International Journal of Soft Computing 3 (1): 9-12, 2008
ISSN: 1816-9503
© Medwell Journals, 2008

An Efficient Algorithm for the Solution of Ackermann Function

Parviz Rashidian
Department of Mathematics, University of Kurdistan, Sanandaj, Iran

Abstract: One of the main problems for running recursive functions with high rate of growth is execution time and memory spaces, in which case we can not run the program. In addition many commonly used programming languages do not allow recursive programs. This study presents an efficient algorithm for Ackermann recursive function. The algorithm is tested by comparing with recursive function in high level language such as Pascal.

Key words: Recursive function, execution time, procedure, memory, Ackermann function

INTRODUCTION

The Ackermanns function is an excellent mathematical example to a recursive process, in which a short term goals must be accomplished prior to attainment of a long term goal. Recursive process in mathematics is requiring the repetition of the basic function with its succeeding output values until a primitive is obtained. The primitive is then resolved and driven back through the basic function until the answer for the original variables is attained. If the function is truly recursive, as is Ackermanns function, any values for the variables in the function will eventually result in a simple valued answer, this requires that the recursive function have a primitive and that the function converges to a solution after finite number of recursions. The Ackermanns function is defined recursively for non negative integers m, n as follows:

$$\begin{aligned} &A(0,n) = n+1 \\ &A(m,0) = A(m-1,1), \text{if } m>0, \\ &A(m,n) = A(m-1,A(m,n-1)), \text{if } m,n>0 \end{aligned} \tag{1}$$

Equation 1 is the primitive equation and when resolved gives a simple value (Brassard and Bratley, 1990). The special properties of the Ackermanns function are consequence of its phenomenal rate of growth, which the value of:

$$A(0,0) = 1,\quad A(1,1) = 3,$$
$$A(2,2) = 7,\quad A(3,3) = 61,$$

And A(4,4) is greater than 10^{19199} (Harry, 1987; Seymour, 1988). The Ackermanns function plays a major role in computer science and computational complexity theory. A more recent use of Ackermanns function as a compiler benchmark is in computer language shootout which compares the time required to evaluate this function for fixed arguments in many different programming language implementations (Gento, 2006; Benchmarks, 2005).

Table 1: values of m and n

m	n
0	1-100
1	0-100
2	0-100
3	1-11

For the purpose of this study, the Table 1, showes that the values of m and n used in the analysis.

MATERIALS AND METHODS

The approach used in this analysis is to code a Fortran program resolving Ackermann's function for various values of m and n, the program constructed with table (Look-up table) and without table the program searches the table and extracts previously calculated values of A(m,n) instead of recalculating each value, comparison of a program using the table and program without the table and the Pascal recursive function, gives an idea of the difference between the necessary recursion and execution times. Recursive procedures are not handled in Fortran 77 due to the inability of a Fortran routine, to call itself. However, this problem can be accomplished using operating system stacking capabilities in failure. Each time the ackermanns function stores recursive all variables as well as the return address in a last in first out (lifo) data type list.

To keep track of of changing values of the variables during the recursions, a pointer or address keeper must be used to indicate the variables locations in the stack.

The pointer is incremented to pop them. In addition to a pointer (called t in the program), the number of elements in the satack and the number of steps or recursions performed in the program and execution time are calculated by changing m and n (Fig. 1 and 2).

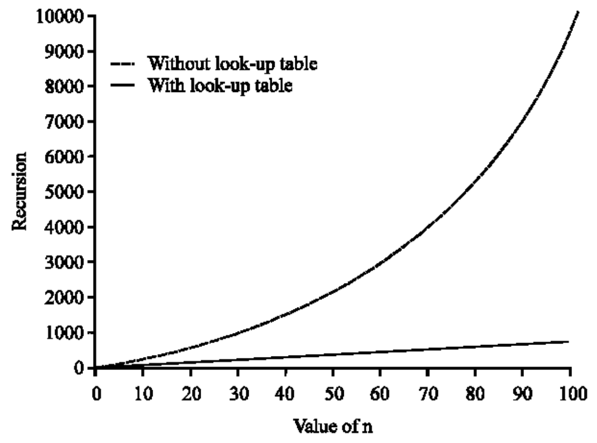

Fig. 1: Recursions required to evaluated A(2, n)

