

CONSIDERING THE DEPTH OF RECURSION IN COMPLEXITY MEASUREMENT

A.E. Okeyinka*
Department of Mathematical Sciences,
Redeemer's University
Redemption City, Ogun state, Nigeria
remi_okeyinka@yahoo.com

O.M. Bamgbola
Department of Mathematics,
University of Ilorin,
Ilorin, Kwara state, Nigeria

***Corresponding Author**

CONSIDERING THE DEPTH OF RECURSION IN COMPLEXITY MEASUREMENT

Abstract

Recursion is exemplified when a function is defined in terms of simpler often smaller versions of itself. The solution to the problem is then devised by combining the solutions obtained from the simpler versions of the problem. However, this implicit computation is often not considered when measuring the complexity of algorithms, thus resulting in incorrect computed complexity. In this study, the depth of recursion in recursive functions is considered and invoked in computing analytically the complexity of algorithms. Our results show that contrary to the traditional belief, a recursive function is more complex than its equivalent non-recursive counterpart.

1. Introduction

Computer Scientists are usually tantalized when told that something cannot be done. Once researchers began to identify problems that are unsolvable in the sense that they have no algorithmic solutions, others began to study these problems to try to understand their complexity. Today, this field of research is a major part of the subject known as recursive function theory, and much has been learned about these super-difficult problems. Indeed, just as mathematicians have developed number systems that reveal quantitative levels beyond infinity, recursive function theorists have uncovered multiple levels of complexity within problems that exist well beyond the capabilities of algorithms. The ability of a function to call itself is called recursion. The use of recursion is particularly convenient for those problems that can be defined in naturally recursive terms, though such problems can be programmed using non-recursive techniques.

2. Formal definition

To define a set of objects recursively, we identify a few core objects as belonging to the set and give rules showing how to build new set of elements from old. More formally, a recursive definition for a set consists of the following two components:

- (i) Base: A statement that certain elements belong to the set.
- (ii) Inductive step: A collection of rules indicating how to form new set of objects from those already known.

A sequence is recursively defined if certain values are specified and later terms of the sequence are defined by relating them to a fixed number of earlier terms.

The following problem posed by Fibonacci in 1902 is a good example of recursive functions:

A single pair of rabbits (male and female) is born at the beginning of a year. Assume the following conditions:

- (i) Rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male and female pair at the end of every month
- (ii) No deaths occur during the year.

Then how many rabbits will there be at the end of the year?

Using recursion we fashion out a solution in terms of how many rabbit pairs there were at the end of previous months.

-----, $k - 4$, $k - 3$, $k - 2$, $k - 1$ up to present month k .

We find that the number of rabbit pairs born at the end of month k is the same as the number of rabbit pairs alive at the end of month $k - 2$. This is because it is exactly the rabbit pairs that were alive at the end of month $k - 2$ that were fertile during month k . The rabbits born at the end of the month $k - 1$ were not. At the end of the year, that is, end of the 12th month, there will be 233 rabbit pairs and so the original pair produced 232 additional pairs.

The following recursive algorithm solves the problem

If $N = 1$ then Fibonumber = 1

Else

If $N = 2$ then Fibonumber = 1

Else

Fibonumber = Fibonumber ($N - 1$) + Fibonumber ($N - 2$)

Other examples of recursive functions include Factorial function, Towers of Hanoi, and Ackermann function. Many recursive functions can be translated to non-recursive functions. An exception is perhaps the Ackermann function which is too complex to evaluate on any but a trivial function. The Ackermann function is a function with two arguments each of which can be assigned any non-negative integer: 0, 1, 2, This function is defined as follows:

(a) If $m = 0$, then $A(m, n) = n + 1$

(b) If $m > 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$

(c) If $m > 0$ and $n > 0$ then $A(m, n) = A(m - 1, A(m, n - 1))$

The value of $A(1, 3)$ is calculated as follows, resulting in 15 steps:

(1) $A(1, 3) = A(0, A(1, 2))$

(2) $A(1, 2) = A(0, A(1, 1))$

(3) $A(1, 1) = A(0, A(1, 0))$

(4) $A(1, 0) = A(0, 1)$

(5) $A(0, 1) = 1 + 1 = 2$

(6) $A(1, 0) = 2$

(7) $A(1, 1) = A(0, 2)$

(8) $A(0, 2) = 2 + 1 = 3$

(9) $A(1, 1) = 3$

(10) $A(1, 2) = A(0, 3)$

(11) $A(0, 3) = 3 + 1 = 4$

(12) $A(1, 2) = 4$

(13) $A(1, 3) = A(0, 4)$

(14) $A(0, 4) = 4 + 1 = 5$

(15) $A(1, 3) = 5$

The forward indentation indicates that we are postponing an evaluation and are recalling the definition, and the backward indentation indicates that we are backtracking.

