Solving Double Execution of Java's `paint()` Method by Counting Down to the Heat Death of the Universe (plus language compendium)

Braden Oh, Vedaant Kuchhal, Junseok Kang, Andrew Mascillaro, Justin Kunimune, Shashank Swaminathan, Devlin Ih, Elias Gabriel, Audrey Lee, Colin Snow, Hyunkyung Rho, Ben Morris, Solomon Greenberg, Mahima Beltur, Anusha Datar, Jonathan Kelley, Pranavi Boyalakuntla, Xander Hughes, Devyn Oh, Aidan Schmitigal

Olin College of Engineering: Computing

Contents

1 Background and Motivation

The final project of my high school "Honors C++" course was to program a graphical animation in a Java applet. While writing this program I found that the animation would consistently run twice in a row, meaning that the pre-programmed animation would run all the way through then would immediately restart, preventing the user from being able to either recognize or appreciate the ending of the cinematic masterpiece as they were forced to watch the film a second time.

Investigation of this bug led me to the `paint()` method employed by the Java Virtual Machine's (JVM) graphics engine. The JVM calls the `paint()` method to generate the graphics displayed in a window. This method is called automatically anytime the JVM determines that the graphical user interface (GUI) needs to be refreshed. Ideally this occurs only when the GUI actually does need to be refreshed, but, in practice, the JVM will execute the paint method when no reason for a GUI refresh is apparent. The JVM's double execution of `paint()` command is a symptom described on numerous message boards including Stack Overflow, MacRumors, Code Ranch, and Tek Tips. Unfortunately the JVM's reason for re-executing the `paint()` method is opaque to the programmer, so the actual cause of the double execution could not be easily investigated. Thus, in order to prevent the user from becoming confused, a clever programmatic solution would be required. Because the program was only an animation that did not require any responsiveness to user input, an easy way to make the double execution invisible to the user was to insert a block of code at the end of the animation, right at the end of the `paint()` method, to consume an enormous amount of time, giving the user more time to view the final image displayed in the window. After implementing enormous addition problems which produced only seconds of delay I began to experience a burning desire to defeat the JVM and guarantee that it could never run my animation again.

What better way to assure this than to have the `paint()` method wait until the heat death of the universe? This would guarantee that the immutable laws of thermodynamics would step in to prevent the JVM from executing a second time, even in the event that the computer remained switched on and plugged into a consistent power source for billions of years. The Wikipedia page entitled "Heat death of the universe" reports that the Hawking radiation evaporation of a supermassive black hole of 10^{11} Earth masses is on the order of 10^{100} years. This amount of time seemed sufficient to prevent `paint()` from ever running again, so I wrote a simple Java function to count down that many seconds and called it at the end of my implementation of the `paint()` function.

2 Results

I am happy to report that this solution worked perfectly. The `paint()` function executed for the first time and then began counting down in the background. My apparent fix to the double execution of `paint()` impressed my programming teacher, but also confused him as he saw the script continued to run beyond the animation's apparent conclusion. He terminated the program's operation prior to `paint()`'s second execution, rendering this solution a practical success.

This algorithm of counting down to the heat death of the universe provides a valuable solution to any programmer needing to assure that no further computations can take place following a particular block of code. In order to provide as universal a solution as possible, we have provided translations of this algorithm in the only programming languages that matter.

3 Language Compendium

3.1 Java

The maximum 32-bit integer value that Java can store is 2,147,483,647, which is on the order of 10^9. To make the math easy, I used the integer 1×10^9 as a loop counter, then looped through 10^{100} years, Calling `Thread.sleep(1000)` (1000 ms) to count through each of the 31,536,000 seconds in a year.

```
import java.lang.Thread;
public void wait_for_heat_death_of_the_universe() {
    // Loop through 10^100 years
    for(long count1 = 0; count1 <= 1000000000; count1++) {
    for(long count2 = 0; count2 <= 1000000000; count2++) {
    for(long count3 = 0; count3 <= 1000000000; count3++) {
    for(long count4 = 0; count4 <= 1000000000; count4++) {
    for(long count5 = 0; count5 <= 1000000000; count5++) {
    for(long count6 = 0; count6 <= 1000000000; count6++) {
    for(long count7 = 0; count7 <= 1000000000; count7++) {
    for(long count8 = 0; count8 <= 1000000000; count8++) {
    for(long count9 = 0; count9 <= 1000000000; count9++) {
    for(long count10 = 0; count10 <= 1000000000; count10++) {
    for(long count11 = 0; count11 <= 1000000000; count11++) {
    // Loop through 31,536,000 seconds per year
    for(long count12 = 0; count12 <= 31536000; count12++) {
        try{Thread.sleep(1000);}    // Wait 1000 ms
        catch(InterruptedException ie) {}  // This is a syntactical formality
    }}}}}}}}}}}}
}
```

3.2 Python 3

Astoundingly, Python 3 allows a user to handle the number `1e100`, which is represented as a floating-point number. Even more astoundingly, Python even allows a user to convert it into an integer which can be used as a loop counter (for reference, the `sys.getsizeof` method indicates that this integer requires 72 bytes of memory). This number is actually represented in memory as a number slightly larger than `1e100`, but the difference is mere fractions of a percent (albeit that doesn't mean much at numbers of this order of magnitude).

```
import time
def wait_for_heat_death_of_the_universe():
    for count in range(0, int(1e100)):  # Loop through 10^100 years
        time.sleep(31536000)            # Wait for one year (in seconds)
```

3.3 B

B is a predecessor of C written by B. W. Kernighan at Bell Labs. It does not implement types, for loops, or sleep commands, but does support while loops and recursion as well as system calls. Here we recursively call the UNIX "sleep 31536000" command 10^{100} times to achieve our desired time.

```
main() {
    WaitForHeatDeath(100);
}

WaitForHeatDeath(n) {
    if (n == 0) {
        system("sleep 31536000");
        return;
    }
    auto a, b;
    a = 10;
    b = 0;
    while(a > b) {
        b = b + 1;
        WaitForHeatDeath(n - 1);
    }
}
```

