considered from the beginning as well: the algorithm is *explicitly* designed to run on computers with two-way tests (see Gear's remark[13] and Baudoin and Meyer's remark[10]).

To summarize, the designers of typical algorithms 1, 2 and 3

(i) first found the solution to a particular case (= a less general problem than the stated one);
(ii) then found out how to generalize this particular solution to solve the initial problem.

5.1.2 Behavioural analysis of the programmer who built typical algorithm 4. The programmer of typical algorithm 4 works in the opposite way; we note that

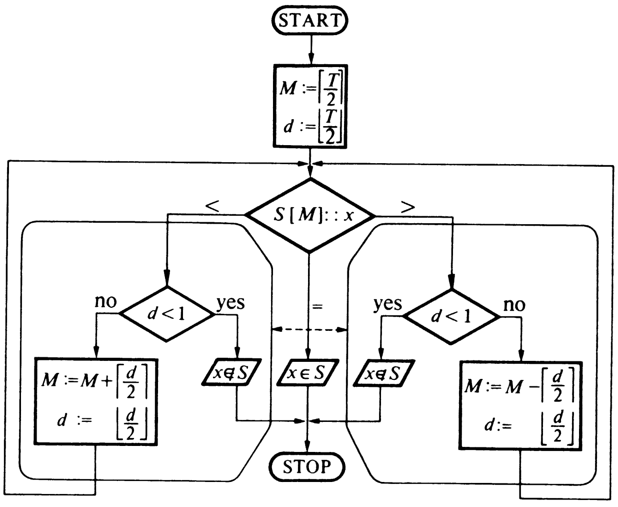
(i) the problem is first solved for a search sequence $S[L:U]$ where L and U have arbitrary values $(1 \leq L, U \leq T)$;
(ii) the initial problem is then shown to be a particular case of the preceding one ($L = 1$, $U = T$);
(iii) no representation details nor performance concerns, other than logarithmic complexity, are considered *a priori*.

Thus, the programmer of typical algorithm 4

(i) first found a solution to a more general problem;
(ii) then showed that the initial problem is a particular case of the general problem which had been solved.

5.1.3 Some symptoms that show that an algorithm is not a general one. The poor lonesome programmer does not know anything about the history of the algorithm he is building. Are there any symptoms to show that his algorithm is not a general one? In the case of the binary search algorithm, there are at least 3 symptoms which are probably more general in application.

(1) *The programmer has arbitrarily restricted the original problem*
For instance,
(a) the search sequence $S[1:T]$ is assumed
(i) to contain at least 1 term (algorithms of families 1, 3);
(ii) to always contain $2^n - 1$ terms, $n \geq 1$ (algorithms of family 1);
(b) an artificial term is assumed to be added on the left (right) of the search sequence (cf. the artificial term $S[0]$ in the algorithm 2);
(c) the number x searched for is assumed to have a restricted value, e.g. Dijkstra[12] arbitrarily assumes that
$$S[0] \leq x < S[T]$$
in order to make his correctness proof easier.
(2) *The algorithm is badly-structured, in that it contains almost-identical parts*
For instance, the typical algorithm 2 contains 2 almost-identical parts (see Fig. 22).


START
$M := \left\lceil \frac{T}{2} \right\rceil$
$d := \left\lfloor \frac{T}{2} \right\rfloor$
$S[M] :: x$
<
>
=
no
$d < 1$
yes
yes
$d < 1$
no
$M := M + \left\lceil \frac{d}{2} \right\rceil$
$d := \left\lfloor \frac{d}{2} \right\rfloor$
$x \notin S$
$x \in S$
$x \notin S$
$M := M - \left\lceil \frac{d}{2} \right\rceil$
$d := \left\lfloor \frac{d}{2} \right\rfloor$
STOP
Figure 22

The typical algorithm 3 contains almost-identical parts as well (Fig. 23).

