Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp187-195)

A Complexity Measure Based on Information Contained in the Software

DHARMENDER SINGH KUSHWAHA and A.K.MISRA
Department of Computer Science and Engineering
Moti Lal Nehru National Institute Of Technology
Allahabad,
INDIA.

Abstract: - Cognitive Informatics is a field that studies internal information processing mechanism of the human brain and its application in software coding and computing. This paper attempts to empirically demonstrate the amount of information contained in software and develops a concept of cognitive information complexity measure based on the information contained in the software. It is found that software with higher cognitive information complexity measure has more information units contained in it. Therefore cognitive information complexity measure can be used to understand the cognitive information complexity and the information coding efficiency of the software. For any complexity measure to be robust, Weyuker properties must be satisfied to qualify as good and comprehensive one. In this paper, an attempt has also been made to evaluate cognitive information complexity measure in terms of nine Weyuker properties, through examples. It has been found that all the nine properties have been satisfied by cognitive information complexity measure and hence establishes cognitive information complexity measure based on information contained in the software as a robust and well-structured one.

Keywords: - Cognitive Informatics, Information Unit, Complexity Information Complexity, Weighted Information Count, Cognitive Information Complexity Unit, Information Coding Efficiency.

1 Introduction

The first obvious question is "What is Complexity?" IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify [2].

Over the years, research on measuring the software complexity has been carried out to understand, what makes computer programs difficult to comprehend. Few measures have shown concern to propose the complexity measures whose computation itself is not complex. A major force behind these efforts is to increase our ability to predict the effort, quality, coding efficiency, cost or all of these. Major Complexity measures of software's that refer to effort, time and memory expended have been used in the form of Halstead's software metric [1], McCabe's cyclomatic complexity [4], Klemola's KLCID Complexity Metric [6], Wang's cognitive functional complexity [10] and many more.

Cognitive complexity measures are still in the nascent stage and it refers to the human effort needed to perform a task or the difficulty experienced in understanding the code or the information packed in it. In cognitive informatics, it has been found that the functional complexity of software is dependent on internal architectural control flows and their inputs and outputs [8, 9]. Cognitive informatics process also takes into account the information contained in a software [11]. Our proposed metric goes a step ahead to model the functional complexity of software based not only on the inputs and outputs but also on the identifiers and operators used in the code. Our measure is based not only on the syntactic aspects of the program but semantically derives cognitive information complexity measure. A fundamental research of cognitive informatics is the difficulty encountered in understanding the software. Therefore we need to answer the following question:

- **What makes software difficult to understand?**

To answer this question, we need to answer another question:

- **Is there a relationship between the above-mentioned question and the cognitive complexity of the software?**

To answer both the above-mentioned questions, we begin with a review of existing software complexity measures in Section 2. Section 3 proposes a new cognitive information complexity measure based on the information contained in the software and answers of the above questions. Section 4 carries out the robustness analysis of our measure using Weyuker properties. Section 5 carries out the comparative study of cognitive information complexity measure and other measures in terms of Weyuker properties. Section 6 concludes our paper.

2 Existing Cognitive Complexity Measures of Software Complexity

2.1 KLCID Complexity Metric

Klemola and Rilling [6] proposed KLCID based cognitive complexity measure in 2004. It defines identifiers as programmers defined labels (variable name, class name, object name etc) and based on this Identifier Density (ID) is defined as :

ID = Total No. of Identifiers / LOC

For calculating KLCID, it finds number of unique lines of code, lines that have same type and kind of operands with same arrangement of operators would be considered equal. Hence (a = b + c) ≈ (d = e + f) when a, b, c, d, e and f are of the same type. It then defines KLCID as:

KLCID = No. Of identifiers in the set of unique lines / No. of unique lines containing identifier

It is noteworthy that the above approach can become very difficult, complex and time consuming when comparing a line of code with each line of the program. It also assumes that internal control structures for the different software's are same.