3.4 C

This is a very similar implementation to that of Java, except for one key difference - the `long` type in C can be 32 or 64 bit depending on the operating system, so a 64-bit integer - using the standard integer library - was specified, and for loops were nested accordingly with a one year sleep function from the Unix standard library.

```
#include <unistd.h>
#include <stdint.h>
int main(){
    for(int64_t count1 = 0; count1 <= 10000000000; count1++){
    for(int64_t count2 = 0; count2 <= 10000000000; count2++){
    for(int64_t count3 = 0; count3 <= 10000000000; count3++){
    for(int64_t count4 = 0; count4 <= 10000000000; count4++){
    for(int64_t count5 = 0; count5 <= 10000000000; count5++){
    for(int64_t count6 = 0; count6 <= 10000000000; count6++){
    for(int64_t count7 = 0; count7 <= 10000000000; count7++){
    for(int64_t count8 = 0; count8 <= 10000000000; count8++){
    for(int64_t count9 = 0; count9 <= 10000000000; count9++){
    for(int64_t count10 = 0; count10 <= 10000000000; count10++){
        sleep(31536000);
    }}}}}}}}}}
}
```

3.5 D

D is a general-purpose programming language with static typing, systems-level access, and C-like syntax. Thus, the implementation is effectively the same as C, noting the different modules.

```
import std;
import core.thread;
void main(){
    for(int64_t count1 = 0; count1 <= 10000000000; count1++){
    for(int64_t count2 = 0; count2 <= 10000000000; count2++){
    for(int64_t count3 = 0; count3 <= 10000000000; count3++){
    for(int64_t count4 = 0; count4 <= 10000000000; count4++){
    for(int64_t count5 = 0; count5 <= 10000000000; count5++){
    for(int64_t count6 = 0; count6 <= 10000000000; count6++){
    for(int64_t count7 = 0; count7 <= 10000000000; count7++){
    for(int64_t count8 = 0; count8 <= 10000000000; count8++){
    for(int64_t count9 = 0; count9 <= 10000000000; count9++){
    for(int64_t count10 = 0; count10 <= 10000000000; count10++){
        Thread.sleep( dur!("seconds")( 31536000 ) );
    }}}}}}}}}}
}
```

3.6 JavaScript

With the new update of JavaScript, there is no need to consider the Number data type's MAX_SAFE_INTEGER. The BigInt Datatype allows a way to represent whole numbers larger than 2^{53}-1. With this, we can use a similar structure to Python's for loop and create this function.

```
function end_of_time(){
    for (let i = BigInt(0); i < BigInt(1e100); i++) {
        sleep(31536000);
    }
}
```

3.7 TypeScript

Since the JavaScript implementation is nearly unreadable, TypeScript fortunately allows us to declare the type of each variable in the source code, which is desirable for any large and complicated program like this one.

```
function end_of_time(): void {
    for (let i: number = BigInt(0); i < BigInt(1e100); i++) {
        sleep(31536000);
    }
}
```

3.8 MATLAB

This implementation is as simple as it can get - initially, it was tempting to iterate across an array from 1 to 1e100 (it's difficult to find exact data on MATLAB's maximum array size), but an array of that size would well exceed the maximum size that any PC can handle. So looping through 10^{10} 10 times proved to be an effective solution. MATLAB provides the inbuilt `pause` function to pause each iteration for one year.

```
function wait_for_heat_death_of_the_universe()
    % Loop through 10^100 years
    for year1 = 1:10000000000
        for year2 = 1:10000000000
            for year3 = 1:10000000000
                for year4 = 1:10000000000
                    for year5 = 1:10000000000
                        for year6 = 1:10000000000
                            for year7 = 1:10000000000
                                for year8 = 1:10000000000
                                    for year9 = 1:10000000000
                                        for year10 = 1:10000000000
                                            % Pause for one year
                                            pause(31536000);
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

3.9 Ruby

As with Python, Ruby allows for the definition of arbitrarily sized integers. At scales larger than a machine word, where a buffer overflow may occur, Ruby will automatically convert any integer to a bignum representation. In this representation, integers consume multiple bytes. The number 10^{100} consumes only 42 bytes (compared to the 72 used by Python), making it a better-suited implementation choice for those concerned with memory usage.

Unfortunately, Ruby's `sleep` implementation can only act on integers within machine precision, so though our desired amount of time `(10 ** 100) * 31536000` can be represented, it cannot be used. Fortunately, as with other methods, we can get around this by instead delaying for 1 year 10^{100} times.

```
(10 ** 100).times { sleep 31536000 }
```

3.10 Elisp

Emacs Lisp, or Elisp, is the extension language for the GNU/Emacs text editor. It is probably the most commonly used dialect of the Lisp family.

GNU/Emacs users tend to constantly promote their silly editor from the 1980s, and how it is better than modern IDEs (me). Sometimes you want to tell those people to shut up. You can lock their single-threaded editor until the heat death of the universe by evaluating the following snippet. Hilariously, ELisp, a language meant primarily for text manipulation and writing editor commands, allows arbitrary precision integers with its bignum type.

```
(dotimes (count (expt 10 100))
  (sleep-for 31536000))
```

3.11 Common Lisp

Common Lisp is a powerful, general purpose, multi-paradigm programming language in the Lisp family. CL is known for its stability, speed, powerful macro system based on s-expression manipulation, and it's unparalleled interactive programming experience through the SLIME environment.

CL supports arbitrary precision integers through the bignum type, which will let you loop over 10^{100} times.

The following code should be implementation agnostic, and is currently being tested with SBCL.