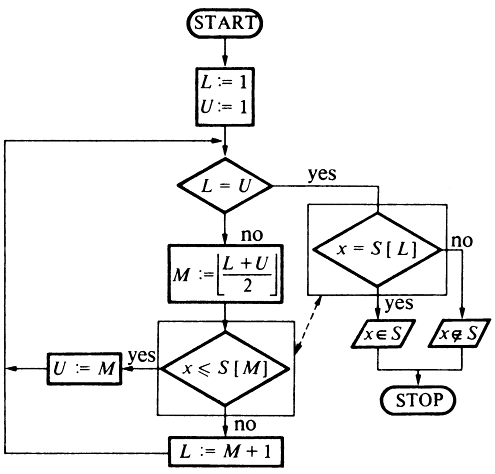


Figure 23

This redundancy suggests that the programmer reasoned case by case.

(3) *The algorithm makes redundant tests*
For instance, assuming the initial search sequence has 11 terms, we obtain the binary tree of Fig. 24 for typical algorithm 2.

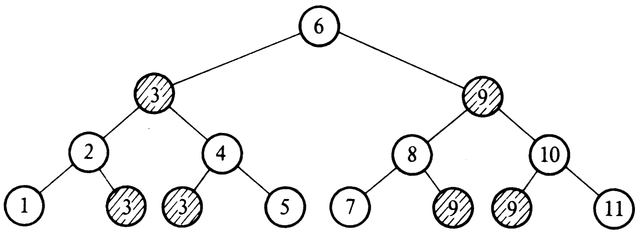


Figure 24

Thus, as shown by the shaded (repeated) nodes, the algorithm performs more tests than required.

5.2 The history of the binary search algorithm suggests that programs designed with *a priori* micro-optimization concerns in mind do not yield better performances

5.2.1 Standard opinion. Many programmers believe that the real performance of an algorithm is estimated by measuring its mean execution time; from this point of view typical program 1 which assumes that the search sequences have $2^n - 1$ terms, $n \geq 1$ would be the best one; in any case, it would seem that this program has better performances than typical program 4 which is the most general one.

5.2.2 Performance assessment. We computed the mean execution time of these two programs on a SIEMENS 4004/151 computer, using the mean execution time of each instruction given by the constructor and we obtained the results in Tables 1 and 2.

5.2.3 Conclusion. The results in Tables 1 and 2 show that the difference between the two mean execution times is negligible; thus, contrary to general opinion, the most general program has good performances as well.

Table 1. Successful search

T	Mean execution time (μs) of typical program 1 [1]	Mean execution time (μs) of typical program 4 [2]	Δ (μs) [1] − [2]
23	73.46	74.03	−0.57
191	126.37	127.08	−0.71

Table 2. Unsuccessful search

T	Mean execution time (μs) of typical program 1 [1]	Mean execution time (μs) of typical program 4 [2]	Δ (μs) [1] − [2]
23	97.815	98.80	−0.985
101	152.08	154.88	−2.80

5.3 The analysis of errors contained within some of the 26 algorithms considered leads us to make some further observations

(1) *Four algorithms do not terminate* (or may not terminate), i.e. 1 algorithm in every 5 considered.

Contrary to general opinion, it appears that toy-programs designed under ideal conditions, i.e. without time constraints and for publication (therefore submitted to the refereeing process) may contain serious errors.

From the above observation, what might we infer for the case of 'everyday' programs, i.e. large, anonymous programs that have been designed to fit hard, ill-estimated deadlines?

Our study thus strongly supports the opinion that programmers underestimate the difficulty of designing a program.

(2) *The distribution of errors among families is not uniform.* As a matter of fact,
 (i) within family 1, one algorithm[4] out of the three considered is incorrect;
 (ii) within family 2, two algorithms[5,6] out of the three considered contain serious errors;
 (iii) within family 3, the eight algorithms considered are correct;
 (iv) within family 4, one algorithm[19] out of the twelve considered is incorrect.

We note that 2 (out of 3) programs of family 2 are erroneous. Knuth[1] correctly pointed out that the design of an algorithm from the basic idea of family 2 is critical:

'it is possible to do this, but only if extreme care is paid to the details; simpler approaches are doomed to failure' (p. 441).

