Informatics in Education, 2007, Vol. 6, No. 2, 385–396
© 2007 *Institute of Mathematics and Informatics, Vilnius*

# Some Competition Programming Problems as the Beginning of Artificial Intelligence

Boris MELNIKOV, Elena MELNIKOVA
*Department of Mathematics and Information Science, Togliatti State Univ.*
*Belorusskaya str., 14, 445667 Togliatti, Russia*
*e-mail: b.melnikov@tltsu.ru, e.melnikova@tltsu.ru*

Received: November 2006

**Abstract.** We consider in this paper some programming competition problems (which are near to some problems of ACM competitions) of the following subjects: we can make their solution using both Prolog and a classical procedure-oriented language. Moreover, the considered problems are selected that their solution in Prolog and in a classical procedure-oriented language are similar – i.e., in other words, they use the same working mechanism, the same approach to constructing recursive functions etc. However, the main goal of this paper is to demonstrate, that the solution of most of these problems by the students is a step to design the real problems of artificial intellegence.

**Key words:** competition programming problems, prolog, procedure-oriented languages.

## 1. Introduction

We can say, simplifying the matters, that we consider in this paper some interesting competition problems and their solutions; in this sentence, this paper can be considered as a continuation of (Melnikov and Melnikova, 2006). But, unlike that paper, all the problems considered below are connected by the following subject: we can construct their programs both in classical procedure-oriented languages and in Prolog.

However, it is the real simplifying the matters; it is that the main goal of this paper is not the solving problems, even in Prolog. Authors consider this paper (like (Melnikov and Melnikova, 2006)) as a paper of the set of ones, connected with solving similar problems; and the goal of this set of the papers is not to demonstrate some methods of solving competition problems, but to show, that this solving can give the real "transition" to design the real programs. A possible variant of this transition is the considered here "transition into artificial intelligence", i.e., the considered solving can be considered as a step to the real problems of the artificial intelligence.

However, like (Melnikov and Melnikova, 2006), we shall give consideration of the description of the solution of our problems using both classical procedure-oriented languages and Prolog. And we shall try to show not only *how* we solved the problem, but also *why* we solved it in such way[1], i.e., *how* we can *guess* the need solution.

[1]"You're far to keen and *where* and *how*, and not so hot on *why*" (Andrew Lloyd Webber and Tim Rice, "Jesus Christ Superstar").

Much of our problems were considered in the ACM competitions ((2005/2006 ACM International Collegiate Programming Contest) etc.), or, more often, are the subproblems of the problems considered there. And it is important to remark, that the considered problems are selected such as their solutions in Prolog and in a classical procedure-oriented language are almost the same – i.e., in other words, they use the same working mechanism, the same approach to constructing recursive functions etc.

Let us also remark in advance, that we shall not consider the classical problems of the graph theory (like constructing spanning tree, vertex cover etc.), which also coul be called by competition problems and are connected with the subjects considered below. There are two reasons of such thing. First, there are the limitations for the size of our paper. And second, there exist very good descriptions of such problems in a lot of books of applying Prolog for solving problems of artificial intelligence, see, e.g., (Bratko, 2001).

## 2. The "classical" Problems of the Representation a Number by a Sum

At first, we consider some classical problems, where the representation of a natural number as a sum of some other numbers is considered.

*Problem* 1. For the given numbers $n$ and $k$, we need to compute $r(n,k)$, which is the number of its representations as a sum of exactly $k$ positive integer items; we do not consider the permutations of items by new representations. For instance,

```
7 = 5+1+1 = 4+2+1 = 3+3+1 = 3+2+2
```

and we have no other variants, therefore, therefore $r(7,3) = 4$.