```
(dotimes (count (expt 10 100))
  (sleep 31536000))
```

3.12 Go

The Go implementation follows a process similar to both JavaScript and the more legible TypeScript alternative. Sadly, Go is more verbose in defining big integers, as standard operators such as `**`, `<`, and `+` are not overloaded. Rather, function calls to perform said operations are used (`big.Int.Exp`, `big.Int.Cmp`, `big.Int.Add` respectively). As with Ruby, this implementation consumes 42 bytes (333 bits), but is likely more efficient as it is compiled rather than interpreted.

```
import (
    "time"
    "math/big"
)

one := big.NewInt(1)
eons := new(big.Int).Exp(big.NewInt(10), big.NewInt(100), nil)
counter := big.NewInt(0)

for ; counter.Cmp(eons) == -1; counter.Add(counter, one) {
    time.Sleep(31536000 * time.Second)
}
```

3.13 Racket

Racket is a member of the Lisp family of programming languages and is a descendent of Scheme. It is a language designed for defining domain specific languages, most well known for "teaching languages" used to teach functional programming concepts.

Racket, like the other lisps included here, supports arbitrary precision integers with bignums.

```
#lang racket

(define (heat-death year)
  (when (< year (expt 10 100))
    (sleep 31536000)
    (heat-death (add1 year))))

(heat-death 0)
```

3.14 Scratch

Scratch is a visual block-based programming language and game engine. It is commonly used in schools and classrooms to introduce young children to core programming concepts. As such, it may be useful to utilize when teaching concepts such as the inevitability of universal heat death, and its applicability to painting. See Figure 1 for implementation.

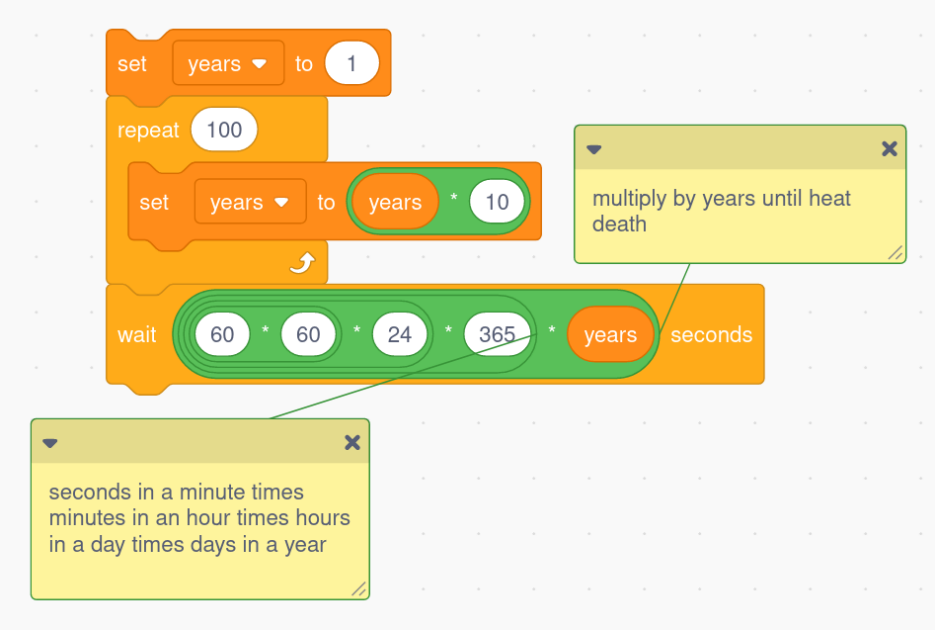

Figure 1: In the code above, comments are provided for visual comprehension.

3.15 OCaml

OCaml is a powerful statically functional programming language, commonly used to teach core computer science topics. It's portability, speed, and expressiveness make it an ideal candidate for development, thus a situation in which one would want to pause execution of a misbehaving program is likely to occur. Tragically, OCaml removed built-in support for arbitrarily-sized integers

in recent versions, offloading that functionality to a separate packaged `zarith`.

Installing that package, we can make use of it in a similar way as above (that is, instructing your program to wait for 31536000 seconds 10^{100} times). On a 64bit architecture, OCaml uses 48 bytes to store our eon counter.

```
#load "zarith.cma";;

let rec wait_heat_death (remaining: Z.t): bool =
    if remaining = Z.zero then true else
        let _ = Unix.sleep 31536000 in wait_heat_death (Z.pred remaining);;

let _ = wait_heat_death (Z.pow (Z.of_int 10) 100);;
```

3.16 Bash

Since Bash coerces string data into integers whenever necessary, the best way to define the value 10^{100} without floating point errors is to create a string of 1 followed by 100 zeroes, as done in the first line below. Then, iterate over each of these years and sleep for 365 days.

```
years="1"$(printf "0%.0s" {1..100})
for i in seq $years; do sleep 365d; done
```

3.17 Fortran

Algorithmically, the Fortran implementation is similar to all other nested loop-based approaches.