2.2 Cognitive Functional Complexity (CFS)

Wang and Shao [10] have proposed cognitive functional size to measure the cognitive complexity. The measure defines the cognitive weights for the Basic Control Structures (BCS) as under:

Category	BCS	Weight
Sequence	Sequence (SEQ)	1
Branch	If-Then-Else (ITE)	2
	Case	3
Iteration	For-do	3
	Repeat-until	3
	While-do	3
Embedded Component	Function Call (FC)	2
	Recursion (REC)	3
Concurrency	Parallel (PAR)	4
	Interrupt (INT)	4

Table 2 : Definition of BCS's and their Equivalent Cognitive Weight

Cognitive functional size of a software is defined as :

$$\mathbf{CFS = (N_i + N_o)^* W_c}$$

Where N_i = Number of inputs,
N_o= Number of outputs,
W_c = Total cognitive weight of the software.

Wc is defined as the sum of cognitive weights of its q linear blocks composed in individual BCS's. Since each block may consist of 'm' layers of nesting BCS's, and each layer with 'n' linear BCS, total cognitive weight is defined as:

$$W_c = \sum_{J=1}^{q} \left[\prod_{k=1}^{m} \sum_{I=1}^{n} Wc(j,k.i) \right]$$

Only one sequential structure is considered for a given component.
Although the cognitive functional size is a good measure, it does not provide an insight into the amount of information contained in a software.

3 Cognitive Information Complexity Measure (CICM)

Various theories have been put forward in establishing a clear relationship between a piece of code and its information content. Once we have a measure of the information contained in the software, study of information processing mechanism becomes easier.

Wang [11] demonstrated that software obeys the laws of Informatics and the Cognitive Science based on the following assertions:

- Software represents computational information.
- Software is a mathematical entity.
- Software is the coded solution to a given program.
- Software is a set of behavioral instructions to computer.

Wang [11], further says that Information is the third essence in modeling the natural world supplement to matter and energy. Wang [12] defines software as "Software in cognitive informatics is perceived as formally described design information and implementations instructions of computing application" i.e.

Software ≈ Information

Thus, if software is equivalent to information, it implies that

Difficulty in understanding the software ≈ Difficulty in understanding the information

Hence the cognitive complexity of the software should be based on the measure that takes into account the amount of information contained in the software.

Since software represents computational information and is a mathematical entity, the amount of information contained in the software is a function of the identifiers that hold the information and the operators that perform the operations on the information.

Information = f (Identifiers, Operators)
Identifiers are variable names, defined constants and other labels in a software. Therefore information can be defined as:

Definition 1: Information contained in one line of code is the number of all operators and operands in that line of code. Thus in k_{th} line of code the Information contained is:

$$\mathbf{I_k = (Identifiers + Operands)_k}$$
$$\mathbf{= (ID_k + OP_k)\ IU}$$

Where ID = Total number of identifiers in the k_{th} LOC of software,
OP = Total number of operators in the k_{th} LOC of software,
IU is the Information Unit representing that at least any identifier or operator has one unit information in them.

Definition 2: Total **Information** contained in software (ICS) is sum of information contained in each line of code i.e.

$$\mathbf{ICS = \sum_{k=1}^{LOCS} I_k}$$

Where I_k = Information contained in k_{th} line of code,
LOCS = Total lines of code in the software.

Thus, it is the information contained in the identifiers and the necessary operations carried out by the operators in achieving the desired goal of the software, which makes software difficult to understand. This answers our first question.

Once we have established that software can be comprehended as information defined in information units (IU's), the measure of the complexity of the software should contain the above parameters. Based on this fact weighted information count is been introduced as in definition 3.

Definition 3: The **Weighted Information Count of a line of code (WICL)** of a software is a function of identifiers, operands and LOC and is defined as:

$$\mathbf{WICL_k = ICS_k\ /\ [LOCS - k]}$$

Where WIC_k = Weighted Information Count for the k_{th} line,
ICS_k = Information contained in a software for the k_{th} line.

Therefore The **Weighted Information Count of the Software (WICS)** is defined as :

$$\mathbf{WICS = \sum_{k=1}^{LOCS} WICL_k}$$

In order to be a complete and robust measure, the measure of complexity should also consider the internal control structure of the software. These basic control structures have also been considered as the Newton's law in software engineering [9, 10]. These are a set of fundamental and essential flow control mechanisms that are used for building the logical architectures of software.

Using the above properties, we introduce a new cognitive information complexity measure in definition 4.

Definition 4: The **sum of the cognitive weights of basic control structures (SBCS)** is defined as under: Let W_1, W_2 ….. W_n be the cognitive weights of the basic control structures (as given in Table 2) in the software.

