**September 2014**

CSC 342 – Exploration 2

James Francis/Sam Jentsch

Account Numbers: CSC342107/CSC342111

08

**Fall**

­­

Table of Contents

Problem Statement 3

Algorithm 3

Implementation 4

Java Implementation 4

C Implementation 4

Experiment 5

Machines 5

Timing Methods 5

Data Files and Sample Output 6

Charts and Timing Results 8

Conclusions 12

Future Work 13

Problems Encountered 13

Time Chart 13

Appendix A – C Implementation Listing 14

Appendix B – Java Implementation Listing 15

Appendix C – References 16

# Problem Statement

The Ackermann function was designed by Wilhelm Ackermann in the 1920’s to provide a counterexample to the argument that all total computable functions are primitive recursive. The Ackermann function illustrates that although all primitive recursive functions are total computable, not every total computable function is primitive recursive.

Here, primitive recursive means that the function can be implemented iteratively, a total function is a function that is defined for all possible input values, and a computable function is an idea postulated by the Church-Turing Thesis providing a theoretical method for evaluating algorithms. A function is computable if it can be calculated given unlimited amounts of time and space.

The Ackermann function grows faster than an exponential function. It is categorized as being super exponential, a behavior now identified with functions that must be implemented recursively.

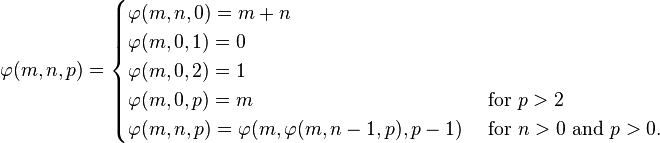
The Ackermann function can be categorized as computable (even if for large values it takes a seemingly infinite amount of time), because every time the parameters are modified, they decrease. Therefore, the Ackermann function will eventually reach its base case and terminate, which implies it is a computable function.

# Algorithm

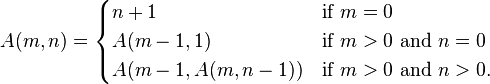
The Ackermann function was originally presented in Ackermann’s original publication as a 3-argument function. The more modern implementation of the function is a 2-argument implementation known as the Ackermann-Péter function.

The original 3-argument function is defined as follows:

Let m, n, and p be nonnegative integers.



The 2-argument Ackermann-Péter function is defined as follows:



The 2-argument and 3-argument functions are essentially equivalent and the 2-argument variant is widely used for testing and demonstrations today.

# Implementation

### Java Implementation

The Java implementation of the Ackermann function we used for testing is an implementation found from Princeton University. It is an implementation of the 2-argument Ackermann-Péter function and was used to generate data on the Unix machine and a Mac. No issues were encountered during the testing of the Java implementation of the program.

### C Implementation

In order to gather a well-rounded data set we wanted to run the Ackermann-Péter function in a language that could be used on the AXP, Unix, and personal computers. Originally JavaScript was attempted. While preliminary results were interesting on a PC, for JavaScript, we were unable to use the JavaScript program on the AXP due to language limitations; Node.js was used for the PC and was unavailable on the AXP. Java was looked at next, however the most recent version of Java available for OpenVMS is 6.0-5, which was last updated in November of 2013, and is not currently installed on the AXP. Due to the previously stated issues we decided on C. We implemented the 2-argument Ackermann-Péter function.

There was a slight learning curve in using the AXP, this was mostly due to an unfamiliarity with the syntax used to navigate the system. One curious point is that by default the AXP does not overwrite files when using the EDIT command. When a file is opened, altered, and finally saved, a new version of that file is created. This added to the testing time of the function on the AXP. Another point of note was passing command line arguments on the AXP. Unlike Unix simply typing in command line arguments after the program name at the command prompt results in an invalid statement on the AXP. With this in mind we had the program prompt the user for integer values to run the Ackermann-Péter function with.

# Experiment

### Machines

The machines used for testing include:

* **15” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2GHz quad-core Intel core i7
* **13” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2.4 GHz Intel Dual Core i5
* **SFA Unix Server**
  + OS: Linux 2.6.18-371.12.1.el5PAE i686
  + MEMORY: Undetermined (required sudo access)
  + PROCESSOR: eight-core Intel(R) Xeon(TM) MP CPU 3.66GHz

### Timing Methods

In the Java implementation, System.nanoTime() was utilized to find the system time before execution of the Ackermann function. The system time was found in the same manner after the completion of the Ackermann function’s execution and subtracted from the start time. This method follows the timing method for Java described in the timing handout.

Timing was achieved in the C implementation of the program through the use of the <time.h> header. The system clock time was found before execution of the Ackermann function and the end time was subtracted from the start time to calculate the total execution time.

For testing Ackermann function calls that resulted in a stack overflow, the Unix “time” command was used to track the time it took for the stack overflow to occur. This method was used for both the Java and C implementations of the Ackermann function on Unix and OS X. A similar but slightly different method was used on the AXP system.