```
Program HeatDeath
do a = 0, 1000000000
do b = 0, 1000000000
do c = 0, 1000000000
do d = 0, 1000000000
do e = 0, 1000000000
do f = 0, 1000000000
do g = 0, 1000000000
do h = 0, 1000000000
do i = 0, 1000000000
do j = 0, 1000000000
call sleep(31536000)
end do
end do
end do
end do
end do
end do
end do
end do
end do
end do
end do
End Program HeatDeath
```

3.18 ArnoldC

ArnoldC is an imperative JVM-bytecode assembly language in which instructions and basic keywords are replaced with one-liners from different Arnold Schwarzenegger movies. The language, having its syntax comprised of well-known phrases, is an ideal choice for applications developed by fans and movie-goers. Promisingly, its code is also readable by almost any individual, with or without any formal programming knowledge, as it only requires proficiency in English and Schwarzenegger mannerisms. The language, compiling directly to JVM bytecode, is also able to run on any platform without recompilation.

Tragically, ArnoldC has no built-in support for bignums, timing functionality, nor any way to make system calls. As such, traditional methods like those presented in other languages are provably impossible.

Building on the work done by Lambert and Power[1], however, we can approximate a solution using the execution time of a known bytecode. As shown in their paper, we know with confidence that any used bytecode time is platform-independent (in every regard except processor speed, for which we account during usage), making it an useful drop-in as a base unit of elapsing time in a general-purpose solution. From the 137 timed bytecodes, we pick integer division (`idiv`) due to its comparatively (by an order of magnitude) longer execution time of 3.449739×10^{-8} seconds vs. all other used operations; thus, our implementation can delay for longer increments and be more efficient. Floating point conversions (`d2i` and `f2i`) take longer, but cannot be used as ArnoldC only supports 16-bit signed integers.

Combining this novel unit of time with the recursive approach demonstrated in B and the common looping approach, and knowing that our selected bytecode consumes a magnitude more time than the other programmatic instructions, we can successfully implement an ArnoldC program that will amortizedly delay for our desired time. To achieve heat death, we must recurse the number of times it takes to reach 10^{100} years, which assuming a time unit of 3.449739×10^{-8} seconds, equates to $10^{100} * \frac{31536000}{3.449739 \times 10^{-8}} \approx 10^{116}$ iterations. The implementation provided achieves this through deep recursion, where each recurse recurses 10 times.

Lambert and Power calculated the used bytecode instruction time using a ≈1GHz dual core Intel Pentium III. To account for processors operating outside of the 1-10GHz range, the programmer can simply decrement or increment the magnitude of iterations for each magnitude difference in processor speeds ($116 \pm 1n$).

[1] Lambert, J. M., J. F. Power, *Platform Independent Timing of Java Virtual Machine Bytecode Instructions*, Electronic Notes in Theoretical Computer Science. **220** (2008), pp. 97–113.

```
LISTEN TO ME VERY CAREFULLY HEATDEATH
I NEED YOUR CLOTHES YOUR BOOTS AND YOUR MOTORCYCLE n
    HEY CHRISTMAS TREE flag
    YOU SET US UP @I LIED
    GET TO THE CHOPPER flag
        HERE IS MY INVITATION n
        YOU ARE NOT YOU YOU ARE ME @I LIED
    ENOUGH TALK
    BECAUSE I'M GOING TO SAY PLEASE flag
        DO IT NOW WAITOPCODE
    BULLSHIT
        GET TO THE CHOPPER n
            HERE IS MY INVITATION n
            GET DOWN 1
        ENOUGH TALK
        DO IT NOW WAITMULTI n
    YOU HAVE NO RESPECT FOR LOGIC
HASTA LA VISTA, BABY
LISTEN TO ME VERY CAREFULLY WAITMULTI
I NEED YOUR CLOTHES YOUR BOOTS AND YOUR MOTORCYCLE n
    HEY CHRISTMAS TREE isLessThan10
    YOU SET US UP @NO PROBLEMO
    HEY CHRISTMAS TREE i
    YOU SET US UP @I LIED

    STICK AROUND isLessThan10
        GET TO THE CHOPPER i
            HERE IS MY INVITATION i
            GET UP 1
        ENOUGH TALK
        DO IT NOW HEATDEATH n
        GET TO THE CHOPPER isLessThan10
            HERE IS MY INVITATION 10
            LET OFF SOME STEAM BENNET i
        ENOUGH TALK
    CHILL
HASTA LA VISTA, BABY
LISTEN TO ME VERY CAREFULLY WAITOPCODE
    HEY CHRISTMAS TREE pi
    YOU SET US UP @I LIED
    GET TO THE CHOPPER pi
        HERE IS MY INVITATION 355
        HE HAD TO SPLIT 113
    ENOUGH TALK
HASTA LA VISTA, BABY
IT'S SHOWTIME
    DO IT NOW HEATDEATH 116
YOU HAVE BEEN TERMINATED
```

3.19 LOLCODE

Unfortunately, LOLCODE 1.2 (the most recent specification, released in 2007) does not yet support a time-based pause function such as Java's `Thread.sleep` or Python's `time.sleep`, nor does it support allowing a user to access the clock of the host computer. As a result, the original algorithm cannot be implemented directly in LOLCODE. LOLCODE is extremely similar to Arnold C in capability, but due to the short time available to this author, a similar outcome was achieved with an infinite loop.

```
HAI 1.2
CAN HAS STDIO?
HOW IZ I WAIT_4_HEAT_DTH_OF_UNVRSE
    VISIBLE "COUNTING DOWN"
    IM IN YR LOOP UPPIN YR VAR WILE WIN
        VISIBLE VAR
    IM OUTTA YR LOOP
IF U SAY SO
I IZ WAIT_4_HEAT_DTH_OF_UNVRSE MKAY
KTHXBYE
```

3.20 PHP

PHP, or Personal Home Page, or PHP Hypertext Protocol, is a programming language that was originally used for a personal project and that became mainstream. This adequately explains the language's *unique* inconsistencies in structure, syntax, and convention, as well as some other language "features". First, it creates a BigInteger of the size 10^{100} by creating a string with the number and inputting this to the BigInteger constructor. Note that the PHP operator for concatenating strings is "." and not "+" like most languages. Then, it iterates over each year, sleeping for a year.

```
function waitForHeatDeath() {
    include('Math/BigInteger.php');

    $years_str = "1";
    for($i = 0; $i < 100; $i++) {
        $years_str .= "0";
    }

    $years = new Math_BigInteger($years_str);
    $big_1 = new Math_BigInteger(1);

    for($i = new Math_BigInteger(0); $i < $years; $i += $big_1) {
        sleep(31536000);
    }
}
```

3.21 Verilog

Verilog is a hardware description language. It can be used in designing and verifying circuits at the register-level of abstraction. By using a 32 bit register containing a value of 10^9, we can loop through this value exponentially 11 times, which will get us an integer value of 10^{99}. And then, we can loop through a separate additional 10 times. This will get us a total integer value of 10^{100}. By using the using `timescale 1s / 1s` command, the time passes once every second in the program. To count once per year, 31536000 seconds need to pass using the `#31536000` command.

