journal homepage: www.sabauni.net/ojs

Article

Increasing the Speed of the Recursive Algorithm and Reducing Stack Memory Consumption by Using the Dynamic Rule(base)

Nashwan Saeed M.G. Al-Thobhani [1] *, Naser Ahmed O. Al-Maweri[1] , Jamil Sultan[1]

[1] ***Sana'a Community College, Sana'a, Yemen***

Article info

Article history:

Accepted: April. 2020

Keywords:

Recursion;
Rule (base);
Dynamic;
Fibonacci;
Recursive algorithms

Abstract

Recursive algorithms always consume a computer's memory stack, and in this paper we worked to increase the speed of the recursive algorithm through a dynamic rule(base) that changes during its implementation process. Dynamic rule(base) regulation often allows avoiding repeated calculations of the same sets of parameter values, which reduces the number of repeated calls and simplifies slow calculations. Here, a mechanism will be created for the rule using simple and well-known examples to calculate the Fibonacci sequence, recurring linear sequences of general shape, and binomial transactions.

* Corresponding author: Nashwan Saeed M.G. Al-Thobhani
E-mail: nashwansg@gmail.com

1. Introduction

Recursive algorithms belong to the class of algorithms with high resource consumption, since with a large number of self-calls of recursive functions, the stack area is quickly filled [1]. In addition, organizing the storage and closing of the next layer of the recursive stack are additional operations that require time. The complexity of recursive algorithms is also affected by the number of parameters passed by the function [9]. Consider one of the methods for analyzing the complexity of a recursive algorithm, which is built on the basis of counting the vertices of a recursive tree [9]. To estimate the complexity of recursive algorithms [2], a complete recursion tree is constructed like Figure 1. It is a graph, the vertices of which are the sets of actual parameters for all calls to the function, starting from the first call to it, and the edges are the pairs of such sets corresponding to mutual calls. In this case, the nodes of the recursion tree correspond to the actual calls of the recursive functions. It should be noted that the same sets of parameters can correspond to different nodes of the tree. The root of the complete recursive call tree is the top of the complete recursion tree corresponding to the initial call to the function.

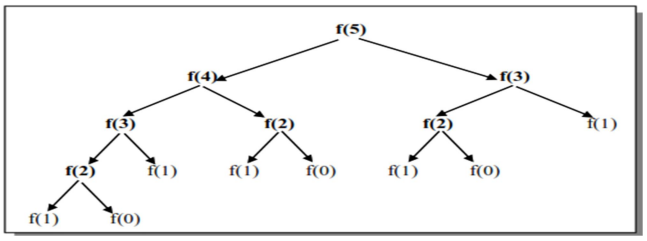

Figure 1. Schematic representation of recursive calls when finding f (5)

When constructing any recursive algorithm, in addition to parameter determination and decomposition, the choice of its rule (base) β (it's just a symbol that we refer to the dynamic rule (base)), that is, the selection of a subset of sets of acceptable parameter values through which calculations are made the algorithm is very simple and provides a return mechanism for repeated calls. Usually this rule does not change during calculations. With a fixed rule, it is often necessary to perform multiple function calculations for the same sets of parameter values [6].

Some systematic solutions related to increasing the speed of recursive algorithms through dynamic rule(base) during implementation. The dynamic rule (base) often allows avoiding repeated calculations of the same sets of parameter values, which reduces the number of repeated calls and simplifies slow calculations [6]. The dynamic rule (base) mechanism is explained by using simple and well-known examples of calculating the Fibonacci sequence, recurring linear general-form sequences and binomial coefficients.

2. Sequence Fibonacci

Let's begin with calculation a member of sequence:

$$f(0) = f(1) = 1, \quad f(n) = f(n-1) + f(n-2) \quad (n = 2, 3, \dots). \tag{1}$$

The recursive function for calculating f (n) with a static base (0,1) directly implemented by formulas (1), looks like this:

```
function f(n:integer):int64;
  begin
    if n<2 then result:=1                    (2)
    else result:=f(n-1)+f(n-2);
  end;
```