(3) *The most critical errors* have been found in *programs designed by mental simulation of execution* rather than by inductive arguments.

6. SUMMARY

Our aim was to study the behaviour of programmers designing binary search algorithms.

To this end, we analysed 4 typical correct binary search algorithms; we showed in particular that

(1) The most general program was designed by finding

a solution to a more general problem than the one originally stated and then by restricting this solution in order to fit the initial problem; moreover, its designer did not consider the algorithm's representation directly.

(2) Contrary to common opinion, a general program also has good time performances.

(3) There are at least 3 indications that a binary search algorithm is not as general as possible:

(a) the programmer has arbitrarily restricted the stated problem;

(b) the algorithm contains almost-identical parts;

(c) the algorithm contains redundant tests.

Then, we analyse 4 erroneous binary search algorithms; in particular, we showed that:

(1) Design errors occur inside algorithms probably designed by simulation of their execution.

(2) Coding errors are made in the description of actions to be executed when certain exceptional events occur.

We feel that the analysis of other algorithms could well verify some of these observations.

Acknowledgement

We are grateful to our colleague A. van Lamsweerde and to M. Hewitt for reading a draft of our paper.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison Wesley, Reading, Massachusetts (1973).
2. D. D. MacCracken, *Programmation des calculatrices numériques*, pp. 204–205, Dunod, Paris (1960).
3. C. Cherton, *Cours de Fichiers*, Vol. VI, pp. 24–26, F.N.D.P., Namur (1973).
4. J. J. Donovan, *Systems Programming*, pp. 82–84, McGraw-Hill, New York (1972).
5. I. Flores, *Computer Sorting*, pp. 39–54, Prentice-Hall, Englewood Cliffs, New Jersey (1969).
6. C. Bonnin, *Le Cobol ANS*, Dunod, Paris.
7. Chandra, see Ref. 1.
8. Bottenbruch, see Ref. 1.
9. T. Bartee, *Introduction to Computer Science*, pp. 310–311, McGraw-Hill, New York (1975).
10. C. Baudoin and B. Meyer, *Méthodes de programmation*, Eyrolles, Paris (1978).
11. O. Baufay, *Introduction au Hardware et au Software, Notes de cours*, pp. 74–79, CIGER, Namur.
12. E. W. Dijkstra, *Dijkstra's Lectures on the Design of Correct Programs*, Liège (1979).
13. C. W. Gear, *Computer Organization and Programming*, McGraw-Hill, New York (1969).
14. B. Lecharlier, private communication, Namur (1980).
15. G. Weinberg, N. Yasukawa, R. Marcus, *Structured Programming in PL/C. An Abecedarian*, p. 72, Wiley, New York (1973).
16. A. V. Aho, T. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, pp. 113–114, Addison Wesley, Reading, Massachusetts (1974).
17. A. T. Berztiss, *Data Structures—Theory and Practice*, p. 332, Academic Press, New York (1971).
18. W. H. Burge, *Recursive Programming Techniques*, p. 228, Addison Wesley, Reading, Massachusetts (1975).
19. S. De Angelo and P. Jorgensen, *Mathematics for Data Processing*, p. 228, McGraw-Hill, New York (1966).
20. J. R. Kelly and C. L. Mac Gowan, *Top-down Structured Programming*, pp. 33–34, Petrocelli-Charter, New York (1975).
21. H. D. Leeds and G. M. Weinberg, *Computer Programming Fundamental*, p. 228, McGraw-Hill, New York (1966).
22. H. Leroy. *Méthodologie de la Programmation*, F.N.D.P., Namur (1974).
23. C. E. Price, Table lookup techniques. *Computing Surveys* **3**, 5 (June 1971).
24. H. S. Stone and D. P. Siewiorek, *Introduction to Computer Organization and Data Structures*, pp. 313–316, McGraw-Hill, New York (1975).
25. N. Wirth, *Systematic Programming: Introduction*, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
26. J. Fichefet, *A propos de la recherche de Fibonacci de l'informaticien et de la recherche de Fibonacci du numéricien*, F.N.D.P., Namur (1977).

.Received July 1982