Using of Memoization in Arithmetic Operations Sign Placement Problems

https://doi.org/10.31713/MCIT.2019.05

Oleksandr Shportko
Department of Information Systems and Computing Methods
International University of Economics and Humanities
Academician Stepan Demianchuk
Rivne, Ukraine
ITShportko@ukr.net

Andrii Bomba
Department of applied mathematic
National university of water and environmental engeneering
Rivne, Ukraine
e-mail a.bomba@ukr.net

Kateryna Malash
Department of Computer Sciences and Applied Mathematics
Rivne State University of Humanities
Rivne, Ukraine
katemalash@gmail.com

Abstract—**The article compares two ways of solving the arithmetic operations sign placement problem: method based on recursion and dynamic programming based one which uses the memoization. Limitations on the intermediate results of the problem solution are identified and justified. It is shown that the use of memoization to cut off ineffective search options makes it possible to accelerate the corresponding algorithms execution by tens times.**

Keywords—memoization; dynamical programming prolems; arithmetic operations sign placement

I. Introduction

As it's known, programming memoization is the storing code snippets to prevent recalculations [1]. Traditionally, memoization is used when memorizing the results of a function for the given parameters values. Then at repeated calls of the same function with the same parameters the stored value is returned at once, and the same calculations are not performed again (for example, for the recurrent calculations problem of the factorials sequence [2]). Also memoization is used not only to increase the speed of program execution, but also, for example, during recursive parsing in a generalized descending algorithm [3] or when tabulating predicate values in logical programming languages [1]. In this way, memoization makes it quicker to perform calculations, although it does require memory costs to save the calculation results. In this article we show how memoization can be used to speed up arithmetic signs placement problem.

II. Problem Statement. Literary Sources Analysis

Arithmetic sign placement problems are offered are widely used in the educational process, in mathematics and computer science competitions both in schools and universities. It is clear that we would like to get all the solutions for every arithmetic operator sign placement task quickly so developing software to solve these problems effectively is a **pressing problem today**. Traditionally it is suggested to place the arithmetic operations signs and parentheses between the given digits of the entered line so that to maximize the value of the arithmetic expression obtained in this type problems. The following problems are solved using the principle of optimality on the snippets [4] of the dynamic programming method [5], since it is not necessary to store all possible values of the arithmetic expressions obtained for each digits sequence of the entered string, and it is sufficient to remember only that signs sequence at which the resulting arithmetic expression value will be maximal. We consider a more complex problem, in which we need to minimize the number of arithmetic operations for **each value of the expression** for the next sequence of digits, which starts from the beginning of the line. Let us explore, for example, ways to accelerate resolution of task D from 1/8 of the ACM-ICPC 2018 Command Program Olympiad, held in Ukraine on April 21, 2018 by memoizing. This task was solved by only a few teams at the Olympics, which also indicates the relevance of this study. The condition of this problem is formulated in the paraphrased form as follows:

Task D. Strange Equation. Write the program that inserts **the least number of operations** to the left part of the string $a = s$ $(0 \leqslant a < 10^{1000},\ 0 \leqslant s \leqslant 5000)$ so that the expression is correct. The numbers in the corrected equation may contain an arbitrary number of leading zeros. If the task has multiple solutions, output any of them. For example, for input string 4475 = 56, 4 + 47 + 5 = 56 or 44 + 7 + 5 = 56 should be displayed. The task should take up to 1.5 seconds to complete.

At first glance, it seems that this task could be solved by the direct iteration method [6]: after each digit, except the last one, there may be a '+' sign or may not be. That is, there are two options. The total count of different variants of arithmetic operations sign placement between all the numbers of the corrected equations is $2^{countDigit-1}$. Given the limitations of the task, this number can reach 2^{999}, which makes it impossible to solve the task in the allotted time by the mentioned direct iteration method. Therefore, we propose options for solving the problem by recursion and dynamic programming with optimality in the prefix [5], using memoization.

III. Limitations on Intermediate Results

It is clear that after reading the input line, a must be a string and s must be a number. Let $countDigit = \text{length}(a)$ is the count of digits in the left part of expression $a = s$, $digit_i = Convert.ToInt(a[i])$, $i = \overline{0, countDigit - 1}$ is the current digit from this part, and $number_{i,j} = \sum_{l=i}^{j} 10^{j-l} digit_l$ is the decimal number that starts with $digit_i$ and ends with $digit_j$.