With increasing n, the number k (n) of recursive calls by (2) grows approximately as $0.725*(1.62)^n$ (see the statement of Theorem1) [4, p71]. Therefore, calculations according to (2) are rather laborious and already at n = 50 it is difficult to implement. We modify (2), turning it into a function with a dynamic rule (base). For this, in the scope of the projected function, we define the variables n, k and the array v:

var n, k: integer,
v: array [0..1000] of int64;
In the array v we will store the expanding base, and in k the current number of elements in it. The initial values for the base β and the counter of the number of elements in it are determined as follows: v [0]: = 1; v [1]: = 1; k = 2. In the future, it is supposed to enter into the base each newly calculated value of the function. Then the "dynamic modification" f(n) can be written in the form (3).

```
function fbase1(n:integer):int64;
  var a,b:int64;
  begin
    if n>=k then
     begin
       a:= fbase1(n-2);b;= fbase1(n-1);
       v[n]:=a+b;k:=k+1;
     end;
  result:=v[n];
end;
```
(3

When solving the problem, each recursive call, including the initial start of the calculations, initiates work, as it were, from the original algorithm. The sequence of calculations of the values of local and global variables corresponding to one specific "virtual instance" [6] of the algorithm and not including Calculations on calls from a given instance itself are called a slice of recursive calculations. It is convenient to consider exits from a particular slice a to a slice of the next depth of recursive nesting or to any subprogram as an appeal to some "black box" [4, p. 272] that transforms and returns all or some values from the scope of α. A specially designed calculation form, which somehow fixes the calculation of a particular recursive slice, is called the form. The form should indicate the relationship between the steps of the calculations and, in addition, a location for the calculations to be proposed. A completed form is called an embodiment, and a sequence of incarnations corresponding to a sequence of recursive calls is called a recursogram [4, p. 109]. An embodiment is generated for each recursive slice on a separate form. The sequence diagram of recursive calls and computations with a dynamic rule(base) for the function fbase1(n) is extremely simple. It is presented in Figure 2. The points of start and end of calculations on the diagram are depicted by an oval. The incarnations of recursive slices are numbered and presented in the figure in the form of rectangles. The incarnations themselves are written out in sufficient detail. Recursive calls and returns from them for organizing deferred calculations are represented by solid curved arrows between the individual forms. The dashed lines indicate the options for completing calculations based on values from a base.

We modify function (3) by reversing the order of the recursive calls fbase1(n-2) and fbase1(n-1). The calculation scheme for the obtained function (4) is presented in Figure 3, where the forms and the relationships between them are arranged in the same way as in Figure 2. The sequence of recursive calls in calculating f (n) according to (4) will correspond to the passage through the tree D, along its left branch from the root f (n) down to f (2).

```
function fbase2(n:integer):int64;
  var a,b:int64;
  begin
    if n>=kthen
     begin
      a:= fbase2(n-1);b:= fbase2(n-2);
      v[n]:=a+b;k:=k+1;
     end;
  result:=v[n];
end;
```
(4)

If we rewrite function (4) in the form (5), then, contrary to expectations, it will not work more slowly. The scheme of recursive calls on it will just exactly coincide with the corresponding call scheme for function (4), and the absence of additional local variables a and b in the calculations will even lead to some decrease in the calculation time. This happens because in the assignment v [n]: = fbase2 (n-1) + fbase2 (n-2), recursive calls are realized only at the expense of

the first term, and when it comes to the second term, its value is in the form v(n-2) is already among the elements of the base in the form of an indicator of completion of calculations.

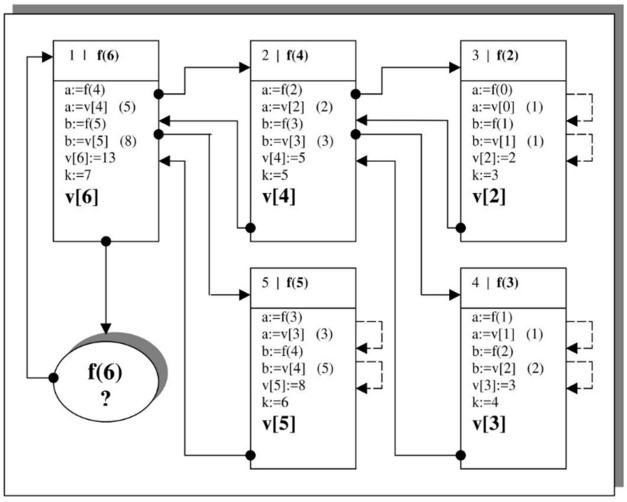

Figure 2. Diagram of recursive calculations with a dynamic rule (base) for the function f (n) = fbasel (n) with n = 6