Then $\mathbf{SBCS} = \sum_{i=1}^{n} (W_i)$

Definition 5: **Cognitive Information Complexity Measure (CICM)** is defined as the product of weighted information count of the software (WICS) and sum of the cognitive weights of basic control structures (SBCS) of the software.

CICM = WICS * SBCS

Our complexity measure encompasses all the major parameters that have a bearing on the difficulty in comprehending software or the cognitive complexity of the software. It clearly establishes a relationship between difficulty in understanding software and its cognitive complexity. It introduces a method to measure the amount of information contained in the software thus enabling us to calculate the coding efficiency (E_I) as in definition 5.

Definition 6: Information Coding Efficiency (E_I) of a software is defined as

(E_I) = ICS / LOCS.

The cognitive information complexity is higher for the programs, which have higher information coding efficiency [3]. All the above measures have been illustrated with the help of an example below.

Example 1: An algorithm to calculate the average of a set of numbers as shown below is used to illustrate the application of CICM to measure the complexity.

```
# define N  10
main( )
{
int count;
float sum, average, number;
sum = 0;
count = 0;
while (count < N)
{
    scanf("%f", &number);
    sum = sum + number;
    count = count + 1;
}
average = sum/N;
Printf("N=%d sum = %f"',N,sum);
Printf("average = %f"'average);
}
```

An algorithm to calculate the average of a set of 'n'numbers

Calculation of KLCID complexity measure
Total no. of identifiers in the above program = 18
Total no. of lines of code = 17
ID = 18/17 = 1.05
No. of unique lines containing identifier = 9
No. of identifiers in the set of unique lines = 11
KLCID = 11 / 9 = 1.22

Calculation of CFS
Number of inputs N_i = 1
Number of outputs N_o = 3
BCS(sequence) W1 = 1
BCS(while) W2 = 3
W_c = W1 + W2 = 1+3 = 4
CFS = $(N_i + N_o)$* W_c = (1+3)*4 = 16

Calculation of CICM
LOC = 17
Total no. of identifiers = 18
Total no. of operators = 4
BCS(sequence) W_1 = 1
BCS(while) W_2 = 3
SBCS = $W_1 + W_2$ = 1+3 = 4
WICS = [1/16 + 1/13 + 3/12 + 1/11 + 1/10 + 3/9 + 1/7 + 4/6 + 3/5 + 4/3] = 3.63
CICM = WICS * SBCS = 3.63 * 4 = 14.53

Information Coding Efficiency (E_I) of the above program = 22/17 = 1.29

The cognitive information complexity of the algorithm in fig. 1 is 14.53 CICU (Cognitive Information Complexity unit).

5 Evaluation of Cognitive Information Complexity Measure

Weyuker [7] proposed the nine properties to evaluate any software complexity measure. These properties also evaluate the weakness of a measure in a concrete way and in turn lead to the definition of really good notion of software complexity. With the help of these properties one can determine the most suitable measure among the different available complexity measures [5]. In the following paragraphs, the cognitive information complexity measure has been evaluated against the nine Weyuker properties for establishing itself as a robust and comprehensive measure.

Property 1: $(\exists P)(\exists Q)(|P| \neq |Q|)$ Where P and Q are program body.

This property states that a measure should not rank all programs as equally complex. Now consider the following two examples given in Fig. 1 and Fig. 2 in Appendix I. For the program given in Fig. 1 in Appendix I, there are two control structures: a sequential and a iteration. Thus cognitive weight of these two BCS's is 1 + 3 = 4.
Weighted information count for the above program is as under:
WICS = 3/6 + 1/4 +6/3+4/2 = 4.75
Hence Cognitive information complexity measure (CICM) is:
CICM = WICS * SBCS = 4.75 * 4 = 19.0

For the program given in Fig. 2 in Appendix I there is only one sequential structure and hence the cognitive weight SBCS is 1. WICS for the above program is 2.56. Hence CICM for the above program is 2.56 * 1 = 2.56.
From the complexity measure of the above two programs, it can be seen that the CICM is different for the two programs and hence satisfies this property.

Property 2: Let c be a non-negative number. Then there are only finitely many programs of complexity c.

Calculation of WICS depends on the number of identifiers and operators in a given program statement as well as on the number of statements remaining that very statement in a given program. Also all the programming languages consist of only finite number of BCS's. Therefore CICM cannot rank complexity of all programs as c. Hence CICM holds for this property.