*Solution of Problem* 1 *in Pascal.* Solving this problem, we need to use the following standard technique (we use it almost always considering the representation of a number as a sum, see (Aho *et al.*, 1979) etc.): we add to the set of arguments of the computed function not only the arguments given in the problem situation, but also maximal possible item[2]. We shall not describe this method in details: it was many times described in a lot of books, and, besides, it can be explained following to the next text of the program. Thus, the solution can be found by the function

```
function R(N,K:integer): integer;
begin
  R:=RR(N,K,N);
end;
```

where the last argument of the function `RR` is the above-mentioned maximal possible item. Here, function `RR` can be realized (e.g., by the internal function for function `R`) in the following way:

```
function RR(N,K,Max:integer): integer;
  var S,i: integer;
```

[2]More strictly, such value is maximum possible for the item which could be choosen for the considered call of the function.

```
begin
  if (N<0) or (K>N) then RR:=0
  else if (N=0) and (K=0) then RR:=1
  else begin
    S:=0;
    for i:=Max downto 1 do inc(S,RR(N-i,K-1,i));
    RR:=S;
  end;
end;
```

*Solution of Problem* 1 *in Prolog.* As we said before, we can have “almost the same program” in Pascal and Prolog; i.e., we simple rewrite function `RR` into Prolog, for the serviceability changing the loop for the recursive call in the following way:

```
r(N,K,Answer) :- Max is N-K+1, r(N,K,Max,Answer).
r(N,_,_,0) :- N<0, !.
r(N,K,_,0) :- N<K, !.
r(_,_,0,0) :- !.
r(0,0,_,1) :- !.
r(N,K,Max,Answer) :-
  N1 is N-Max, K1 is K-1, r(N1,K1,Max,Answer1),
  Max2 is Max-1, r(N,K,Max2,Answer2),!, Answer is Answer1+Answer2.
```

Probably, we do not need comments. Let us only remark, that comparing to Pascal version, we only add the special computation of the value `r` if both its main arguments are equal to 0, i.e., `r(0,0,_,1)`.

Thus, we have almost the same program in a procedure-oriented language (Pascal) and in Prolog. But the main thing is that the same situation will be considered in the problem of Section 4, which is much more complicated. For two following problems, we shall consider only the statements: the mechanism of their solution is almost the same as one considered for Problem 1.

*Problem* 2. For the given number $n$, we need to compute $s(n)$, which is the number of its representations as a sum of some positive integer items; we do not consider the permutations of items by new representations. For instance,

```
4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1
```

and we have no other variants, therefore $s(4) = 5$ (we also consider 4 as a sum of the only item).

*Problem* 3. For the given number $n$, we need to compute $q(n)$, which is the number of its representations as a sum of some *different* positive integer items; we do not consider the permutations of items by new representations. For instance,

```
7 = 6+1 = 5+2 = 4+3 = 4+2+1
```

and we have no other variants, therefore $q(7) = 5$ (we also consider 7 as a sum of the only item).

Thus, we shall not consider the solutions of problems 2 and 3.

We can often reduct a competition problem to one of the considered three problems, including ones, where such reduction is not evident. For instance, let us consider the following problem.

*Problem* 4. Let us call a $2k$-digit number as a *good* number, if the sum of its first $k$ digits is equal to its sum of the last $k$ digits. For the given number $k$, we shall to calculate the number of $2k$-digit good numbers.

*Solution of Problem* 4. Let us consider only the algorithm; we will explain, that we can simply realize it both in Pascal and in Prolog, using the programs of Problem 1.

It is evident, that the trivial solution (i.e., the loop, in which we consider all the $2k$-digit numbers) is very bad. And we can obtain a good solution using the simply guess, that the variants of the digits in the first and the second parts of the number are independent. Therefore, knowing the numbers of variants of each of such possible sum (from 0 to $9k$; this computation is made like Problem 1), we obtain the answer as the sum of squares of all the variants of such sums.

Let us remark, that we have to make some little changes for the program of Problem 1, for the possibility of applying 0 as minimum possible item; however, this change is very simple, and we shall not describe such new program detailed. Thus, using the same notation $r(n, k)$ for the possibility $n = 0$, we obtain the following answer, which can be programed by a simple loop:

$$\Sigma_{0\leqslant n\leqslant 9k}\big(r(n,k)\big)^2.$$

Vice versa, the following problem seems to be like to the problems of representation a number as a sum, but its simple solution needs other methods.