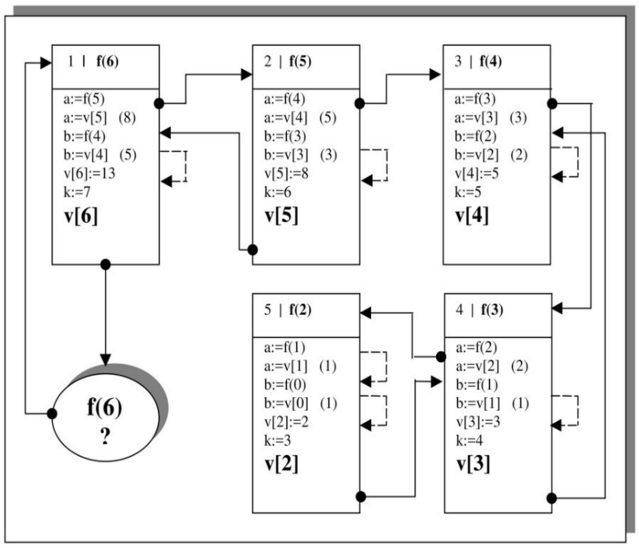

Figure 3. Diagram of recursive calculations with a dynamic rule (base) for the function f (n) = fbase2 (n) with n = 6

```
function fbase2(n:integer):int64;
  begin
    if n>=k then
     begin
       v[n]:= fbase2(n-1)+fbase2(n-2);k:=k+1;
     end;
    result:=v[n];
end;
```
(5)

When working with a dynamic rule(base) β, it is not always possible to get access to the sets of values of its parameters as easily as it was in examples (3) and (4). There, due to the specifics of the problem and the algorithm used, it was actually possible to organize direct access to the required values. In general, it is important not to overload the base with unnecessary elements. With a "large" base, checking for completion in each recursive call can be very laborious. Moreover, the dynamics of the rule (base) implies not only its expansion, but also a possible narrowing. If, for one reason or another, after a specific recursive call, certain sets of values can no longer be used as indicators for completing calculations, it is advisable to remove them from the base.

In the next version of the program-function (6) of calculating f (n), the elements added to the base are not used as indicators of completion of calculations in recursive calls, but only as values in deferred calculations.

```
function fbase2(n:integer):int64;
  var a,b:int64;
  begin
    if n>=k then
     begin
       if n-1<k then a:=v[n-1]else a:=fbase2(n-1);
       if n-2<k then a:=v[n-2]else a:=fbase2(n-2);
       v[n]:=a+b;k:=k+1;
     end;
   result:=v[n];
end;
```
(6)

For end of the given item we needed to formulate and prove the statement about quantity of recursive calls at calculation of value of function Fibinacii under the program (2).

The theorem 1. The quantity k (n) recursive calls at function evaluation Fibonacci f (n) by (2) is equal to the program.

$$k(n) = -1 + \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right]. \quad (7)$$

The proof for function to (п) by obvious image in the following recurrent ratio are carried out:

$$\begin{cases} k(0) = 0, \quad k(1) = 0, \\ k(n) = k(n-1) + k(n-2) + 1 \quad (n = 2, 3, \dots). \end{cases} \quad (8)$$

From here we have:

$$k(2) = 1, \quad k(3) = 2, \quad k(4) = 4, \quad k(5) = 7, \quad \dots.$$

Proceeding from the general theory of linear returnable sequences, it is uneasy to receive at the final formula for calculation *to (n)*. In it also we shall engage. For the further reasoning's it is convenient for us to predetermine *k (n)* for the whole negative values *n,* believing k (n) =0 at n<0. Using K.Aiversona's notation [3], (8) it is possible to copy as:

$$k(n) = k(n-1) + k(n-2) + [n \geqslant 2].$$

Let's remind, that K.Aiverson, the author of the programming language of APL, has entered into it a design of a kind [L], where L a logic condition :

$$[L] = \begin{cases} 1, & L = True \\ 0, & L = False \end{cases}$$

This simple notation turned out to be very useful in transformations and calculations of various sums [4, p. 403-422], [5, p. 50].