Timing the C implementation, on the AXP was very similar to timing on the Unix machine. <time.h> was used to provide a millisecond breakdown of the function’s run time when the function was able to reach its base case and return a value. However, when the function caused an: SYSTEM-F- EXCPUTIM, CPU TIME LIMIT EXPIRED, error causing the program to crash, there was a slight difficulty in gaining reliable data on run time. Some research what done, and the AXP command SHOW STATUS provided usable information. The SHOW command returned the total CPU Usage in time. After trying the SHOW command on several occasions it became apparent that only login actions, system commands, and program executions cause the CPU Usage timer to increase. The SHOW command was entered prior to running the program, and when the program terminated the command was run again, then the difference was taken and converted to milliseconds. This method was utilized to time test cases where the CPU time limit was exceeded.

### Data Files and Sample Output

Our sample data consisted of manually entering Ackermann function parameters as either command line arguments or user input to the program. We tested and recorded the timing results for the Ackermann function for the following parameters:

**Sample Input: Ackermann(M,N)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| M/N | M = 0 | M = 1 | M = 2 | M = 3 | M = 4 |
| N = 0 | **Ack(0,0)** | **Ack(1,0)** | **Ack(2,0)** | **Ack(3,0)** | **Ack(4,0)** |
| N = 1 | **Ack(0,1)** | **Ack(1,1)** | **Ack(2,1)** | **Ack(3,1)** | **Ack(4,1)** |
| N = 2 | **Ack(0,2)** | **Ack(1,2)** | **Ack(2,2)** | **Ack(3,2)** | **Ack(4,2)** |
| N = 3 | **Ack(0,3)** | **Ack(1,3)** | **Ack(2,3)** | **Ack(3,3)** | **Ack(4,3)** |
| N = 4 | **Ack(0,4)** | **Ack(1,4)** | **Ack(2,4)** | **Ack(3,4)** | **Ack(4,4)** |
| N = 5 | **Ack(0,5)** | **Ack(1,5)** | **Ack(2,5)** | **Ack(3,5)** | **Ack(4,5)** |
| N = 6 | **Ack(0,6)** | **Ack(1,6)** | **Ack(2,6)** | **Ack(3,6)** | **Ack(4,6)** |
| N = 7 | **Ack(0,7)** | **Ack(1,7)** | **Ack(2,7)** | **Ack(3,7)** |  |
| N = 8 | **Ack(0,8)** | **Ack(1,8)** | **Ack(2,8)** | **Ack(3,8)** |  |
| N = 9 | **Ack(0,9)** | **Ack(1,9)** | **Ack(2,9)** | **Ack(3,9)** |  |
| N = 10 | **Ack(0,10)** | **Ack(1,10)** | **Ack(2,10)** | **Ack(3,10)** |  |
| N = 11 | **Ack(0,11)** | **Ack(1,11)** | **Ack(2,11)** | **Ack(3,11)** |  |
| N = 12 | **Ack(0,12)** | **Ack(1,12)** | **Ack(2,12)** | **Ack(3,12)** |  |

Values greater than Ackermann(4,6) were not calculated as anything above Ackermann(4,1) resulted in either a SEG FAULT, stack overflow error, or exceeding available CPU time. Ackermann(4,2) through Ackermann(4,6) were tested in order to find the time it took for the systems to run out of stack space.

Our output consisted of the result of the Ackermann function and the time it took to complete. The result of the function was either found or an error was thrown as a result of the program running out of call stack space.

### Charts and Timing Results

UNIX COMPARISONS

Charts

MAC COMPARISONS

# Conclusions

  For each of the 3 systems tested, testing Ackermann(4,2) resulted in the program crashing. These crashes appear to be the result of the programs running out of available call stack space.

  The Java version of the implementation threw a StackOverflowError, which the Java documentation states is a result of the Java Virtual Machine running out of stack space for a thread, typically because the thread is performing an unbounded number of recursive invocations. In this case, the enormous number of recursive invocations associated with the Ackermann function is the cause of this exception.

  The C version of the implementation also resulted in a seg-fault. Like the Java implementation, we assume this is the result of the program running out of call stack space due to the amount of recursive calls generated by the Ackermann function.

  Some research into how stacks operate indicated that stack space is set aside as contiguous memory space for a program thread *before* it begins executing. This means that there is a limited amount of stack space available for the program when it begins execution. When this limited amount of space is exceeded, an overflow occurs, resulting in the exceptions seen in the Java implementation and the seg-faults seen in the C implementation.

  The data for the execution times gathered shows that the C implementation of the Ackermann function is entirely dependent on the speed of the machine it is run on. It appears that in situations where a stack overflow occurs, the faster machines exceed the available stack space *faster* than the slower machines. This seems to be the result of the more powerful machines actually generating the stack frames responsible for filling the stack space faster than the slower machines.