*Problem* 5. We have to store information about all the possible representations of a given set as combining some subsets; and we have to know only the number of elements in each subset. For instance, we have the following representations for the set of 4 elements:

```
4,  3+1,  2+2,  2+1+1,  1+1+1+1
```

(compare Problem 2). We have to store information about the subsets only by one of two the following possibilities:

– either by the matrix of $N$ columns, where each string is the description of the subsets in decreasing order of the numbers of their elements; for instance, for 4 elements we obtain the following string for the representation $3 + 1$:

  ```
  3  1  0  0
  ```

– or by the array of the lists, where each element of this array is a list containing the next representation; for instance, considered representation 3+1 is represented here as the list containing 2 units; and each unit needs 3 bytes (where 1 byte is the information, and 2 other bytes are the reference to the next unit).

There is given a number of elements of the set, which is no more than 256. Which of two possibilities of storing is better?

*Solution of Problem* 5. The recurrent formulation, which gives the number of representations of N elements using such numbers for little subsets, is very complicated.

Therefore for the counting, we shall use the standard method which was already used before: our recursive function will use some additional arguments.[3] Let us describe such new arguments for our solution; we shall reference to the program given below.

First, it is variable `Max`, i.e., our artificially added limitation for the maximum number of elements in each set.[4] Second, it is variable `Deep`, i.e., the deep of nesting of the recursive calls. The returned value of function `Max` is the number of representations, and the common number of units (we need the deep of nesting for its counting) will be counted in global variable `Answer`, which has to receive value 0 before the main call of function `Rec`. Thus, function `Rec` is as follows:

```
function Rec (N,Max,Deep: byte): word;
  var S: word;
  I: byte;
begin
  if N=0 then begin KD:=1; inc(Answer,Deep); exit; end;
  if Max>N then Max:=N;
  S:=0;
  for I:=1 to Max do inc(S,Rec(N-I,I,Deep+1));
  KD:=S;
end;
```

As usually, it is better to define described recursive function Rec inside the following procedure, which returns the number of representations and the number of units:

```
procedure Razbivka (N: byte; var Razb,Answer: word);
begin
  Answer:=0;
  Razb:=Rec(N,N,0);
end;
```

We shall not compare need values, i.e., `Razb(N)*N and Answer(N)*3`: it is simple.

The considered function has the following defect, which is clear for big values `N`: while working, we call it many times using the same set of arguments.[5] We can improve this little defect by the next standard method: to put calculated values into special mini-DB.

We shall not consider the design of this mini-DB using Pascal; consider only Prolog version. Let us remark, that we can write the program without cuts, and, simplifying it, we do not write calculating number of units. Thus,

```
represent(N,Ans) :- rec(N,N,Ans).
rec(0,_,1) :- !.
rec(N,Max,Ans) :- db(N,Max,Ans), !.
```

[3]However, we can also write a recurrent formulation, which depends on more than 1 variable.

[4]We decided, that this value is no more than the value of the called argument. In the next recursive calls, we can only decrease it.

[5]The similar defect, in the much more simple interpretation, can be described, for example, counting `N`-th element of Fibonacci sequence using recursive function having statement `F(N):=F(N-1)+F(N-2)`: counting $F(N-1)$, we call $F(N-2)$ once more, etc. To improve this thing, we have to use, e.g., dynamic programming. In fact, some its parts are considered in this paper, but we do not consider this approach detailed.

```
rec(N,Max,Ans) :- cycle(N,Max,Ans,0), assert(db(N,Max,Ans)).
cycle(N,Max,Ans,Max) :- !.
cycle(N,Max,Ans,J) :- I is J+1, NI is N-I, rec(NI,I,Ans1),
  cycle(N,Max,Ans2,I), Ans is Ans1+Ans2.
```

## 3. An Arithmical Expression of the Given Sequence of Digits

The problem considered in this section has some analogues with ones considered in Section 2.

*Problem* 7. There is given a 5-digit number. We can insert signs (statements) `+, -, *, /, ^` (the last one means degree) and `!` (means factorial) between its digits. We can also use round brackets and the sign – (unary minus) before the number.

We shall consider the obtained text as the arithmetic expression and calculate its value; for instance, for the given number `12345`, we can obtain expressions `-12+34*5=158`, `12/3+4/5=1028` etc. We allow the division only if its result is integer.