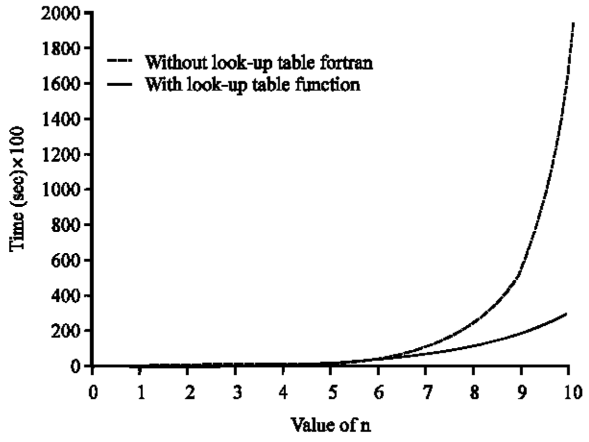

Fig. 3: Time required to evaluated A(3, n)

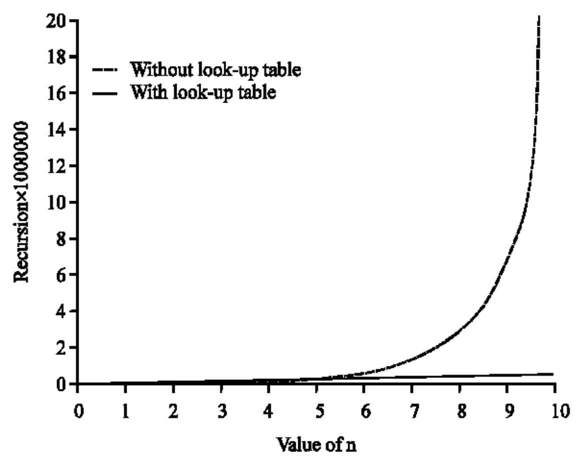

Fig. 2: Recursions required to evaluated A (3, n)

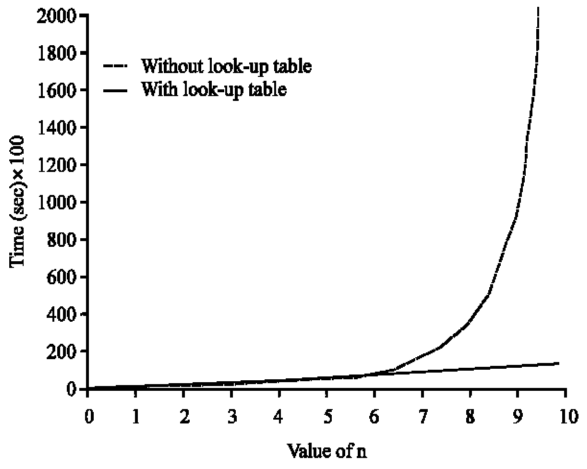

Fig. 4: Time required to evaluated A(3,n) Fortran program

RESULTS AND DISCUSSION

The values of the Ackermanns function using the Fortran program versus its arguments are plotted in Fig. 3 and 4. As can be seen in the figures, the function increases extremely fast when m is greater than 3 and n is increasing.

The look-up table was found to be beneficial when m and n are increasing, Table 2, showes that the result of problem along with the values of m and n that can be used. The number of recursions required to solve the function both with and without a look-up table are also recorded. The number of recursions with and without table are plotted in Fig. 3 and 4, respectively. We will notice that considerable decreasing the number of recursions reduce the problem to one we can solve very quickly.

Procedure 1: Ackermann function without look-up table. Array stm( ), stn( ):

```
•  Put m, n set t = 0, istep = 0
•  If m = 0 then, goto 7
•  If n > 0 then, goto 5
•  Set t = t + 1, st = t, istep = istep + 1
   Set stm(t) = m, stn(t) = n, m = m - 1, n = 1, goto 2
•  Set t = t + 1, st = t, istep = istep + 1
•  Set stm(t) = m, stn(t) = n, n = n - 1, goto 3
•  n = n + 1
•  t = t - 1
   If t <= 0 then, goto 9
   If stm(t) = 1 then, goto 7
   If stn(t) = 0 then, goto 8
   m = stm(t) - 1 then, goto 6
•  Write st, ist ep, m, n
•  Return.
```