  The Java implementation run times yield results that show that the OS X machine runs the Ackermann function in a similar fashion to the C implementation. On Unix, however, the Java version throws a stack overflow exception for Ackermann(3,11) and Ackermann(3,12), function values that it was capable of calculating using the C implementation.

  We believe that the JVM on the Unix machine likely has a smaller default stack size than the JVM on the OS X machine. This would explain the Unix machine’s inability to calculate Ackermann(3,11) and Ackermann(3,12) and why the Unix machine is faster at reaching overflow for Ackermann(4,2) through Ackermann(4,6) than the OS X machine.

  All three systems tested appear to have a cap or upper bound on the time it takes for them to reach stack overflow. As can be seen from the graphs, each system’s time to reach stack overflow levels off beginning with Ackermann(4,2) and the time it takes to reach stack overflow varies only slightly for each system for larger Ackermann function calls. This behavior is the result of the predefined stack size determined when the program begins executing. Since each machine as its own predefined stack size, it takes each machine approximately the same amount of time to exceed its respective stack space regardless of the increase in calls required by a larger Ackermann call.

The AXP system behaved somewhat differently than the other two systems tested. While the OS X and Unix machines both encountered seg-faults for large Ackermann function calls, the AXP system threw an error stating that the CPU time limit had been exceeded. It is possible that this error is the result of a limit on our account rather than the system itself. If this is the case, the call stack space for the AXP might not have been exceeded, and instead a process manager terminated the program.

### Future Work

Out of curiosity, we decided to see if the runtimes of the programs would increase if more stack space were allocated for the program. We tested this for the Java implementation of the program on OS X, using the compiler flag –Xss1024m to increase the JVM’s stack size for the program to 1024 megabytes.

Ackermann(4,2) ran for 624 minutes, 44.893 seconds before a stack overflow occurred. This result seems inline with the idea that the amount of stack space is the limiting factor on how long the Ackermann function can run on a system. Increasing the stack space enough would likely allow larger values of the Ackermann function to be computed.

### Problems Encountered

Problems encountered regarding implementation choices were covered in the C Implementation section, and dealt almost entirely with finding the appropriate language to use; as well as passing command line arguments through on the AXP.

### Time Chart

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Thursday | Friday | Saturday | Sunday | Monday | Tuesday | Wednesday |
| James | 2 hrs | 1.5 hrs | 0 hrs | 3 hrs | 2.5 hrs | 3.5 hrs | 9.5 hrs |
| Sam | 2 hrs | 1.5 hrs | 0 hrs | 3 hrs | 2.5 hrs | 3.5 hrs | 9 hrs |

# Appendix A – C Implementation Listing

#include <stdio.h>

#include <time.h>

#define INT long long int

INT Ackermann(INT m, INT n)

{

if (m == 0) {

return n + 1;

} else if (n == 0) {

return Ackermann(m - 1, 1);

} else {

return Ackermann(m - 1, Ackermann(m, n - 1));

}

return Ackermann(m - 1, Ackermann(m, n - 1));

}

int main(int argc, char\* argv[])

{

int m = 0;

int n = 0;

printf("ENTER M AND N SEPARATED BY A SPACE: ");

scanf("%d", &m);

scanf("%d", &n);

clock\_t start = clock(), diff;

INT r = Ackermann(m, n);

printf("For Ackermann(%d,%d) the result is: %lli\n", m, n, r);

diff = clock() - start;

INT msec = diff \* 1000 / CLOCKS\_PER\_SEC;

printf("Time taken %lli seconds %lli milliseconds\n\n", msec/1000, msec%1000);

}//end main

# Appendix B – Java Implementation Listing

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac Ackermann.java

\* Execution: java Ackermann M N

\*

\* Calculate the Ackermann function A(M, N) using a straightforward

\* recursive program.

\*

\* % java Ackermann 3 8

\* 2045

\*

\* % java Ackermann 3 9

\* StackOverflowError

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class Ackermann {

public static long ackermann(long m, long n) {

if (m == 0) return n + 1;

if (n == 0) return ackermann(m - 1, 1);

return ackermann(m - 1, ackermann(m, n - 1));

}

public static void main(String[] args) {

long M = Long.parseLong(args[0]);

long N = Long.parseLong(args[1]);

long startTime, stopTime, overTime = 0;

startTime = System.nanoTime();

stopTime = System.nanoTime();

//overhead time for the program is calculated

overTime = stopTime - startTime;

long time;

startTime = System.nanoTime();

System.out.println("ack(" + M + "," + N + ") = " + ackermann(M, N));

stopTime = System.nanoTime();

time = stopTime - startTime - overTime;

System.out.println("Total time for ack(" + M + ", " + N + ") is " + (time/1000000));

}//end main

}

# Appendix C – References

<http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.3>

<http://introcs.cs.princeton.edu/java/75universality/Ackermann.java.html>

<http://demonstrations.wolfram.com/RecursionInTheAckermannFunction/>

Introduction to the Design and Analysis of Algorithms by Anany Levitin

Assignment sheet – Dr. Robert Strader