Let G(z) be the generating function for k(n). Then

$$G(z) = \sum_n k(n) \cdot z^n = \sum_n k(n-1) \cdot z^n + \sum_n k(n-2) \cdot z^n + \sum_n ([n \geqslant 2]) \cdot z^n =$$
$$= z \cdot G(z) + z^2 \cdot G(z) + \sum_{n \geqslant 2} z^n = z \cdot G(z) + z^2 \cdot G(z) + \frac{z^2}{1-z}$$

And therefore,

$$G(z) = \frac{z^2}{(1-z) \cdot (1-z-z^2)}. \quad (10)$$

Let p (z) and Q(Z) - accordingly numerator and a denominator of the right part (10) and

$$\lambda_1 = 1, \qquad \lambda_2 = \frac{1+\sqrt{5}}{2}, \qquad \lambda_2 = \frac{1-\sqrt{5}}{2}.$$

Then we have:

$$\alpha_k = \frac{-\lambda_k \cdot P\left(\frac{1}{\lambda_k}\right)}{Q'\left(\frac{1}{\lambda_k}\right)} \qquad (k = 1, 2, 3).$$

$$Q(z) = (1-\lambda_1 \cdot z) \cdot (1-\lambda_2 \cdot z) \cdot (1-\lambda_3 \cdot z);$$

$$P\left(\frac{1}{\lambda_1}\right) = 1, \quad P\left(\frac{1}{\lambda_2}\right) = \frac{3-\sqrt{5}}{2}, \quad P\left(\frac{1}{\lambda_3}\right) = \frac{3+\sqrt{5}}{2}; \quad (11)$$

$$Q'\left(\frac{1}{\lambda_1}\right) = 1, \quad Q'\left(\frac{1}{\lambda_2}\right) = \frac{-\sqrt{5} \cdot (3-\sqrt{5})}{2}, \quad Q'\left(\frac{1}{\lambda_3}\right) = \frac{\sqrt{5} \cdot (3+\sqrt{5})}{2}. \quad (12)$$

From Theorem1 on the expansion of the rational function P (z) / Q (z) in a power series in the case of different roots of Q (z) [7 p. 374] for our case we get:

$$k(n) = \alpha_1 \cdot \lambda_1^n + \alpha_2 \cdot \lambda_2^n + \alpha_3 \cdot \lambda_3^n,$$

Where

$$\alpha_k = \frac{-\lambda_k \cdot P\left(\frac{1}{\lambda_k}\right)}{Q'\left(\frac{1}{\lambda_k}\right)} \qquad (k = 1, 2, 3).$$

Using (11) and (12), the last relation we have:

$$\alpha_1 = -1, \qquad \alpha_2 = \frac{1+\sqrt{5}}{2 \cdot \sqrt{5}}, \qquad \alpha_3 = -\frac{1-\sqrt{5}}{2 \cdot \sqrt{5}}.$$

From here

$$k(n) = -1 + \frac{1+\sqrt{5}}{2 \cdot \sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1-\sqrt{5}}{2 \cdot \sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n =$$
$$= -1 + \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right]$$

and (7) it is ***proved***.

Investigation 1. From (7) for and the Bine formula [7, p. 331] implies that

$$k(n) = f(n+1) - 1. \quad (13)$$

Investigation 2. In the binary tree of recursive calls when calculating f(n), there are vertices calculations on f(n).

$$p(n) = 2 \cdot k(n) + 1 = 2 \cdot f(n+1) - 1$$

If formula (13) could be anticipated in one way or another, for example by considering the first few terms of the sequence k(n), then its proof could be carried out by the method of mathematical induction. In this case, the statement of the theorem would follow from the recurrence relations (8) and the Binet formula [7 p. 331].

Indeed, 7 for n = 0 and n = 1, relation (13) is valid:

$$k(0) = 0 = f(1) - 1 = f(0+1) - 1, \qquad k(1) = 0 = f(2) - 1 = f(1+1) - 1.$$

Let (13) be satisfied for n = m-2 and n = m-1 (m> = 2):

$$k(m-2) = f(m-1) - 1, \qquad k(m-1) = f(m) - 1.$$

But in this case

$$k(m) = k(m-1) + k(m-2) + 1 = f(m) - 1 + f(m-1) - 1 + 1 = f(m+1) - 1,$$

That is, the induction is justified and the formula (13) is ***proved***.

Let us now show relation (7). For n = 0 and n = 1 it is valid. Let n> = 2. Applying Binet's formula [7, p. 331] in (13) to f (n + 1), we obtain (7).

3. Returnable Sequences

We shall consider one more example on рекурсию with dynamic rule (base). Let *to* — natural number, **C0, C1..., Ck-1** — real numbers also there is a returnable equation of the order *to* general view:

$$v_n = c_0 \cdot v_{n-1} + c_1 \cdot v_{n-2} + \cdots + c_{k-1} \cdot v_{n-k} \quad (n = k, k+1, \ldots). \tag{14}$$

This equation is a recurrent and generates numerical sequence:

$$v_0, \quad v_1, \quad v_2, \quad \ldots, \quad v_n, \quad \ldots. \tag{15}$$

let's make the recursive program for calculation of the general member (15).