```
‘timescale 1s / 1s
module HeatDeathUniverseCounter;
    reg [31:0] W = 32’h3B9ACA00; //10^9
    integer i, j, a, b, c, d, e, f, g, h, k;
    integer val_c = 0;
    always@(W)
    begin
        //10^99
        for(i=0;i<W;i=i+1) begin
            for(j=0;j<W;j=j+1) begin
                for(a=0;a<W;a=a+1) begin
                    for(b=0;b<W;b=b+1) begin
                        for(c=0;c<W;c=c+1) begin
                            for(d=0;d<W;d=d+1) begin
                                for(e=0;e<W;e=e+1) begin
                                    for(f=0;f<W;f=f+1) begin
                                        for(g=0;g<W;g=g+1) begin
                                            for(h=0;h<W;h=h+1) begin
                                                for(k=0;k<W;k=k+1) begin
                                                    #31536000 val_c += 1;
                                                    $display(val_c);
                                                end
                                            end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end

        //10
        for(i=0;i<10;i=i+1) begin
            #31536000 val_c += 1;
            $display(val_c);
        end
    end
endmodule
```

3.22 R

R is a language specialized in statistical computing and graphics. Surprisingly, R has similar structure as that of Python for loops, which results in the code stub below looking almost identical to that of the Python code stub except a few differences: exponentiation is done with a karat symbol, and replacing *time* (Python) with the conveniently built-in *Sys* (R).

```
wait_for_heat_death_of_the_universe <- function() {
    # Loop through 10^100 years
    for (count in 0:10^100) {
        # Wait for one year (in seconds)
        Sys.sleep(31536000)
    }
}
```

3.23 Swift

Swift is Apple's language for app development, and as such is an easy target for stalling forever if one ever wanted to make an app that does nothing for all time. Since Swift has a limited integer size, this must be implemented with several for loops. For abstraction, this is done with a recursive function.

```
import Foundation
func waitForNYears(_ base: Int, toThe exp: Int) {
  if exp == 0 {
    // Sleep one year
    sleep(31536000)
  } else {
    for _ in 1...base {
      waitForNYears(base, toThe: exp-1)
    }
  }
}
// Wait for (10^10)^10 years
waitForNYears(Int(pow(10.0, 10.0)), toThe: 10)
```

3.24 Dyalog APL

Quite possibly one of the best examples of a "write-only" programming language, APL (and the modern flavor required here, Dyalog APL) uses a non-ASCII character set to allow for incredibly powerful, terse, and illegible matrix and vector operations. While APL has support for staggeringly large arrays and numbers (able to directly represent 10^{100}), the delay function does not have support for such large arguments. A "pedantic" way of working around this would be to compound delay calls through an array and the use of the handy $\ddot{\star}$ operator, delaying at each step. However, we hit an array size limit at 2^{64}, much smaller than the 10^{100} needed here. As such, we must commit what is paramount to heresy in such a beautiful language (with a global-state iteration being wholly unacceptable), and rely on traditional recursive methods. A simple function is presented here, to recursively decrement the right argument of a monadic function, delaying one second at each iteration. While the author would prefer to use the elegance of tacit functions, this

is impossible due to the inability to rely on the $\ddot{\star}$ operator for numbers of such size.

```
wait_for_heat_death ← {{ω=0:⋄▽ω-1⊣⍞DL⊢1} 31536000×1E100}
wait_for_heat_death ρ0
```

3.25 COBOL

COBOL, misgivingly, does not support sleeping nor does it support a traditional loop structure. To get around this issue, we call the standard UNIX sleep command with an argument of `"infinity"`. By default, sleeping infinitely on UNIX will result in a delay of 9223372036854775807 seconds (about 10^{11} years). To achieve heat death, we must do this repeatedly (roughly 10^{89} times), or by running 8 recursive loops of 10^{10} followed by one loop of 10^9.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HEAT-DEATH.
DATA DIVISION.
WORKING-STORAGE SECTION.
    01 my-var PIC X(6) VALUE "Hello!".
    01 cmd-line   pic x(15)  value "sleep infinity" & x"00".
PROCEDURE DIVISION.
MAIN-PROCEDURE.
    PERFORM Child1 10000000000 TIMES
    STOP RUN.
Child1.
    PERFORM Child2 10000000000 TIMES.
Child2.
    PERFORM Child3 10000000000 TIMES.
Child3.
    PERFORM Child4 10000000000 TIMES.
Child4.
    PERFORM Child5 10000000000 TIMES.
Child5.
    PERFORM Child6 10000000000 TIMES.
Child6.
    PERFORM Child7 10000000000 TIMES.
Child7.
    PERFORM Child8 10000000000 TIMES.
Child8.
    PERFORM Child9 1000000000 TIMES.
Child19.
    call "SYSTEM" using cmd-line
    DISPLAY my-var.
END PROGRAM HEAT-DEATH.
```

3.26 Mathematica

Mathematica has a max array size of $2^{31} - 1$, so our wait can not be performed in one operation. Like other approaches, we can split this operation into smaller chunks.

```
Do[
    Do[
        Do[
            Do[
                Do[
                    Do[
                        Do[
                            Do[
                                Do[
                                    Pause[31536000],
                                {n, 10000000000}],
                            {n, 10000000000}],
                        {n, 10000000000}],
                    {n, 10000000000}],
                {n, 10000000000}],
            {n, 10000000000}],
        {n, 10000000000}],
    {n, 10000000000}],
{n, 10000000000}]
```