Property 3: There are distinct programs P and Q such that |P| = |Q|.
For the program given in Fig. 3 in Appendix I, the CICM for the program is 19, which is same as that of program in Fig. 1. Therefore this property holds for CICM.

Property 4: $(\exists P)(\exists Q)\ (P \equiv Q\ \&\ |P| \neq |Q|)$

Referring to program illustrated in Fig.1, we have replaced the while loop by the formula "sum = (b+1)*b/2" and have illustrated the same in Fig.2. Both the programs are used to calculate the sum of first n integer. The CICM for both the programs is different, thus establishing this property for CICM.

Property 5: $(\forall P)(\forall Q)(|P| \leq |P;Q|$ and $|Q| \leq |P;Q|)$.

Consider the program body given in Fig.4 in Appendix I: The program body for finding out the factorial of a number consists of one sequential and one branch BCS's. Therefore SBCS = 3. For the program body for finding out the prime number, there are one sequential, one iteration and two branch BCS's. Therefore SBCS = 1+ 2+3+2 = 9. For the main program body for finding out the prime and factorial of the number, there are one sequential, two call and one branch BCS's. Therefore SBCS = (1+2+2+2)+3+9 = 19. WICS for the program is 5.1. Therefore the Cognitive Information Complexity Measure for the above program = 5.1 * 19 = 96.9.

Now consider the program given in Fig.5 in Appendix I to check for prime. There is one sequential, one iteration and three branch BCS's. Therefore SBCS = 1+2+3+2+2 = 10. WICS = 1.85. So CICM = 1.85 * 10 = 18.5.

For the program given in Fig.6 in Appendix I, there is one sequential, one iteration and one branch BCS's . SBCS for this program is 6 and WICS is .5.11. Hence CICM = WICS * SBCS = 5.11 * 6 = 30.66.

It is clear from the above example that if we take the two-program body, one for calculating the factorial and another for checking for prime whose CICM are 18.5 and 30.66 that are less than 96.9. So property 5 also holds for CICM.

Property 6(a) : $(\exists P)(\exists Q)(\exists R)(|P| = |Q|) \& (|P;R| \neq |Q;R|)$
Let P be the program illustrated in Fig.1 and Q is the program illustrated in Fig.3. The CICM of both the programs is 19. Let R be the program illustrated in Fig.6. Appending R to P we have the program illustrated in Fig.7 in Appendix I.
Cognitive weight for the above program is 10 and WICS is 8.3. Therefore CICM = 8.3*10=83.
Similarly appending R to Q we have SBCS = 10 and WICS = 8.925. Therefore CICM = 8.925*10 = 89.25 and $83 \neq 89.25$. This proves that Property 6(a) holds for CICM.

Property 6(b): $(\exists P)(\exists Q)(\exists R)(|P| = |Q|) \& (|R;P| \neq |R:Q|)$

To illustrate the above property let us arbitrarily append three program statements in the programs given in Fig.1, we have the program given in Fig.8 in Appendix I. There is only one sequential and one iteration BCS. Hence cognitive weight is 1 + 3 = 4. There is only one sequential and one iteration BCS. Hence cognitive weight is 1 + 3 = 4 and WICS = 5.58. So CICM = 5.58 * 4 = 22.32.
Similarly appending the same three statements to program in Fig.3 we again have cognitive weights = 4 and WICS = 5.29. Therefore CICM = 21.16 ≠ 22.32. Hence this property also holds for CICM.

Property 7: There are program bodies P and Q such that Q is formed by permuting the order of the statement of P and ($|P| \neq |Q|$).
Since WICS is dependent on the number of operators and operands in a given program statement and the number of statements remaining after this very program statement, hence permuting the order of statement in any program will change the value of WICS. Also cognitive weights of BCS's depend on the sequence of the statement [1]. Hence CICM will be different for the two programs. Thus CICM holds for this property also.

Property 8 : If P is renaming of Q, then |P| = |Q|.

CICM is measured in numeric and naming or renaming of any program has no impact on CICM. Hence CICM holds for this property also.