Let files of factors with = (C0, CI are set..., Ck-i) and initial members

$$v = (v_0, v_1, \ldots, v_{k-1}).$$

To write recursive program - function with static base *{V0, VI..., Vk-i}* calculations V_n (n = *to, to* 4-1...) work does not make. Those is, for example, function *rbase (n).* It is supposed, that a variable *to* and files with and *v* are in the field of visibility *rbase (n):*

```
function rbase(n:integer):int64;
  var i:integer; su:int64;
  begin
    if n<k then rbase:=v[n]
    else
      begin
        su:=0; for i:=0 to k-1 do su:=su+c[i]*rbase(n-1-i);
        rbase:=su;
      end;
  end;
```

Calculations on rbase (n) are very laborious even for п> 35 because of the rapidly growing number of recursive calls along with n.

Let's construct analogue for rbase (n) function with dynamic rule (base) rbase1 (n.) we shall count, that a variable k matrix c and v are in the field of visibility rfeasel (n), under v is preserved not less п elements (n>= к), and they are initiated by initial values and zero: v = (v0, v1..., v_{n-1} ,0,0...). Then rbase1 (n) could look as follows:

Originally the base will consist of all nonzero elements of a matrix v. Each recursive call expands base on one nonzero element v. We shall note, that rbasel (n) — already rather effective program function with quantity of recursive calls at n> = to, equal to (п — to + 1). Thus only in п — to + 1 from them calculations are really spent, and in other cases all terminates on values from extending base.

```
function rbase1(n:integer):int64;
  var i:integer;
  begin
    if v[n]=0 then
      for i:=0 to k-1 do
        v[n]:=v[n]+c[i]*rbase1(n-1-i);
        rbase1:=v[n];
      end;
```

Recursive function *rbase2 (n)* with dynamic rule (base) is arranged the same as and *fbase1 (n),* but for it is not required preliminary initiation of a "tail" part of a file *v* by zero. There is enough in the field of visibility *rbase2 (n)* to define initial value of a variable s, equal *to.* The size *s* will serve further the counter of quantity of elements already placed in dynamic rule (base). Quantity of recursive calls at calculation up to *rbase2 (n)* same, as well as at calculation on *rbasel (n).*

```
function rbase2(n:integer):int64;
  var i:integer; su:int64;
  begin
    if n>=s then
      begin
        su:=0;
        for i:=0 to k-1 do su:=su+c[i]*rbase2(n-1-i);
        v[n]:=su;s:=s+1;
      end;
    rbase2:=v[n];
  end;
```

4. Binomial coefficient

Let n and m the non-negative integers numbers 0 < = m < = n . $C_n^m = C(n,m)$ to calculate under the following recurrent formula:

$$C(n,m) = C(n-1,m-1) + C(n-1,m) \qquad (C(k,0) = 1, C(k,k) = 1, \quad k = 0,\dots,n)), \qquad (16)$$

note using operations of multiplication or division and generating known triangle. Direct calculation of coefficient *C(n, m)* on "true" recursive program – function already

```
function C(n,m:integer):int64;
  var i:integer; su:int64;
  begin
    if (m=0) or (n=m) then C:=1
    else C:=C(n-1,m-1)+C(n-1,m);
  end;
```
(17)

at *n*> = 40 becomes rather bulky and consequently it is difficultly feasible for real time. At the same time anything in this case does not interfere with the organization of recursive calculations with dynamic rule (base). Let in the program, where formed function C2 will be located (n, m) calculations of binomial factors, are available definitions:

```
var i,j:integer;
    v:array[0..100] of int64;
```

We initialize a part of a matrix v under base as follows:

```
for j:=0 to m do
  for i:=j to n-m+j do
    if (j<>0) and (i<>j) then v[i,j]:=0
    else v[i,j]:=1;
```
(18)