3.27 Perl

Perl is a general-purpose programming language currently used for a including system administration, web development, network programming, GUI development, and more. As with other languages, PERL comes with a limit on data sizes depending on the version of Perl. For the 32-bit version, the maximum array size is 9007199254740992. So, for the ease of computation, I used a similar approach to previously discussed languages in using 10 nested for loops that each count to 1e10.

```
sub(wait_for_heat_death_of_universe){
    for ( 0;  10000000000; 1 ) {
        for ( 0;  10000000000; 1 ) {
            for ( 0;  10000000000; 1 ) {
                for ( 0;  10000000000; 1 ) {
                    for ( 0;  10000000000; 1 ) {
                        for ( 0;  10000000000; 1 ) {
                            for ( 0;  10000000000; 1 ) {
                                for ( 0;  10000000000; 1 ) {
                                    for ( 0;  10000000000; 1 ) {
                                        for ( 0;  10000000000; 1 ) {
                                            sleep(rand(31536000))
    }}}}}}}}}}
}
```

3.28 Lua

Lua is a scripting language built on top of C. Lua does not support a sleep function by default, so we start by defining one. Note that because these *os* calls are expensive, a more optimized solution would use and call a C function to block instead of busy waiting as shown here. However, the spirit of this exercise requires sticking to Lua, so we write the following function:

```
function wait(time)
    local duration = os.time() + time
    while os.time() < duration do end
end
```

The maximum value Lua supports is a 64-bit integer, meaning that we need to use a set of nested loops to wait 10^{100} years. One may choose to indent for stylistic reasons, but we show the loops collapsed here for ergonomics when reading on a page.

```
for count1 = 0, 10000000000 do
for count2 = 0, 10000000000 do
for count3 = 0, 10000000000 do
for count4 = 0, 10000000000 do
for count5 = 0, 10000000000 do
for count6 = 0, 10000000000 do
for count7 = 0, 10000000000 do
for count8 = 0, 10000000000 do
for count9 = 0, 10000000000 do
for count10 = 0, 10000000000 do
    wait(31536000)
end
end
end
end
end
end
end
end
end
end
```

3.29 Julia

In this case, the structure is similar to the Python 3 approach - create a range object to delimit the limits of heat death, nicely assign Julia to sleep for a year (in seconds) each iteration, and let the program go to task. For no other reason than aesthetics, the implementation below uses one-line coding.

```
for count in range(1, 1e100); sleep(31536000); end
```

3.30 Rust - Looping

Rust is a statically typed language pioneered by Grayden Hoare in 2007 and then incubated by Mozilla for use in the Mozilla Firefox web browser. Today, Rust empowers developers to write systems programs that emphasize memory safety and performance. Rust does not natively support arbitrary precision integers: these datatypes are provided by third-party libraries (crates) in the ‘crates.io‘ ecosystem. While Rust does not provide arbitrarily sized integers in the standard library, it does provide unsigned integers up to 128 bits in size, making a nested loop solution more succinct than other languages that only support up to 32 bit or 64 bit integers.

```
fn main() {
    for _ in 0..10_u128.pow(33) {
        for _ in 0..10_u128.pow(33) {
            for _ in 0..10_u128.pow(33) {
                std::thread::sleep(std::time::Duration::from_secs(60 * 60 * 24 * 365 * 10));
            }
        }
    }
}
```

3.31 Rust - Arbitrary Precision

While looping might be more succinct for this particular implementation, it is not as flexible as an arbitrary precision approach. To imitate an arbitrary precision without installing a third-party library, we can use an array of unsigned 128bit integers to achieve the required 333 digits of precision. This approach will simply increment our packed integer array until the target value is reached, sleeping one year between increments.

```
fn main() {
    let mut count: [u128; 3] = [0, 0, 0];
    let target: [u128; 3] = [u128::MAX, u128::MAX, 2u128.pow(77)];

    while count != target {
        for i in count.iter_mut() {
            *i = i.wrapping_add(1);

            if *i != 0 {
                break;
            }

            std::thread::sleep(std::time::Duration::from_secs(60 * 60 * 24 * 365));
        }
    }
}
```

3.32 MIPS Assembly

MIPS (Microprocessor without Interlocked Pipeline Stages) assembly is a RISC (Reduced Instruction Set Computer) ISA (Instruction Set Architecture). It is commonly used in the undergraduate computer engineering teaching curriculum. This approach is very similar to the Java approach. Since the MIPS registers are 32-bit, a maximum number of 2,147,483,647 can be stored. For ease, we use 11 nested loops counting till 10^9 with a one year waiting period for each iteration.

```
.macro sleepSecond
    # wait for 1 second
    li $a0, 1000
    li $v0, 32
    syscall
.end_macro

.macro Terminate
    # end the program
    li $v0, 10
    syscall
.end_macro

li $t0, 0

# wait for one year
waitYear:
    addi $t0, $t0, 1
    sleepSecond
    beq $t0, 31536000, looper1
    j waitYear

# initialize all of the counting variables
li $t1, 0
li $t2, 0
li $t3, 0
li $t4, 0
li $t5, 0
li $t6, 0
li $t7, 0
li $t8, 0
li $t9, 0
li $s0, 0
li $s1, 0

looper1:
    addi $t1, $t1, 1
     beq $t1, 1000000000, looper2
     li $t0, 0
     j waitYear
```

```
looper2:
     addi $t2, $t2, 1
     beq $t2, 1000000000, looper3
     li $t1, 0
     j looper1

looper3:
     addi $t3, $t3, 1
     beq $t3, 1000000000, looper4
     li $t2, 0
     j looper2

looper4:
     addi $t4, $t4, 1
     beq $t4, 1000000000, looper5
     li $t3, 0
     j looper3

looper5:
     addi $t5, $t5, 1
     beq $t5, 1000000000, looper6
     li $t4, 0
     j looper4

looper6:
     addi $t6, $t6, 1
     beq $t6, 1000000000, looper7
     li $t5, 0
     j looper5

looper7:
     addi $t7, $t7, 1
     beq $t7, 1000000000, looper8
     li $t6, 0
     j looper6

looper8:
     addi $t8, $t8, 1
     beq $t8, 1000000000, looper9
     li $t7, 0
     j looper7

looper9:
     addi $t9, $t9, 1
     beq $t9, 1000000000, looper10
     li $t8, 0
     j looper8
```

```
looper10:
     addi $s0, $s0, 1
     beq $s0, 1000000000, looper11
     li $t9, 0
     j looper9

looper11:
     addi $s1, $s1, 1
     beq $s1, 1000000000, exit
     li $s0, 0
     j looper10

exit:
     Terminate
```