Let us analyze the four-tier possible solution tree for the example of the task condition given in fig. 1. The top of this tree corresponds to the initial digit $digit_0$, and each subsequent level are options for including the next digit. To ensure the compactness of the tree at its fourth level, only the values of the left parts of the expressions are given.

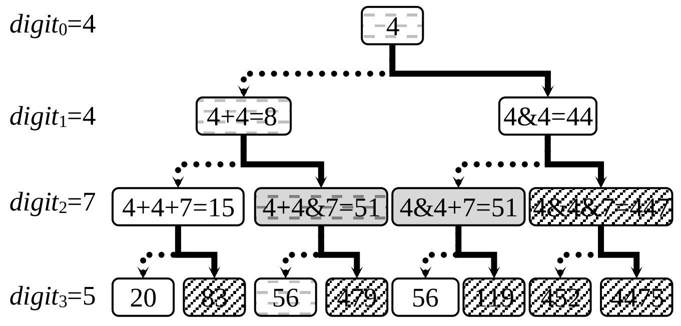

Figure 1. Tree of possible variants for expression 4475=56

With each next digit, the count of variants is doubled, as this digit can both be **added** to each of the previous variants (it's indicated by a dotted line between the blocks on fig. 1) and **accession at the end of the last addition** to them (it's indicated by a solid line between the blocks and by '&' in the middle blocks).

Limitations on the intermediate results follow from the following considerations:

1. The inclusion of each digit does not reduce (it always increases if it is not zero) the value in the left part of the expression. The smallest value of the left side will be when inserting plusses between all digits, and the largest one will be when pluses will be absent. **Therefore, expressions in which the last addition or the entire sum exceeds *s* should be excluded from further consideration.** Such blocks are shaded diagonally on fig. 1. For the example given, such expressions make it possible to reduce the number of calculations in the third tier 25%, and in the fourth one by 62.5%.
2. It's impossible to combine branches with the same expressions values at each level, since further joining of the following digits can give rise to different variants. For example, it's impossible to combine two variants on third tier with the values 51 (displayed on a gray background on fig. 1), since at the fourth tier after the accession of digit 5, they give rise to two different variants with the values 479 and 119. Although, **adding** this digit gives two equal values 56. Therefore, **combining variants with the same values at the same tier is only advisable when those variants have the same last addition.** In other words, it's advisable to combine variants with the same values only before adding. Based on the task statement, for the further consideration of such variants **the one that has fewer pluses should be leaved**.
3. At the last tier, among all the variants, it's necessary to choose the one, which is, first, equal to *s*, and, second, among all variants equal to *s*, has the least number of pluses. For example, on fig. 1 every of two options with value 56 on forth tier can be left out, since each of them has two pluses. Then it's necessary to define the pluses that form the selected variant (the blocks of this path are shaded horizontally on fig. 1).

IV. A Recursive Method of Solving the Arithmetical Signs Placement Task

The left side value does not decrease with each addition. Therefore, in order to solve this task, it is necessary to divide the total sum s, written in the right-hand side between the additions of the left-hand side and choose among all such distributions the one containing the least number of pluses. Therefore, let examine the mechanism of procedure *InsertPlus*, written by C# execution, for insertion the next plus sign after each valid number $number_{startPos,i}$ (not greater than *remainder* – retained balance *s*), if *countLeftPlus* has already met before *startPos* position:

```
static void InsertPlus(int startPos,
            int countLeftPlus, int remainder)
```

This procedure is aimed for finding the smallest number of pluses for the distribution of *s* between the left-hand side additions, so if the number of pluses has accumulated to the left of the *startPos* position equal to the minimum at this time for such distributions, then placing the following pluses is no longer relevant:

```
 {if (countLeftPlus == minCountPlus)
   return; //a better distribution can't be got
```

Otherwise this procedure recursively generates numbers $number_{startPos,i}$ by increasing *i* until all the digits of the left part are processed or the next number does not exceed the retained balance *remainder*:

```
  int number = 0; //variable for valid numbers
  int i = startPos, last;
  //previous insignificant zeros are skipped
  while (i<countDigit-1 && digit[i] == 0) i++;
  while (i < countDigit)
   {//last indicates the last digit processed,
    //and i indicates the next digit
    number = number * 10 + digit[last=i++];
    if (number > remainder)
     return; }
```