Generally speaking, the Ackermann function is too complex to evaluate on any but a trivial function. We observe from the case of $A(1, 3)$ how complicated the recursion part of a definition could be.

One example application of recursion is in parsers for programming languages. The great usefulness of recursion is that an infinite set of possible sentences, designs or other data can be defined, parsed or produced by a finite computer program. The main advantage of recursion is usually its simplicity of expression; while the main disadvantage is often that the algorithm may require large amount of memory if the depth of the recursion is very large.

Consequent upon this disadvantage coupled with the seeming simplicity in expressing recursive functions, the results obtained when computing the complexity of recursive functions are often times inaccurate and far less than what the complexity is really.

A common method of simplification is to divide a problem into sub-problems of the same type. This is called divide-and-conquer strategy in programming technique; and is the key to the design of many important algorithms. Divide-and-conquer serves as a top-down approach to problem solving, where problems are solved by solving smaller and smaller instances.

3. The Recursion theorem

In set theory, this is a theorem guaranteeing that recursively defined functions exist. Given a set x , an element a of x and a function $f: X \rightarrow x$, the theorem states that there is a unique function $F: \mathbb{N} \rightarrow X$ (where $\mathbb{N}$ denotes the set of natural numbers including zero) such that

$$F(0) = a$$

$$F(n+1) = f(F(n)) \quad \text{For any natural number } n.$$

Proof of uniqueness

Let us take two functions

$$F: \mathbb{N} \rightarrow X \text{ and}$$

$$G: \mathbb{N} \rightarrow X \text{ such that}$$

$$F(0) = a$$

$$G(0) = a$$

$$F(n+1) = f(F(n))$$

$$G(n+1) = f(G(n))$$

where a is an element of x

It can be proved by mathematical induction that $F(n) = G(n)$ for all natural numbers n

Base case: $F(0) = a = G(0)$ so the equality holds for $n=0$

Inductive step: Suppose $F(k) = G(k)$ for some $k \in \mathbb{N}$ then

$$F(k+1) = f(F(k)) = f(G(k)) = G(k+1)$$

$$\text{Hence } F(k) = G(k) \text{ implies } F(k+1) = G(k+1)$$

$$\text{By induction, } F(n) = G(n) \text{ for all } n \in \mathbb{N}$$

4. Motivation for computing the complexity of recursive procedure

From the definition and illustrations of recursion stated above, we observe the following about recursive procedure:

- (i) It is deceptively simple to model
- (ii) It has the same computational complexity as a simple sequential procedure
- (iii) From (ii), it leads to erroneous result either in measuring the creativity of the programmer or in its contribution to the totality of software complexity.

Arising from these observations, there is need to put recursion in its proper perspective in algorithmic process. And this is what this paper sets out to achieve.

5. The case of the depth of recursion

In the process of algorithm design, the use of recursion in the early stages of the design may be inevitable. Once a recursive algorithm has been validated, the recursion may be removed by translating the algorithm into an equivalent one which uses only iteration. The following recursive functions are considered in this study and the results obtained are presented below: Factorial, Fibonacci, Ackermann, and Towers of Hanoi. Pseudocodes are designed for the functions from both the recursive and iterative perspectives and the Halstead's metrics are computed as shown below. The classical metrics n_1 , n_2 , N_1 , N_2 and their derivatives are computed, tabulated and analysed.

(a) FACTORIAL FUNCTION

Recursive

FACTORIAL (FACT, R)

1. If $R=0$, then: Set FACT: =1, and Return
2. Call FACTORIAL (FACT, R-1).
3. Set FACT: = R *FACT.
4. Return.

Result

$$n_1 = 4 (=, :=, -, *)$$

$$n_2 = 5 (R-1, \text{FACT}, 1, 0, R)$$

$$N_1 = 5$$

$$N_2 = 11$$

$$V = N \log_2 n$$

$$V = 16 \log_2 9$$

$$V = 16 * (\log_{10} 9 / \log_{10} 2)$$

$$V = 16 * (0.954/0.30102)$$

$$V = 16 * (3.169)$$

$$V = 50.704$$

Non-Recursive

FACTORIAL (FACT, R)

1. If $R=0$, then: Set FACT: =1, and Return
2. Set FACT: =1 [Initializes FACT for loop]
3. Repeat for $K= 1$ to R

Set FACT: = K * FACT

[End of loop]