3.33 NetFuck

Unfortunately, as the base *brainfuck* language does not provide the ability to 'sleep' for a certain amount of time, we need to look towards an extension of the language that does, such as *Alarm Clock Radio* or the language we have selected, *NetFuck*. Because of the limited instruction set (including no direct *if* operators, etc.), and because we assume some typical limitations to *brainfuck*-derived machines (32-bytes per memory cell, $\tilde{1}00$ memory cells), we need to rely on some pretty brute-force methodology: iteratively adding value to memory cells until we have 10^9 stored in 11 cells; do the same for our other large number counters; use those set values as counters for our loops and iterate through, sleeping for 10ms on each iteration. It amounts to an astonishing amount of un-optimized computation, meaning that on a slow single-cycle CPU the computation alone would be enough to reach the heat death of the universe. With the addition of a timer, we can ensure heat death will happen faster than the end of computation even on high-powered machines. On the bright side, it is amazingly memory efficient - we only require 15 32-byte registers.

```
Set 10^9 in cell 12:
>>>++++++++++
[
 >++++++++++
 [
  >++++++++++
  [
   >++++++++++
   [
    >++++++++++
    [
     >++++++++++
     [
      >++++++++++
      [
       >++++++++++
       [
        >++++++++++
```

```
        <-
       ]
       <-
      ]
      <-
     ]
     <-
    ]
    <-
   ]
   <-
  ]
  <-
 ]
 <-
]

Pre-load 10^9 into the preceding 11 cells, zero cell 12
>>>>>>>>
[
 <+<+<+<+<+<+<+<+<+<+<+<+
 >>>>>>>>>>>>-
]

Set 31536000 in the cell 15:
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
++++++++++++++++++++++++++++
+
[
 >+++++++++++++++++++++++++
 [
  >+++++++++++++++++++++++++
  ++++++++++++++++++++++++++
  +++++++++
  [
   >++++++++++++++++++++++
   ++++++++++++++++++++++++
```

```
   ++++++++++++
   <-
  ]
  <-
 ]
 <-
]

Move 31536000 back into cell 12, zero cell 15:
>>>
[
 <<<+
 >>>-
]

Store 100 in cell 13, move to start:
<++++++++++
[
 <++++++++++
 >-
]
<<<<<<<<<<<<

Use 10 consecutive timers, special to NetFuck, of 10 ms * [cell 13] = 1000 ms each.
Loop 3153600 times, then 10^9 times, 11 times.
[
 >
 [
  >
  [
   >
   [
    >
    [
     >
     [
      >
      [
       >
       [
        >
        [
         >
         [
          >
          [
           >
           [
            >
            [
```

```
            >~~~~~~~~~~
            <-
           ]
           <-
          ]
          <-
         ]
         <-
        ]
        <-
       ]
       <-
      ]
      <-
     ]
     <-
    ]
    <-
   ]
   <-
  ]
  <-
 ]
 <-
]
```

For the sake of conciseness, here is the same NF code as above, but without white-spacing (line-wrapping is done for ease of reading).

```
>>>++++++++++[>++++++++++[>++++++++++[>++++++++++[>++++++++++[>++++++++++[>+++++
+++++[>++++++++++[>++++++++++<-]<-]<-]<-]<-]<-]<-]<-]>>>>>>>>>[<+<+<+<+<+<+<+<+<+
<+<+>>>>>>>>>>>>>-]++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++[>++++++++++++++++
++++++++[>++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++[>++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++<-]<-]<-]>>>[<<<+>>>-]<+++++
+++++[<++++++++++>-]<<<<<<<<<<<<<[>[>[>[>[>[>[>[>[>[>[>[>~~~~~~~~~~<-]<-]<-]<-]<
-]<-]<-]<-]<-]<-]<-]<-]
```

3.34 brainfuck

We just teased at how to make *brainfuck* work without relying on an infinite loop - under very specific assumptions. If we restrict operation to a single-cycle CPU without branch-prediction and other compiler optimizations, running at low frequencies (1MHz), we can mimic sleeping for 1 second by simply running one million computations each loop and forcing the processor to be delayed for one second per iteration.