Procedure 2: Ackermann function with look-up table. Array stm( ), stn( ), table(,):

```
•  Input m, n set t = 0, istep = 0, table = 0
•  If m = 0 then, goto 6
•  If n > 0 then, goto 4
•  Set stm(t) = m, stn(t) = n
   Set t = t + 1, st = t, istep = istep + 1
   Set m = m - 1, n = 1, goto 2
•  Set t = t + 1, st = t, istep = istep + 1
   If table(m + 1 , n + 1) <= 1 goto 8
•  Set stm(t) = m, stn(t) = n, n = n - 1, goto 3
•  Table(m + 1 , n + 1) = n + 1
•  n = n + 1
•  t = t - 1
   If t <= 0 then, goto 10
   If stn(t) = 0 then, table(stm(t) + 1 , 1) = n, goto 8
   If stm(t) = 1 then, goto 10
   m = stm(t) - 1, goto 5
•  Table(stm(t) + 1 , stn(t) + 1) = n + 1, goto 7
•  Write st, istep, m, n
•  Return.
```

This study provided a procedure for the solution of Ackermanns function and afforded an approach to recursive routines. Using this technique of programming reqires that should be taken care in keeping up with locations of the various elements of the stacks so that they may be recalled later. Using procedure 2, result in considerable decreasing execution time and memory space necessary. Computing A(l,n) takes linear time in n and A(2,n) without a look-up table requires quadratic time, but using look-up table almost takes linear time in n. Also computing A(3,n) with Pascal recursive function takes

Table 2: Result of Fortran and pascal programs

m	n	Ackermann function	Recur (Fortran) with table	Recur (Fortran) without table	Recur (Pascal) using recursive function
0	0	1	1	1	1
0	5	6	1	1	1
0	10	11	1	1	1
0	20	21	1	1	1
0	40	41	1	1	1
0	60	61	1	1	1
0	80	81	1	1	1
0	100	101	1	1	1
1	0	2	2	2	2
1	5	7	7	7	12
1	10	12	12	12	22
1	20	22	22	22	42
1	40	42	42	42	82
1	60	61	61	62	122
1	80	82	82	82	162
1	100	102	102	102	202
2	0	3	4	4	5
2	5	13	19	44	90
2	10	23	34	134	275
2	20	43	64	464	945
2	30	63	94	994	2015
2	40	83	124	1724	3484
2	50	103	154	2654	5355
2	60	123	184	3784	7625
2	70	143	214	514	10295
2	80	163	244	6644	13365
2	100	213	274	11344	207705

Table 3: Result of Fortran programs

m	n	Ackermann function	Recuresive (Fortran) with table	Recuresive (Fortran) without table
3	0	5	4	9
3	1	13	24	52
3	2	29	60	263
3	3	61	136	1194
3	4	125	292	5101
3	5	253	603	21104
3	6	509	1244	85875
3	7	1021	2520	346486
3	8	2045	5076	1391993
3	9	4093	10192	5580156
3	10	8189	20428	22345087
3	11	16381	40904	89429378
3	12	32765	82984	357811565

Table 4: Result of fortran and pascal programs (times in seconds)

m	n	Recuresive (Fortran) with table	Recuresive (Fortran) without table	Recuresive (Pascal) using recursive function
3	5	0.10	0.16	0.10
3	6	0.11	0.66	0.28
3	7	0.12	2.58	1.12
3	8	0.17	10.32	3.03
3	9	0.22	41.36	12.14
3	10	0.39	166.09	46.20
3	11	0.71	0.66	185.26

exponential time in n, but using a look-up table, is almost requires quadratic time.We found that using a look-up table to store the values of the function as calculated will save significant amount of time (Table 3 and 4).

CONCLUSION

The study lead us to conclude that another language having the ability to automatically keep up with stack would be much better suited to recursive routines.

REFERENCES

Benchmarks, X.G.C., 2005.

Brassard, G. and P. Bratley, 1990. Algorithmics Theory and Practice. Prientice-Hall.

Gento, 2004. Intel Pentium 4 computer language shootout.

Harry, F.S., 1987. Data Structures From and Functions. Academic Press.

Seymoure, L., 1988. Data Structuers. McGraw-Hill.