4. Return

Result

$$n_1 = 3 \text{ (}, \text{,}, \text{, } ^*)$$

$$n_2 = 5 \text{ (R, 0, FACT, 1, K)}$$

$$N_1 = 6$$

$$N_2 = 12$$

$$V = N \log_2 n$$

$$V = 18 \log_2 8$$

$$V = 18 * (\log_{10} 8 / \log_{10} 2)$$

$$V = 18 * (0.903 / 0.30102)$$

$$V = 18 * (3)$$

$$V = 5$$

(b) FIBONACCI SEQUENCE

Recursive

FIBONACCI (FIB, F)

1. If $F=0$ or $F=1$, then: Set $FIB := F$, and Return
2. Call FIBONACCI (FIBA, $F-2$).
3. Call FIBONACCI (FIBB, $F-1$).
4. Set $FIB := FIBA + FIBB$.
5. Return.

Result

$$n_1 = 5 \text{ that is (}, \text{,}, \text{, } +, -, \text{ or).}$$

$$n_2 = 9 \text{ that is (FIBA, FIBB, F-2, F-1, FIB, 0, 1, 2, F)}$$

$$N_1 = 8$$

$$N_2 = 17$$

The volume of this procedure is

$$V = N \log_2 n$$

$$V = 25 \log_2 14$$

$$V = 25 * (\log_{10} 14 / \log_{10} 2)$$

$$V = 25 * (1.146 / 0.30102)$$

$$V = 25 * (3.807)$$

$$V = 95.17.$$

Non-Recursive

1. If $F=0$ or $F=1$, then: Set $FIB := F$, and Return.

2. Set $FIBA := 0$.

3. Set $FIBB := 1$.

4. Repeat for $k = 1$ to $F-1$

$TEMP := FIBB$

$FIBB := FIBA + TEMP$

$FIBA := TEMP$

5. $FIB := FIBB$

6. Return

Result

$n_1 = 5$ ($=, :=, +, -, \text{ or}$)

$n_2 = 9$ ($FIBA, FIBB, K, TEMP, FIB, 0, 1, 2, F$)

$N_1 = 13$

$N_2 = 24$

So the volume is

$$V = N \log_2 n$$

$$V = 37 \log_2 14$$

$$V = 37 * (\log_{10} 14 / \log_{10} 2)$$

$$N = 37 * (1.146 / 0.30102)$$

$$V = 37 \cdot (3.807)$$

$$V = 140,85$$

(c) ACKERMANN FUNCTION

Recursive

ACKERMANN (A,m,c)

1. If $m = 0$, then: Set $A(m, c) \leftarrow c + 1$ and Return
2. If $c = 0$ and $m \neq 0$ then: Call ACKERMANN ($A(m, c), (A(m-1, 1))$) and Return
3. If $m \neq 0$ and $c \neq 0$ then: Call ACKERMANN ($A(m, c), (A((m-1), \text{ACKERMANN}(m, c-1)))$)
4. Return.

Result

$$n_1 = 6 \text{ (, , -, +, /, and)}$$

$$n_2 = 7(m, 0, \Lambda, 1, c, m-1, c-1)$$

$$N_1 = 12$$

$N_2 = 34$

$$V_{\text{max}} = N \log_2 n$$

$$V = 46 * (\log_{10} 13 / \log_{10} 2)$$

$$V = 46 * (1.1139 / 0.30102)$$

$$V = 46 * 3.7004$$

$$V = 170.218$$

Non-Recursive

INTEGER FUNCTION Q (m, c)

INTEGER m, c, VAL (m, c)

IF (m,LT,1,OR,c,LT,1) GO TO 70

DO 60 K=1, m

DO 50 L=1, c

IF (K .EQ. 1. OR. L. EQ. 1) GOTO 40

IF (K-L) 10, 20, 30

10 VAL (K, L) = VAL (K, K)

GO TO 50

20 VAL (K, L) = 1 + VAL (K, K-1)

GO TO 50

30 VAL (K, L) = VAL (K, L-1) + VAL (K-L, L)

GO TO 50

40 VAL (K, L) =1

50 CONTINUE

60 CONTINUE

Q = VAL (m, c)

RETURN

70 WRITE (6, 1000) m, c

1000 FORMAT ("IMPROPER ARGUMENTS")

RETURN

END

Result

$n_1 = 6$ (LT, OR, =, -, +, EQ)

$n_2 = 9$ (m, c, K, L, K-L, K-1, L-1, Q, VAL)

$N_1 = 19$

$N_2 = 54$

$V = N \log_2 n$

$V = 73 * (\log_{10} 15 / \log_{10} 2)$

$V = 73 * (1.176 / 0.30102)$

$$V = 73 * 3.906$$

$$V = 285.138$$

(d) TOWERS OF HANOI

Recursive

TOWER (T, BEG, AUX, END)

1. If $T = 1$, then:

(a) Write: $BEG \rightarrow END$.