If all the digits of the left part are processed and the next number coincides with the undistributed remainder (i.e. the total sum of regular additions of the left part is equal to *s*) and the received number of pluses is less than the minimum at this time, then we remember the received number of pluses and their positions:

```
    if (i==countDigit)
     {if (number == remainder)
       if (countLeftPlus < minCountPlus)
        {minCountPlus = countLeftPlus;
         for (int j = 0; j < minCountPlus; j++)
          minPozPlus[j] = pozPlus[j]; }
```

The procedure is completed after processing all the digits of the left side:

```
      return; }
```

Otherwise, if after the formed number $number_{startPos,i}$ there are still numbers, then we fix '+' after that number and place the pluses **recursively** starting from the next digit. In this case, the retained balance is reduced by a number $number_{startPos,i}$:

```
m:  pozPlus[countLeftPlus] = last;
    InsertPlus(i, countLeftPlus + 1,
               remainder - number); }}
```

Before calling this procedure for the first time, we put a minimum number of pluses equal to the number of digits on the expression left side. Any acceptable variant of pluses placement between these digits provides the less count of them (after all it is always possible to put less pluses between digits than there are digits). Therefore, if the minimum number remains unchanged after recursive placement of the pluses in the whole left part, then it is impossible to place them correctly:

```
minCountPlus = countDigit;
/* we distribute the right part from the
beginning of the left one */
InsertPlus(0, 0, s);
if (minCountPlus == countDigit)
 {Console.WriteLine("Problem solving is absent");
  return; }
```

Otherwise, the pluses between the digits of the left side are inserted and the result is output:

```
string res = ""; j=0; //the next plus index
for (i = 0; i < countDigit; i++)
 {res+= a[i].ToString();
  if (j < minCountPlus && minPozPlus[j] == i)
   {res += "+"; j++; }}
res += "=" + s.ToString();
Console.WriteLine(res);
```

The above recursive procedure for solving the problem, of course, takes much less than the $2^{countDigit-1}$ placement options, since it takes into account the first and last restrictions on the intermediate results from the previous section. But at the maximum *countDigit* and *s* values, this procedure takes longer than twenty-four hours (!). Therefore, let's speed up this procedure using memoization. To do this, we keep minimum number of pluses placed on the left for **each** digit and **each** subsequent unallocated remainder in the two-dimensional array. These values prevent further recursive calls if you need to distribute the same remainder after the same position, and the number of pluses on the left has not decreased (in fact, this is a pooling of intermediate results under the second constraint). The code snippet for such an acceleration of the above procedure is inserted before the mark *m* i and may look like this:

```
/* if such a remainder was considered from this
position and the number of pluses on the left
was not greater */
if (prevCountPlus[last, remainder-number]>0 &&
    prevCountPlus[last, remainder-number]<=
      countLeftPlus)
 continue; //we move to the next digit
//else we remember less count of pluses
prevCountPlus[last, remainder-number] =
  (short)countLeftPlus;
```

Such memoization speeds up the execution of the above procedure at maximum values of *countDigit* and *s* up to 11.6 s (in more than 7000 times), although it requires an additional 2×(*countDigit*-1)×*s* bytes of memory to store the *prevCountPlus* array and still does not satisfy the time limit (1.5 s) of solving the task. So, let's look at another way to solve it.

V. Solution of Arithmetic Operations Sign Placement Problem using Dynamic Programming Method

We give the recurrent formulae for the **direct** course of solving the problem by the dynamic programming method. Let the system **state variable** $\xi_{i,j}$ be the accumulated *j*-s sum from the beginning of the string to the opposition *i*. It is clear that

$$\xi_{0,0} = digit_0 . \qquad (1)$$

As noted, each subsequent digit $digit_i$ increases the number of options twice, as this digit can both be added to each of the previous accumulated sums $\xi_{i-1,j}$, and attached to its last number $lastNumber_{i-1,j}$. So,

$$\xi_{i,j}=\begin{cases}\xi_{i-1,j/2}+digit_i,\text{ if } j \text{ is even number,}\\ \xi_{i-1,(j-1)/2}+9\times lastNumber_{i-1,(j-1)/2}+\\ \qquad digit_i,\text{ if } j \text{ is odd number,}\end{cases} \tag{2}$$

$$i=\overline{1,\,countDigit-1},\ j=\overline{0,\,2^i-1},$$

where

$$lastNumber_{0,0}=digit_0, \tag{3}$$

$$lastNumber_{i,j}=\begin{cases}digit_i,\text{ if } j \text{ is even number,}\\ 10\times lastNumber_{i-1,(j-1)/2}+digit_i,\\ \qquad \text{if } j \text{ is odd number.}\end{cases} \tag{4}$$