Property 9: $(\exists P)(\exists Q)(|P| + |Q|) < (|P;Q|)$
OR
$(\exists P)(\exists Q)(\exists R)(|P| + |Q| + |R|) < (|P;Q;R|)$

For the program illustrated in Fig.4, if we separate the main program body P by segregating Q (prime check) and R (factorial), we have the program illustrated in Fig.9 as shown in Appendix I. The above program has one sequential and one branch BCS. Thus cognitive weight is 7 and WICS is 1.475. Therefore CICM = 10.325. Hence 10.325 + 18.5 + 30.66 < 96.9. This proves that CICM also holds for this property.

6 Comparative Study of Cognitive Information Complexity Measure and Other Measures in Terms of Weyuker Properties

In this section cognitive information complexity measure has been compared with other complexity measures in terms of all nine Weyuker's properties.

P.N.- Property Number, S.C.- Statement Count, C N. - Cyclomatic Number, E.M.- Effort Measure, D.C.- Dataflow Complexity, C.C.M. - Cognitive Complexity Measure, CICM – Cognitive Information Complexity Measure, Y- Yes, N – NO

Table l.

P.N.	S.C.	C.N.	E.M	D.C	C.C.M	C.I.C.M.
1	Y	Y	Y	Y	Y	Y
2	Y	N	Y	N	Y	Y
3	Y	Y	Y	Y	Y	Y
4	Y	Y	Y	Y	Y	Y
5	Y	Y	N	N	Y	Y
6	N	N	Y	Y	N	Y
7	N	N	N	Y	Y	Y
8	Y	Y	Y	Y	Y	Y

9	N	N	Y	Y	Y	Y

Table 1: Comparison of complexity measures with Weyuker properties.

It may be observed from the table 1 that complexity of a program using effort measure, data flow measure and Cognitive Information Complexity Measure depend directly on the placement of statement and therefore all these measures hold for property 6 also. All the complexity measure intend to rank all the programs differently

7 Conclusion

This paper has developed the cognitive information complexity measure (CICM) that is based on the amount of information contained in the software. It is a robust method because it encompasses all the major parameters that have a bearing on the difficulty of comprehension or the cognitive complexity of the software. This measure is computationally simple and will aid the developers and practitioners in evaluating the software complexity, which serves both as an analyzer and a predicator in quantitative software engineering. Software quality is defined as the completeness, correctness, consistency, no misinterpretation, and no ambiguity, feasible verifiable in both specification and implementation. For a good complexity measure it is very necessary that the particular complexity measure not only satisfies the above-mentioned property of software quality but also satisfies the nine Weyuker properties. The software complexity in terms of cognitive information complexity measure thus has been established as a well- structured complexity measure.

References:

[1] Halstead, M.H., *Elements of Software Science,* Elsevier North, New York, 1977.

[2] IEEE Computer Society : *IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard* 610.12 – 1990, IEEE.

[3] Kushwaha,D.S and Misra,A.K., "A Modified Cognitive Information Complexity Measure of Software", *ACM SIGSOFT Software Engineering Notes,* Vol. 31, No. 1 January 2006.

[4] McCabe, T.H., A Complexity Measure, *IEEE Transaction on Software Engineering*, SE – 2,6, pp. 308 – 320, 1976

[5] Misra,S and Misra,A.K., Evaluating Cognitive Complexity measure with Weyuker Properties, *Proceeding of the3 rd IEEE International Conference on Cognitive Informatics(ICCI'04)*

[6] Tuomas Klemola and Juergen Rilling, A Cognitive Complexity Metric Based on Category Learning, *IEEE International Conference on Cognitive Informatics,* 2003.

[7] Weyuker, E., Evaluating software complexity measure. *IEEE Transaction on Software Engineering* Vol. 14(9): 1357-1365, september1988.

[8] Wang, Y., The Real-Time Process Algebra (RTPA), *Annals of Software Engineering: An International Journal,* Vol. 14, USA, 2002, pp. 235 - 274.

[9] Wang,Y., On Cognitive Informatics, Keynote Lecture, *Proceedings of IEEE International Conference on Cognitive Informatics,* 2002, pp. 34 - 42

[10] Wang, Y., and Shao, J., Measurement Of The Cognitive Functional Complexity of Software, *IEEE International Conference on Cognitive Informatics,* 2003.

[11] Wang,Y., On The Cognitive Informatics Foundations of Software Engineering, *IEEE International Conference on Cognitive Informatics,* 2004.