(b) Return.

[End of if structure]

2. [Move $T-1$, disks from peg BEG to peg AUX]

Call TOWER ($T-1$, BEG, END, AUX).

3. Write: $BEG \rightarrow END$.

4. [Move $T-1$ disks from peg AUX to peg END.]

Call TOWER ($T-1$, AUX, BEG, END).

5. Return.

Result

$$n_1 = 2(=, -)$$

$$n_2 = 6(=, \text{END, AUX, BEG, } T-1, T, 1)$$

$$N_1 = 3$$

$$N_2 = 14$$

$$V = N \log_2 n$$

$$V = 17 \log_2 8$$

$$V = 17 * (\log_{10} 8 / \log_{10} 2)$$

$$V = 17 * (0.903 / 0.30102)$$

$$V = 17 * (3)$$

V=51

Non-Recursive

0. TOWER (T, BEG, AUX, END)

1. Set TOP: = NULL

If T=1, then:

Write: BEG $\rightarrow$ END.

GO to step5

[End of if structure]

2. [Translation of "Call TOWER (T-1, BEG, END, AUX)."]

[Push current values and new return address onto stacks.]

Set TOP := TOP +1.

Set STN[TOP] := T, STBEG[TOP]:=BEG,

STAUX[TOP]:= AUX, STEND[TOP]:=END,

STADD[TOP]:= 3.

[Reset parameters.]

Set T:= T-1, BEG :=BEG, AUX:=END,END := AUX.

Go to step 1.

3. Write : BEG $\rightarrow$ END.

4. [Translation of "Call TOWER(T-1, AUX,BEG,END)."]

[Push current values and new return address onto stacks.]

Set TOP := TOP +1.

Set STN[TOP] := T, STBEG[TOP]:=BEG,

STAUX[TOP]:= AUX, STEND[TOP]:=END,

STADD[TOP]:= 5.

[Reset parameter.]

Set T:= T-1, BEG :=AUX, AUX:= BEG, END := END.

Go to step 1.

5.[Translation of "Return."]

If TOP := NULL, then: Return.

[Restore top values on stacks.]

Set T:= STN[TOP], BEG:= STBEG [TOP],

AUX:= STAUX[TOP] , STEND[TOP],

ADD := STADD[TOP].

Set TOP := TOP -1.

Go to Step ADD.

Result

$n_1 = 4(=, :=, -, +)$

$n_2 = 15$ (BEG,AUX,END,ADD, T,TOP,5,1,3,NULL,STN,STAUX,STEND,STADD,STBEG)

$N_1=31$

$N_2= 61$

$V = N \log_2 n$

$V=92 \log_2 19$

$V=92 *(\log_{10} 19/\log_{10} 2)$

$V=92*(1.278/0.30102)$

$V=92*(4.246)$

$V=390.632$

Table 1 summarizes the above.

Table 1: Analysis of Results of Recursive and Non – Recursive Functions

	Recursive	Non Recursive
Factorial	50.704	54
Fibonacci	95.17	140.85
Ackermann	170.218	285.138
Towers of Hanoi	51	390.632

The complexities for both the recursive and non-recursive versions of the functions are compiled using Halstead's metrics as shown above. The computation shows that in all cases, recursive computations are more efficient. However, this conclusion is erroneous as it does not take into consideration the complexity of the translation process from recursion to non-recursion. The process of translating recursion to iteration entails:

- creation of linked list, and the use of stack or
- the use of array.

Apparently, the complexity of the process is regarded as negligible by the traditional Halstead. However, it is not true that the complexity is negligible, as will be shown in the subsequent sections where the translation algorithm is presented, and complexity of recursion computed.

6.1 Translating Recursive to Non-Recursive Procedure

Recursion can always be completely removed; compilers do so in converting to assembly language. However, doing so can be quite tedious. The general strategy requires using a stack.

Although non-recursive procedures are certainly faster than equivalent recursive procedures, the speed advantage rarely justifies the lack of clarity that results from removing the recursion.

Two methods are usually employed in translating recursion to non-recursion. These are the use of linked list and the use of array structures. The translation algorithm is presented below in 6.2

6.2 Translation Algorithm

There are standard algorithms for translating recursion to iteration. Most of them make use of pushdown stack, and the following is one of them. Once an algorithm has been translated to iterative, then implementing it becomes clearer.