The upper branches in (2), (4) correspond to the addition of the next digit, and the lower ones correspond to the attaching the last digit. Let us now denote the number of pluses in the formation of the accumulated sum $\xi_{i,j}$ by $countPlus_{i,j}$. Then the **objective function** of the problem is given as follows:

$$countPlus_{0,0}=0, \tag{5}$$

$$countPlus_{i,j}=\begin{cases}countPlus_{i-1,j/2}+1,\text{ if } j \text{ is even,}\\ countPlus_{i-1,(j-1)/2},\text{ if } j \text{ is odd.}\end{cases} \tag{6}$$

To apply the recurrence relations (2), (4), (6) to forming the problem solution, we first determine the smallest number of addition operations in the whole string a and the variant index at which this minimum is reached:

$$minPlus=\min_{0\le j\le 2^{countDigit-1}-1}\left(countPlus_{countDigit-1,j}\mid \xi_{countDigit-1,j}=s\right),$$
$$\exists index^*_{countDigit-1}\mid countPlus_{countDigit-1,index^*_{countDigit-1}}=minPlus. \tag{7}$$

An index of the variant that provides the total minimum number of pluses for each of the preceding characters is calculated iteratively in the process of dynamic programming method **reverse**:

$$index^*_i=\lfloor index^*_{i+1}/2\rfloor,\ i=\overline{countDigit-2,\,0}. \tag{8}$$

Finally in the problem solution we insert pluses between all adjacent characters i and i+1 on the a left side when $countPlus_{i,index^*_i}\neq countPlus_{i+1,index^*_{i+1}}$ (i.e. the minimum number of pluses from the string beginning to adjacent characters is different).

As noted, given the constraints of the problem, the number $\xi_{i,j}$ reach 2^{999}, which makes it impossible to solve it within the allotted time directly by relations (1) - (6). Therefore, for $\xi_{i,j}$ we take into account and strengthen the first two constraints given in the third section (the third of these constraints is taken into account by relations (7), (8)).

The first of these constraints is formalized in the form $\xi_{i,j}\le s$. Given that with each digit the accumulated amounts increase by at least this digit, we tight this limit to

$$\xi_{i,j}\le s-\sum_{k=i+1}^{countDigit-1}digit_k\,. \tag{9}$$

For long strings of the form $a = s$ this restriction allows to reject more than 90% of the system states $\xi_{i,j}$.

The second restriction provides for the possibility of rejection $\xi_{i,j}$, if, **when adding the next digit**, the previous accumulated sum $\xi_{i-1,j/2}$ can be provided with no more pluses, ie rejection is possible if there is $k<j$ (j and k are even), such that

$$\xi_{i-1,j/2}=\xi_{i-1,k/2},\ countPlus_{i-1,j/2}\ge countPlus_{i-1,k/2}. \tag{10}$$

To implement these limitations, we apply memoization: instead of analyzing for each $\xi_{i-1,j/2}$ all the previously calculated $\xi_{i-1,k/2}$ and corresponding amounts of pluses, we create a one-dimensional array *minPlusPrevSuma*, in which for each calculated amount accumulated up to the previous digit the minimum number of pluses is stored. Then, **when adding the next digit**, we save each accumulated amount for the first time only, increasing the minimum number of pluses to the previous digit by "one". Other variants with the same cumulative amount can be ignored when adding the same digit. The *minPlusPrevSuma* array is formed from the *minPlusCurrentSuma* array, which contains the minimum number of pluses up to the current digit, including for each calculated accumulated sum, and is also used to form the result string.

Using in practice the recurrent relations (2), (4), (6), we only store the accumulated sums $\xi_{i,j}$, that satisfy the constraints (9), (10), and, therefore, it is impossible to use (8) for the reverse. Therefore, to implement the reverse of the dynamic programming method, we create a two-dimensional array of digits in the last added number *arrayCDLN*[*i*, *suma*] (abbreviation of array of Counter Digits in the Last Number) for each position *i* and the accumulated sum of *suma*. He same array is used for checking the same accumulated sum in relation to the next digit.