[12] Wang,Y., On The Informatics Laws of Software, *IEEE International Conference on Cognitive Informatics*, 2004.

Appendix I

```
/*Calculate the sum of first n integer*/
main() {
int i, n, sum=0;
printf("enter the number");                    //BCS1
scanf("%d" , &n);
for (i=1;i<=n;i++)                             //BCS2
sum=sum+i;
printf("the sum is %d" ,sumssss);
getch();}
```

Fig. 1 : Source code of the sum of first n integers.

```
main()
{
int b;
int sum = 0;
Printf("Enter the Number");
Scanf("%d", &n);
Sum = (b+1)*b/2;
Printf("The sum is %d",sum);
getch();
}
```

Fig. 2 : Source code to calculate sum of first n integers.

```
# define N 10
main( )
{
int count
float, sum,average,number;
sum = count =0;
while (count < N )
{
scanf (" %f",& number);
sum = sum+ number;
count = count+1;
}
average = sum / N;
printf ("Average =%f",average);
}
```

Fig. 3 : Source code to calculate the average of a set of N numbers.

```
#include< stdio.h >
#include< stdlib.h >
int main() {
long fact(int n);
int isprime(int n);
int n;
long int temp;
clrscr();
printf("\n input the number");              //BCS11
scanf("%d",&n);
temp=fact(n);                                //BCS12
{printf("\n is prime");}
int flag1=isprime(n);                        //BCS13
if (flag1= =1)                               //BCS14
else
{printf("\n is not prime")};
printf("\n factorial(n)=%d",temp);
getch();
long fact(int n) {
long int facto=1;                            //BCS21
if (n= =0)                                   //BCS22
facto=1;else
facto=n*fact(n-1);
return(facto); }
int isprime(int n)
{ int flag;                                  //BCS31
if (n= =2)
flag=1;                                      //BCS32
else
for (int i=2;i<n;i++)                        //BCS33
{ if (n%i= =0)                               //BCS34
{ flag=0; Therefore Wc = 3


break; }
else {
flag=1 ;}}
return (flag);}}
```

Fig. 4: Source code to check prime number and to calculate factorial of the number

```
#include< stdio.h >
#include< stdlib.h >
#include< conio.h >
int main() {                                 //BCS1
int flag = 1,n;
clrscr();
printf("\ n enter the number");
scanf("%d",&n);
if (n= =2)
flag=1;                                      //BCS21
else
{for (int i=2;i<n;i++)                       //BCS22
if (n%i= =0)                                 //BCS23
{ flag=0;
```

```
break;}
else{
flag=1;
continue;} }
if(flag)                              //BCS3
printf("the number is prime");
else
printf("the number is not prime");
grtch();}
```

Fig.5 : Source code for checking prime number

```
#include< stdio.h >
#include< stdlib.h >
#include< conio.h >
int main () {
long int fact=1;
int n;
clrscr();
printf("\ input the number");       //BCS1
scanf("%d",&n);
if (n==0)                           //BCS21
else
for(int i=n;i>1;i--)                //BCS22
fact=fact*i;
printf("\n factorial(n)=%1d",fact);
getch();}
```

Fig.6 : Source code for calculating factorial of a number

```
Int main() {
long fact(int n);
int i, n, sum=0;
printf("enter the number");
scanf("%d" , &n);
temp = fact(n);
for (i=1;i<=n;i++)
sum=sum+i;
printf("the sum is %d" ,sum);
getch();
long fact(int n){
long int facto = 1;
if (n == 0)
facto = 1 else
facto = n*fact(n-1);
return(facto);}}
```

Fig.7: Source code of sum of first n integer and factorial of n.

```
main() {
int a,b,result;
result = a/b;
printf(the result is %d",result);
int i, n, sum=0;
printf("enter the number");
scanf("%d" , &n);
for (i=1;i<=n;i++)
sum=sum+i;
printf("the sum is %d" ,sum);
getch();}
```

Fig. 8 : Source code of division and the sum of first n integers.

```
int main(){
int n;
long int temp;
clrscr();
printf("\n input the number");
scanf("%d",&n);
temp = fact(n);
{printf("\ is prime");}
int flag1 = isprime(n);
if (flag1 == 1)
else
{printf("\n is not prime)};
printf("\n factorial(n) = %d",temp);
getch();}
```

Fig.9 : Source code of main program body of program in Fig.4