We have to obtain the text of the following type:

`1 =` (the concrete arithmetical expression made of the given number, which value is equal to 1)

`2 =` (the concrete arithmetical expression made of the given number, which value is equal to 2)

`3 =` (the concrete arithmetical expression made of the given number, which value is equal to 3)

...

The number of the answer have to be without a break, and, in other words, the “grade” for the problem is equal to the maximum obtained number. Consider the following example (for the given number `12345`):

```
1 = ((12/3)/4)^5
2 = (1*2*3+4)/5
3 = 1-2+3-4+5
```

Here, we have the grade 3.

*Solution of Problem* 7 *in Prolog.* In this problem, there is a most difference between declarative and procedure approaches to the solution in Prolog. We do not consider here the implementation of the program, corresponding to the declarative approach; let us consider only the procedure one. It contains the dividing the given sequence of digits into possible numbers and also inserting signs of statements. In the Prolog solution, we use a little simplification, i.e., we do not allow degree statement. Thus, the main function `all` has the list of the digits as an argument and print its result; we put possible results (including intermediate ones) into a DB of functor `qq`.

```
all(Digits):-
  abolish(qq/2),
  repeat,
  form(Digits,Numbers),
     obr(Numbers,Answer,How,0),
     ifthen( not qq(Answer,_), assert(qq(Answer,How)) ),
     Numbers=[_],
     for(0,999,I),
     ifthenelse(not qq(I,_), (write('Answer: '), write(I),nl), fail ),
( I=0 ;
       dec(I,I1),
       repeat,
       write('Number from 0 till '), write(I1),
                                            write(': '), read(J),
       ifthen((J>=0,J<I),(qq(J,JHow),write('Answer:'),
                                            write(JHow),nl,fail))
     ).
```

The forming of the list of numbers (and also the transposition of the list of digits into the corresponding number) is made by two the following functions:

```
   form([],[]) :- !.
   form(Digits,[Head|Tail]) :-
     append(Digits1,Digits2,Digits), Digits1[̄],
     trans(Digits1,Head,_), form(Digits2,Tail).
   trans([C],C,10) :- !.
   trans([Head|Tail],Answer,U) :-
     trans(Tail,Answer1,U1), Answer is Head*U1+Answer1, U is 10*U1.
```

and the processing of the formed list of numbers is as follows:

```
   obr([C],C,C,_) :- !.
    obr(Numbers,Answer,How,Deep) :-
      ( append(Numbers1,Numbers2,Numbers), Numbers1[̄], Numbers2[̄],
       obr(Numbers1,Answer1,How1,0), obr(Numbers2,Answer2,How2,0),
       ( Answer is Answer1+Answer2, How=(How1+How2) ;
         Answer is Answer1-Answer2, How=(How1-How2) ;
         Answer is Answer1*Answer2, How=(How1*How2) ;
         mydiv(Answer1,Answer2,Answer), How=(How1/How2) ;
         Answer1=:=2, sqrt(Answer2,Answer), How=sqrt(2,Answer)
       )
      );
      ( Deep=<2, inc(Deep,Deep1),
       ( obr(Numbers,Answer1,How1,Deep1),
                               f(Answer1,Answer), How=f(How1) ;
         obr(Numbers,Answer1,How1,Deep1),
         sqrt(Answer1,Answer), How=sqrt(How1)
       )
      ).
```

We do not consider the text of other auxiliary functions. Let us only remark, that Deep is the deep of recursive calls (this variable is used for the possibility of the processing of the lists, which are longer than the given setting problem), and f is the function for calculating factorials.