- Step1 At the beginning of the procedure, code is inserted which declares a stack and initializes it to be empty. In most general case, the stack will be used to hold the values of parameters, local variables and return address for each recursive call.
- Step2 The label L1 is attached to the first executable statement now, each Recursive call is replaced by a set of instructions which do the following:
- Step3 Store the values of all parameters and local variables in the stack. The pointer to the top of the stack can be treated as global.

- Step4 Create the i th new label, L_i , and store i in the stack. The value i of this label will be used to compute the return address. The label is placed in the program as described in step7.
- Step5 Evaluate the argument of this call (they may be expressions) and assign these values to the appropriate formal parameters.
- Step6 Insert an unconditional branch to the beginning of the procedure.
- Step7 If this procedure is a function, attach the label created in step4 to a statement which retrieves the function value from the top of the stack. Add code to use this value in the way described in the recursive procedure. If this procedure is not a function then affix the label created in step4 to the statement immediately following the branch of step6. These steps are sufficient to remove all recursive calls in a procedure. We must now alter all return statements in the following way. In place of each return, do the following:
- Step8 If the stack is empty then execute a normal return.
- Step9 Otherwise take the current values of all output parameters and assign these values to the corresponding variables which are in the top of the stack.
- Step10 Now insert code which removes the index of the return address from the stack if one has been placed there. Assign this address to some unused variable.
- Step11 Remove the stack values of all local variables and parameters and assign them to their corresponding variables.
- Step12 If this is a function, insert instructions to evaluate the expression immediately following return and store the result in the top of the stack.
- Step13 Use the index of the label of the return address to execute a branch of that label.

By following these rules carefully, one can take any recursive program and produce a program which works in exactly the same way, yet which uses only iteration to control the flow of the program. On many compilers, this resultant program will be much more efficient than its recursive version. On other compilers the times may be fairly close. Once the transformation to iterative form has been accomplished, one can often simplify the program even further thereby producing even more gains in efficiency.

7. Theorem and Proof

From the translation algorithm presented above, the complexity of a recursive procedure is derived in form of the Theorem given below. For the purpose of proving the Theorem, operators are defined as "+", "-", "*", "=", and so on, while operands are literal expressions, constants, and variables upon which the operators act.

7.1 Theorem

The abstract complexity of a recursive procedure is the sum of the complexities of the body of the procedure and that of the stack operations.

7.2 Proof

Pushing and popping are the 2 basic stack operations. The complexity of pushing and popping is computed as follows:

Pushing operation:

- Increment stack pointer i.e. $\text{ptr} := \text{ptr} + 1$;
- Save item i.e. $\text{stack}(\text{ptr}) := \text{item}$

Popping operation:

- retrieve item i.e. $\text{item} := \text{stack}(\text{ptr})$;
- decrement pointer i.e. $\text{ptr} := \text{ptr} - 1$;

Therefore we have (for pushing and popping)

- distinct operands : = 4;
- distinct operators : = 3;
- total operators : = 4
- total operands : = 8;

$$\Rightarrow n = (\text{distinct operands} + \text{distinct operators}) (\text{variables} + \text{parameters} + 2) \\ = 7 (\text{variables} + \text{parameters} + 2)$$

where 2 is the complexity of the: function value, and return address.

$$\text{and } N = (\text{total operators} + \text{total operands}) (\text{variables} + \text{parameters} + 2) \\ = 12(\text{variables} + \text{parameters} + 2)$$

Assuming complexity of pushing is equal to the complexity of popping then:
the complexity (recursive procedure) := complexity (sequential statements) +
 $2(\text{complexity}(\text{pushing}) \times \text{variables} + \text{parameters} + 2)$

Assuming,

- $C(R)$:= complexity of recursive procedure
- $C(S)$:= complexity of sequential statements in the recursive procedure
- $C(K\alpha)$:= complexity of stack operations in the recursive procedure
- α := variables + parameters + 2

then,

$$C(R) = C(S) + 2(C(K\alpha))$$

8. Conclusion

Halstead complexity metrics are a means of determining a quantitative measure of complexity directly from the operators and operands in the module or procedure. However, in doing so the traditional Halstead regarded the complexity of sequential procedure, and that of the recursive procedure to be equal. In other words, the depth of recursion was not taken into consideration while computing the complexity of recursion. In this paper, the depth of recursion has been considered. This is achieved by translating recursion into iteration using stack data structure. This consideration

leads to a Theorem whose Proof shows that the complexity of a recursive procedure is indeed greater than that of the equivalent sequential procedure by $2(C(Ka))$.