A snippet of the C# program for "pluses" placement using dynamic programming is given below. As follows from (1) – (6), each variant of the accumulated sum is characterized by the last addition and the number of "pluses". Therefore, to store such variants, we create a suitable structure, where the field *suma* contains $\xi_{i,j}$:

```
struct Variant
 {public int suma, lastNumber;
  public short countPlus; }
```

At the beginning of the program, we form arrays of digits and limits on the accumulated amounts for each position on the right (9):

```
countDigit=a.Length;
digit = new int[countDigit];
for (i = 0; i < countDigit; i++)
 digit[i] = a[i] - 48;
//the maximum amount allowed for each position
int[] maxSuma = new int[countDigit];
maxSuma[countDigit - 1] = s;
for (i = countDigit - 2; i >= 0; i--)
 maxSuma[i] = maxSuma[i + 1] - digit[i + 1];
```

We also create an array of digits count in the last added number for each allowed position and the accumulated sum, two arrays to store the minimum number of "pluses" for the options of the previous and current positions, declare lists of variants of these positions and form an option for the first digit:

```
short[,] arrayCDLN = new short[countDigit, s+1];
int[] minPlusPrevSuma = new int[s + 1];
int[] minPlusCurrentSuma = new int[s + 1];
List<Variant> prevVariant=new List<Variant>(1);
List<Variant> currentVariant,
//we form a node from the first digit
Variant v;
v.suma = v.lastNumber = digit[0]; //(1), (3)
v.countPlus = 0; //(5)
prevVariant.Add(v);
//number includes one digit
arrayCDLN[0, v.suma]= 1;
```

During the **direct course** of the dynamic programming method we create a list with double capacity relative to the number of variants of the previous position, because to the previous variants the next digit can be added as well as attached. For the option of addition the last digit, we save each received amount only once - with a minimum number of "pluses":

```
//the direct course of the dynamic programming method
for (i = 1; i < countDigit; i++)
 {currentVariant=new
        List<Variant>(prevVariant.Count*2);
  //adding a digit to the variants of the previous digit
  for (j = 0; j < prevVariant.Count; j++)
   {suma = prevVariant[j].suma + digit[i];
    //implementation of the constraint (9)
    if (suma>maxSuma[i]) continue;
    /*  we note if such a sum has not yet been met
    with current digit (restriction (10)) */
    if (arrayCDLN[i,suma]==0)
     {//we form a new node
      v.suma = suma; //(2) for even ones
      v.lastNumber = digit[i]; //(4) for even
      /* the number of "pluses" is the minimum of all
      the options that provided the previous amount*/
      countPlus = (short)
       (minPlusPrevSuma[prevVariant[j].suma]+1);
      v.countPlus = countPlus; //(6) for even
      currentVariant.Add(v); //note option
      minPlusCurrentSuma[suma] = countPlus;
      //the last adding contains only one digit
      arrayCDLN[i, suma] = 1; }}
```

If the next digit is attached to the variants of the previous one (in formulas (2), (4), (6) these are branches for odd), then we only reject variants that do not satisfy the constraints (9):

```
  for (j = 0; j < prevVariant.Count; j++)
   {suma = prevVariant[j].suma +
     prevVariant[j].lastNumber*9+digit[i];//(2)
    //implementation of the constraint (9)
    if (suma > maxSuma[i]) continue;
    v.suma=suma;
    v.lastNumber=prevVariant[j].lastNumber*10+
     digit[i]; //(4) attaching the digit
    v.countPlus=prevVariant[j].countPlus; //(6)
    currentVariant.Add(v);
    /* record the variant if such amount has not
    yet met relative to this digit, or was with
    more "pluses" */
    if (arrayCDLN[i, suma] == 0 ||
        (arrayCDLN[i, suma]!=0 && v.countPlus<
          minPlusCurrentSuma[suma]))
     {minPlusCurrentSuma[suma] = v.countPlus;
      /* changing the length of a number when
      attaching a digit */
      arrayCDLN[i, suma] = (short)
       (arrayCDLN[i-1, prevVariant[j].suma]+1);
     }}
  /* moving to the next digit current the minimum
  numbers of "pluses" become preliminary */
  prevVariant = currentVariant;
  prom = minPlusPrevSuma;
  minPlusPrevSuma = minPlusCurrentSuma;
  minPlusCurrentSuma = prom; }
```