*Solution of Problem* 7 *in C.* The following short solution we do not see straight away the analogues with Prolog solution considered before. However, like the problems considered before, we simply rewrite the solution: but here, from Prolog into a procedure-oriented language, i.e., into C++ in our case. The auxiliary function of computing result (which is similar to function `obr` of Prolog) is as follows:

```
long re (long x, long y, long op) {
  switch(op) {
    case 0:x*=10;return x+y;
    case 1:return x+y;
    case 2:return x-y;
    case 3:return x*y;
    case 4:if(x%y==0)return x/y;
           else return 0;
    case 5:long k=x;
           for(int j=1;j<y;j++)k*=x;
           return k;
    }
  return 0;
}
```

and the main actions are made by the following loops:

```
for(i=0;i<5;i++) {
  k=a[i];
  a[i]=a[0];
  a[0]=k;
  for(z[0]=0;z[0]<6;z[0]++) {
    if((x[0]=re(a[0],a[1],z[0]))==0)continue;
    for(z[1]=0;z[1]<6;z[1]++) {
      if((x[1]=re(x[0],a[2],z[1]))==0)continue;
      for(z[2]=0;z[2]<6;z[2]++) {
        if((x[2]=re(x[1],a[3],z[2]))==0)continue;
        for(z[3]=0;z[3]<6;z[3]++) {
          if((x[3]=re(x[2],a[4],z[3]))!=0)
            file«x[3]«"="«a[0]«o(z[0])«a[1]«o(z[1])«a[2]«o(z[2])
            «a[3]«o(z[3])«a[4]«endl;
} } } } }
```

## 4. Painting the Sequence of the Balls

In this section, we shall consider a problem of the quarter-final of the world student programming competition of 2003/04 years (ACM version).

*Problem* 8. We put the N white balls in a line and now we want to paint some of them in black, so that at least 2 black balls could be found among any M successive balls. We need exactly `C[i]` milliliters of dye to paint the `i`-th ball. The task is to find the minimum amount of dye we need to paint the balls.

*Solution of Problem* 8. Is it a complicated problem? But using all the methods we consider in this paper, we obtain very short solution! And the "final culmination" of our paper is as follows: we really use all the methods considered before. Namely:

– we use a procedure-oriented language "in Prolog style";
– we ignore the algorithm which consider all possibilities, and use the procedure approach (which is demonstrated, at first, using special working with argument nMade in the program given below);
– for the recursive calls, we use the additional argument `nSum`;
– the current optimal solution is stored in the global variable `Sum`; in Prolog, it is changed for an additional argument;
– we control the deep of recursive calls by the other additional argument `nDeep`.

As before, we can obtain "the same program" both in Pascal and Prolog. (The realisation of the last method is not difficult, but the corresponding comments are very long. We hope to consider this method in a next paper using some different examples, therefore we do not consider here the program in Prolog.)

Thus, we shall consider only the program in Pascal. Its global variables are the following ones:

```
var C: array [1..10000] of word;
  N,M,Sum: word;
  B: array [1..10000] of word;
```

(the last array marks painted balls for the recursive procedure `Make`). And this procedure `Make` is as follows:

```
procedure Make(nDeep,nMade,nSum:word);
  var bFalse: boolean;
    i,nMinus: word;
begin
  if nSum>=Sum then exit;
  if nDeep>N then begin Sum:=nSum; exit; end;
  if nDeep>=M then nMinus:=B[nDeep-M+1] else nMinus:=0;
  if nDeep>=M then bFalse:=nMade>=2
  else if nDeep=M-1 then bFalse:=nMade>=1
  else bFalse:=true;
  if bFalse then begin B[nDeep]:=0;
                 Make(nDeep+1,nMade-nMinus,nSum); end;
  {allways} B[nDeep]:=1;
                 Make(nDeep+1,nMade-nMinus+1,nSum+C[nDeep]);
  {we do not need "dec(Sum,C[i]);": we have no more
                                          recursive calls}
end;
```

The arguments of this function were already briefly commented. The internal variable `bFalse` defines whether the following ball will be painted. `nMinus` is the number of the balls for decreasing the number of painted ones using the recursive call; strangely enough, this value is calculated by much complicated sub-algorithm.

Thus, the main call of the considered recursive function is as follows:

```
Sum:=65000;
Make(1,0,0);
writeln(Sum);
```

## 5. Conclusion

We did not consider in this paper some important questions, which are connected with intellectual programming, rather, we have considered them very briefly. At first, it is declarative and procedure approaches to the solution of the problems in Prolog; however, such subject (i.e., both these approaches in competition problems) needs a lot of problems to be considered, and we hope to realize this thing in a nearest future.