3.35 Befunge-98

Befunge is a programming language in which an instruction pointer moves along a two-dimensional grid. Using Befunge-98's execute extensions we can call the Unix command "sleep 31536000" to sleep for a year. The row of "!" characters in the source code act as counters; as execution progresses they are modified to store the current iteration number. The range of printable ASCII characters limits the maximum value for each individual counter, but by chaining them and adding carrying logic we can build a timer that counts to 90^{53} years, a value which exceeds the desired wait duration.

```
>"sleep 31536000"=04g1+:13g‘!v
^                          <p40_$b3*03p>23g03gb3*-4p03gb3*-1+:33g‘#@_b3*+03pv
                                       ^                  _v#‘g31:+1g4-*3bg30<
I}!5                       ^                     p4-*3bg30<
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

3.36 Minecraft Redstone

Minecraft is a block-based creative building game with a Turing complete circuit-building system using Redstone and Redstone components. The designed circuit is made up of a series of chained Modified Etho Clocks with Hoppers (MECHs) (See figures 2 3). Each of these MECHs consists of an unmodified Hopper Clock with an added circuit that allows it to be chained when one's observer is pointed at another's final repeater.

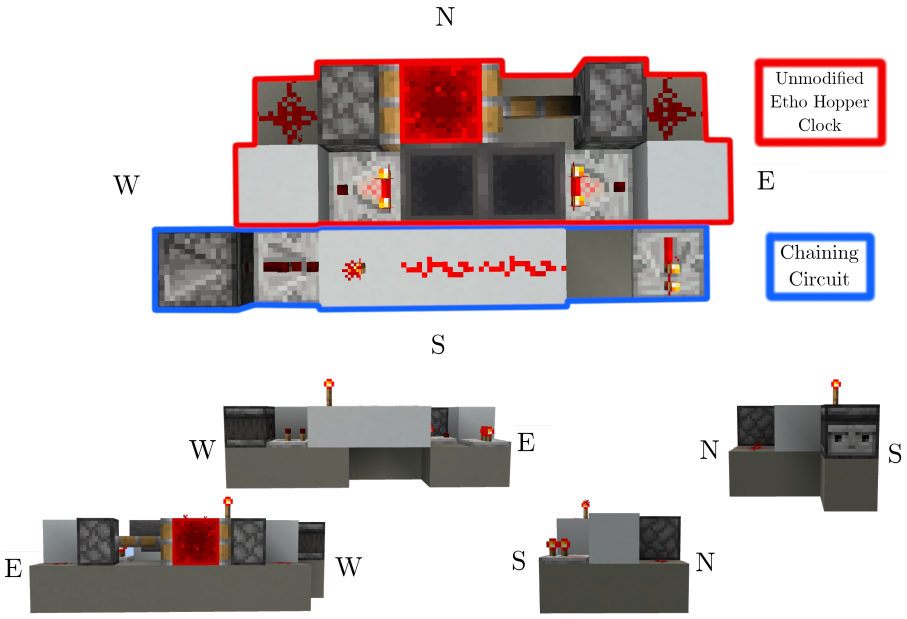

Figure 2: An in-game Modified Etho Clock with Hoppers from several angles, annotations are added for visual comprehension.

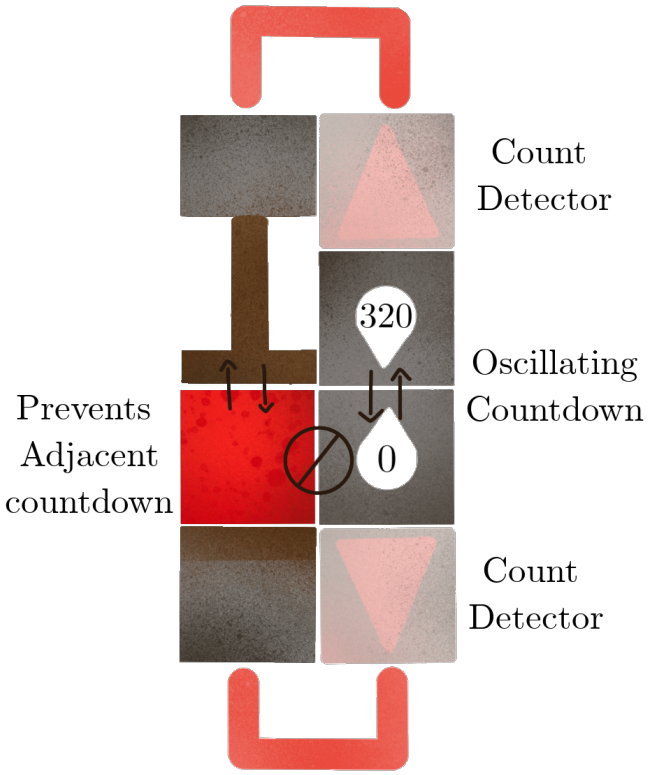

Figure 3: A diagram for an unmodified Etho Hopper Clock with labels added.

The first module in the series is an unmodified Etho Hopper Clock. Hoppers in Minecraft move items once every 8 in game ticks and a hopper can hold up to five 64 item stacks. The output from this clock triggers once every two flip-flops as the items must travel from one hopper to the other and back to the original hopper. The conversion math is shown below.

$$\frac{5 \text{ stacks}}{1 \text{ flipFlop}} \times \frac{64 \text{ items}}{1 \text{ stack}} \times \frac{8 \text{ ticks}}{1 \text{ item}} \times \frac{1 \text{ second}}{20 \text{ ticks}} \times \frac{2 \text{ flipFlops}}{1 \text{ cycle}} = 256 \frac{\text{seconds}}{\text{cycle}}$$

Each MECH is designed to be chainable thus the time between pulses multiplies exponentially as more are tacked on. Conceptually, each can be thought of as a gearbox with a ratio of the number of items multiplied by two as each item travels into and out of the second hopper every cycle. Each MECH is filled full with five 64 item stacks for a total gear ratio of 640:1. This means for every 640 pulses the clock takes in, it will output one pulse. Effectively, each MECH multiplies the total clock's time by the number of items it holds multiplied by two. From this we find that

$$\text{first cycle time} \times (2 \times \text{num items})^{\text{num MECHs}} = \text{total time}$$

We can then plug in known values and solve for the number of MECHs:

$$\text{num MECHs} = \log_{640}\left(\frac{10^{100} \times 3.1536 \times 10^{7}}{256}\right) = 37.4498065 \text{ MECHs}$$

Thus, we need 38 MECHs and one Unmodified Etho Hopper Clock to reach the heat death of the universe.

Minecraft is a game that needs to run on a standard computer so it has limits. Surely that must keep it from being able to count to the end of the universe. The biggest problem is that, redstone only works within 21 chunks or 336 blocks of the player on a standard world. Fortunately, each MECH is only 7 blocks long meaning this circuit would be only 272 (adding 6 for the unmodified hopper clock) blocks long, fitting comfortably within that distance without modification.