According to the given fragment initially to dynamic rule (base) nonzero elements of a matrix v [i, j]: v[i,0] = 1 (i = 0..., n- *m); v[j, j]* = 1 *(j* =0...m). Subsequently in a body of function C2 (n, *m)* the base will extend replacement of zero values *v [i, j]* accordingly on calculated values of coefficient *C (i, j) (i* = 0..., n-*m, j =0*,...,m). We shall note the following fact. Generally the matrix *v* should contain (n+1)^2 elements. For accommodation of triangle Pascal it is necessary (n/2+ 1)*(*n*-1) elements. Really at calculation of concrete coefficient *C(n, m)* for it is required base only *(m* + 1). (n- m + 1) elements. It is so much elements also are exposed to initial initialization. Program-function of calculation of binomial coefficients C2(n, m) in this case could be written down so:

```
function C2(n,m:integer):int64;
  begin
    if v[n,m]=0 then
      v[n,m]:=C2(n-1,m)+C2(n-1,m-1);
    C2:=v[n,m];
  end;
```
(19)

The finding of coefficient *C(n, m)* on this recursive program - function calculations it is not spent and required no more (m + 1)*(n-m + 1) recursive references with the same quantity of operations of addition at the postponed calculations. Therefore, an obstacle for carrying out of calculations now can be only a range of allowable integer values. In our case received values should not surpass size 2^{63}-1= 9223372036854775807. On Figure 4 big and small squares designate elements of triangle for *n* = 8. At a finding of coefficient C(8,5) on program - function *C2* to dynamic rule (base) the elements corresponding to the squares are originally attributed. Then, only the elements corresponding to the large blackened squares located in rows from the first to the third in the selected parallelogram and from left to right along the lines are sequentially calculated and added to the base. Small blackened squares correspond to elements of Pascal's triangle that are not involved in the calculation.

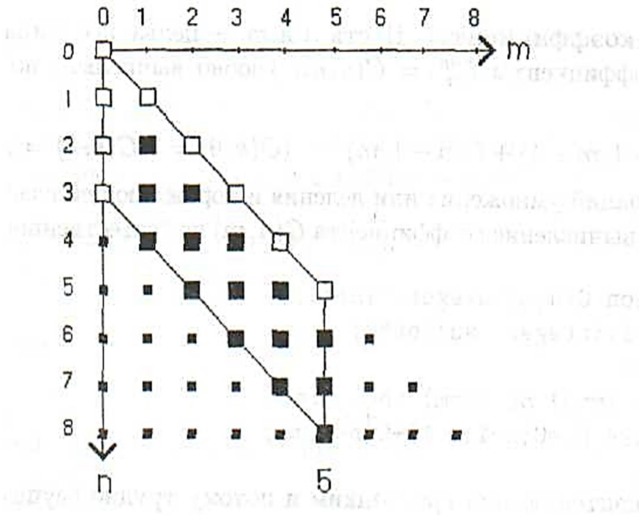

Figure 4. Recursion with dynamic rule (base) at calculation with (8,5) under program *C2*

5. Conclusion

In this paper, the researchers obtained an increase in the speed of the recursive algorithm through a dynamic rule (rule) that changes during the implementation process. The dynamic rule(base) often allows *avoiding repeated calculations* for the same sets of parameter values, which reduces the number of repeated calls and simplifies calculations. A dynamic rule mechanism was implemented using simple, well-known examples to calculate Fibonacci sequences, repeated linear sequences of the general shape, and binomial coefficients.

6. References

[1] Daniel Weibel, (2017, Nov 9). "Recursion and Dynamic Programming"[Online] https://weibeld.net/ algorithms/recursion.html.

[2] Boronenko T.A. The concept of a school course in computer science, St. Petersburg, 1995.

[3] K. E. Iverson, (2020, February 16). *Mathematic* "APL programming language""[Online],https://en.wikipedia.org/wiki/ APL_(programming_language)#Mathematic.

[4] YESAYAN A.R., RECURSION IN COMPUTER SCIENCE, TULA: 2000.

[5] BAUER F.L., HNS R., Hill U. Informatika. Tasks and decisions, Mir, 1978.

[6] Yesayan A.R. Recursion in computer science. Descartes' method. – Tula , 2000.

[7] GRAHAM P., WHIP D., O.Konkretnaja, The bases of computer science, Mir, 1998.

[8] Gutman G., "Realization of recursive algorithms in BASIC", *Informatics and education*, vol. 5 , pp. 56-59, 1989.

[9] G.V.Vanykina, T. O. Sundukova, Algorithms of computer data processing, Tula, 2012

[8] Roberts, Eric S., Thinking Recursively, John Wiley and Sons, 1986.

[9] Eric S., “Thinking Recursively with Java”, *An excellent, intuitive book. Roberts*, John Wiley and Sons, 2006.