It is important to remark, that we do not discuss "the questions of 80's": is it better Prolog than Pascal? than Lisp? Our approach is that all these languages are important for the students.

Thus, authors consider this paper as a paper of the set of ones, connected with solving similar problems. However, as we said before, the goal of this set of the papers is not to demonstrate some methods of solving competition problems. The main goal is to show, that this solving can give the real "transition" to design the real programs. A possible variant of this transition is the considered before "transition into artificial intelligence". And another possible "transition" (seems to be much more important), which will be considered in a next paper, is the "transition" into the real discrete optimization problems (see (Hromkovič, 2004), and also some papers of the authors, (Melnikov, 2005; Melnikov *et al.*, 2006); see also the references, which are in those papers, at first in (Melnikov *et al.*, 2006)).

Authors have the following plans for some next papers. First, it would be a paper about using the classical algorithms and data structures in Olympiad problems, i.e., algorithms and data structures "of Wirth", "of Cormen, Leiserson and Rivest" etc. Second, it would be a paper about some classical hard problems and their simplification as competition problems, and also about some methods of their solution. And the subject of the third paper would be formulated as "Olympiads as a step to PhD".

## References

*2005/2006 ACM International Collegiate Programming Contest*, Saratov State Univ. Ed. (in Russian and English).

Aho, A., J. Hopcroft and J. Ullman (1979). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Bratko, I. (2001). *Prolog Programming for Artificial Intelligence*, 3rd Ed. Addison-Wesley.

Hromkovič, J. (2004). *Algorithms for Hard Problems. Introduction to Combinatorial Optimazation, Randomization, Approximation, and Heuristics*, 2nd Ed. Springer.

Melnikov, B. (2005). Discrete optimization problems – some new heuristic approaches. In *8th International Conference on High Performance Computing and Grid in Asia Pacific Region* (*HPC-Asia–2005*), China. IEEE Computer Society Press Ed., pp. 73–80.

Melnikov, B., and E.Melnikova (2006). Some programming olympiad problems with detailed solutions. In *International Conference on Informatics in Secondary Schools, Evolution and Perspectives* (*ISSEP–2006*), Lithuania, pp. 573–584.

Melnikov, B., A. Radionov, A. Moseev and E. Melnikova (2006). Some specific heuristics for situation clustering problems. In *1th International Conference on Software and Data Technologies* (*ICSOFT-2006*), vol. 2, Portugal, pp. 272–279.

**B. Melnikov** received his BS from Moscow State University in 1984 and PhD at the same University in 1990 under the direction of Larisa Stanevičienė. Since January 1992, he has been at Ulyanovsk Branch of Moscow University and Ulyanovsk State University. He received his Habil. Degree at Moscow State University in 1997. In 1999–2003, he was a full professor of Ulyanovsk State University, and since November 2003, he has been a full professor of Togliatti State University (Russia). Main research interests: finite automata, regular languages, theory of semigroups; heuristical algorithms (genetic, branch-and-bounds etc.); artificial intelligence; mathematical aspects of the object-oriented programming; informatics in education.

**E. Melnikova** received her BS from Moscow State University in 1984. In 1984-2005, she was working in Secondary Schools. Since 2005 she has been an associated professor of Togliatti State University (Russia). Main research interests: heuristical algorithms, artificial intelligence; informatics in education.

## Olimpiadiniai programavimo uždaviniai – kompiuterinės intelektikos pradmenys

Boris MELNIKOV, Elena MELNIKOVA

Straipsnyje nagrinėjama keletas olimpiadinių programavimo uždavinių. Aptariami jų sprendimai naudojant Prologą ir procedūrinę programavimo kalbą. Uždaviniai buvo atrinkti taip, kad vienodai tiktų Prologas ir procedūrinė kalba, t.y. jų sprendimų algoritmai panašiai veiktų, būtų panašus rekursinių funkcijų sudarymas ir t.t. Vis dėlto pagrindinis straipsnio tikslas yra parodyti, kad studentai spręsdami daugumą šių uždavinių, gali įgyti kompiuterinės intelektikos kaip mokslo pradmenų.