In the process of **reverse course** of the dynamic programming method, we attach the digits from the end, focusing on the number of digits of the last addition:

```
if (arrayCDLN[countDigit-1,s]!=0)
 {string res = '=' + s.ToString();
  i = countDigit - 1; //processing position
  suma = s; //undistributed remainder at left side
  while (i >= 0)
  {//the length of the last addition
   j = arrayCDLN[i, suma];
   int multiplier=1; //junior grade weight
   while (j>0)
   {res = s[i].ToString() + res;
    suma-=digit[i]*multiplier;
    /* the weight of the previous digit in
    the decimal system */
    multiplier*=10;
    i--; //move left in the left side
    j--;}; //moving by the digits of the last number
   if (i>=0) //if there are still additions
    res = "+"+res; }
  Console.WriteLine(res); }
else
 Console.WriteLine("No issues”);
```

At maximum *countDigit* and *s* values without memoization (that is, finding the minimum number of pluses by directly searching the *prevVariant* and *currentVariant* list items without using the *minPlusPrevSuma* and *minPlusCurrentSuma* arrays), this program takes 14.76 s, and with memoization only 0.39 s. That is, the memoization in this case accelerated the execution of the program by 37 times and allowed to satisfy the time limit (1.5 s) for solving the problem. For these values, the dynamic programming method made it possible to solve the problem of recursive subroutine calls 29 times faster, primarily by avoiding the repeated storage of values in the local variables of each of these calls. It is also interesting that the dimensions of the *arrayCDLN* array to provide a backward dynamic programming method are almost identical to the size of the recursive memoization array, that is, the program in this section does not use more RAM than the program in the previous section.

VI. Experimental Results

In conclusion, let's analyze the performance metrics for these two ways to solve a given problem for three arbitrary *a* of 1000 digits $s \geq 5000$ (tabl. 1).

TABLE I. PERFORMANCE OF "PLUSES" PLACEMENT PROGRAMS FOR EXPRESSION $A = S$ AT A LENGTH OF 1000 IN DIGITS FOR DIFFERENT VALUES S

Indicator	***s***		
	5000	***26726***	***174144***
Time of placement of recursive calls, s	11.60	220.16	3421
Time of placement by dynamic programming method, s	0.39	2.21	22
The ratio of durations of recursive call method placement to dynamic programming method	29.74	99.62	155.45
Minimum number of "pluses" (reference)	*656*	*472*	*315*

It's clear that with increasing *s* the duration of program execution increases, because the number of possible batch options increases, but the time of arrangement by the method of dynamic programming increases more slowly than the time of execution of recursive calls, which indicates its advantages.

Conclusions

1. Memoization should be used not only to speed up the execution of recursive calls, but also to implement the dynamic programming method and, in general, to reduce the number of nested cycles, which will significantly accelerate the execution of programs. At the same time, only the results of the calculations should be stored, which can be used in the future repeatedly.
2. In order to accelerate the solution of sequential decision-making problems, it is advisable to use a dynamic programming method in the algorithmization process rather than recursive calls of subroutines. The implementations of this method, although they require a large amount of memory to backfire, but each time they do not store in the stack the values of all local variables for each recursive call.
3. To save RAM, the values of the state variables and the target function should be stored only for the current and previous steps of the dynamic programming method, and for all steps and states of the system, only a minimum of data should be stored to provide a solution during the reverse process of this method.
4. In the process of implementing the dynamic programming method, it is advisable to consider not only the target function but also the variable $\xi_{i,j}$ of each state. This not only saves memory while not storing invalid values, but also significantly speeds up calculations for future states.

References

[1] Memoisation – Wikipedia [Online resource]. – Access mode: https://ru.wikipedia.org/wiki/Мемоизация.

[2] What is the fastest factorial feature in JavaScript? [Electronic resource]. – Access mode: http://qaru.site/questions/83626/fast-factorial-function-in-javascript.

[3] P. Norvig, “Techniques for automatic memoization with applications to context-free parsing,” Computational Linguistics, vol. 17, no.1, 1991, pp 91–98.

[4] The Problem of Arrangement of Signs in Expression - Wikisource [Electronic resource]. - Access mode: https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_расстановке_знаков_в_выражении.

[5] Dynamic Programming - Wikisource [Electronic resource]. - Access mode: https://neerc.ifmo.ru/wiki/index.php?title=Динамическое_программирование.

[6] Complete Search [Online Resource]. - Access mode: http://znaimo.com.ua/Повний_перебір