A Unicon Benchmark Suite

Shea Newton and Clinton Jeffery

Unicon Technical Report: 16a

June 9, 2014

Abstract

A benchmark suite was used to demonstrate the performance of various Icon language features on systems and compilers of the late 1980’s. For Unicon, the addition of various language features combined with generations of processor improvements motivated the development of a new benchmark suite. This document describes updates to the existing Icon benchmark suite for use in Unicon as well as additional benchmarking programs that serve to better analyze Unicon’s performance on today’s machines.

Unicon Project
http://unicon.org

University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA
1 Introduction

This technical report describes a benchmark suite for the Unicon language, called the Unicon 12 Benchmark Suite. The Unicon 12 benchmarks include revisions of the Icon benchmark suite as well as new benchmarks that measure the performance of additional language features. In addition to enabling the comparison of different platforms and different Unicon implementations, several multi-language benchmarks enable comparison of Unicon’s performance to other languages implementing the same algorithms.

For this project, the Icon benchmark suite was modified in order to increase computation time. These modifications include either larger data sets, an increased number of calculations or more results generated from existing calculations. Additions to the Icon suite include programs developed for the Computer Language Benchmarks Game found at [http://benchmarksgame.alioth.debian.org/](http://benchmarksgame.alioth.debian.org/). These programs allow comparison between similar algorithms implemented in a wide range of languages and on various machine architectures.

2 Modifications to the Existing Icon Benchmark Suite

Included in the Unicon Version 12.1 source distribution are five Icon benchmarks in the directory tests/bench/icon. These tests were used to compare different platforms for running Icon in the late 1980’s. Figure 1 describes these programs and their runtimes as the average of three executions, on a dual core machine running 64-bit Linux Mint 12 with an AMD Opteron 2212 2.0 GHz processor and 8 GB of RAM. These times were measured with the Unicon keyword &time, which measures CPU time with a clock resolution of 1ms on typical Linux Unicon platforms.

The modifications necessary to increase the run-times of these programs were slight, generally increasing n, where n is the number of outcomes generated or simply providing larger data sets for those programs that used input files. As can be seen from Figure 1, on modern processors the original Icon benchmarks that used to take many seconds now have runtimes small enough that even a 1ms resolution may introduce significant errors in the measurements. These modified benchmarks are available from Unicon.org.
<table>
<thead>
<tr>
<th>Benchmark: Features Represented</th>
<th>Description</th>
<th>Run-times</th>
<th>Changes</th>
<th>Extended Run-times</th>
</tr>
</thead>
<tbody>
<tr>
<td>concord: strings, lists</td>
<td>Produces concordance from standard input to standard output where words less than three characters long are ignored.</td>
<td>157ms</td>
<td>The original 447 line <em>Unix Programmer’s Manual</em> input was replaced with a 4719 line .txt version of Dostoevsky’s, <em>Notes from the Underground</em></td>
<td>11.351s</td>
</tr>
<tr>
<td>deal: strings, lists</td>
<td>Deals bridge hands.</td>
<td>140ms</td>
<td>The number of hands dealt was increased from 1 to 50,000.</td>
<td>7.421s</td>
</tr>
<tr>
<td>ipxref: strings, lists, records and tables</td>
<td>Produces a cross reference for Icon programs.</td>
<td>43ms</td>
<td>ipxref originally cross-referenced itself, 239 lines, but inputs 50,687 lines of source code[3].</td>
<td>6.309s</td>
</tr>
<tr>
<td>queens: recursion, backtracking</td>
<td>Generates solutions to the n-queens problem. Program does generation, backtracking and text synthesis.</td>
<td>197ms</td>
<td>Now implements a larger n, where n is the number of queens for which to generate a solution. N was changed from 6 to 12.</td>
<td>11.593s</td>
</tr>
<tr>
<td>rsg: strings, lists</td>
<td>Generates randomly selected sentences from a grammar.</td>
<td>190ms</td>
<td>Originally generated 1000 sentences, this number was changed in order to generate 50,000 sentences.</td>
<td>7.276s</td>
</tr>
</tbody>
</table>

Figure 1: Existing Icon Benchmark Suite, modifications and performance on a modern machine.

### 3 Additions to the Existing Suite

A major shortcoming of the Icon benchmark suite is that it did not cover many of the important language features that defined Icon. The Icon benchmarks deliberately avoided I/O and traditional numeric processing; their failure to emphasize other language features was unintentional. In any case, in order to overcome this issue, the Unicon benchmark suite needs to expand in order to account for Unicon and Icon’s feature set. Ideally, the Unicon benchmark suite will eventually include coverage of the following:

- integer, real, and arbitrary precision arithmetic
- programmer-defined generators, suspension, and backtracking
- tables / sets
- threads
- object construction, field access, method invocation
- 2D and 3D graphics
- TCP and UDP network communication
- DBM and ODBC databases

Due to time and resource limitations, the Unicon 12 benchmark suite omits I/O benchmarking, but addresses the first four of these areas using algorithms developed for the Computer Language Benchmark Game found at [http://benchmarksgame.alioth.](http://benchmarksgame.alioth.)
The advantage of these algorithms is that their performance on various machines and their implementations in various languages is well documented and timed.

Figure 2 describes the programs that make up the Computer Language Benchmarks Game [1]. The source code for Unicon versions of these programs are included in the appendix of this report, and available from Unicon.org.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Features Represented</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-body</td>
<td>performs an N-body simulation of the Jovian planets</td>
<td>real numbers, records</td>
<td>50,000,000</td>
</tr>
<tr>
<td>fannkuch-redux</td>
<td>repeatedly accesses a tiny integer-sequence</td>
<td>integers, lists</td>
<td>12</td>
</tr>
<tr>
<td>meteor-contest</td>
<td>searches for solutions to a shape packing puzzle</td>
<td>integers, records</td>
<td>2098</td>
</tr>
<tr>
<td>fasta</td>
<td>generates and writes random DNA sequences</td>
<td>records, lists, strings, suspend, generators</td>
<td>25,000,000</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>calculates an eigenvalue using the power method</td>
<td>real numbers, lists</td>
<td>5,500</td>
</tr>
<tr>
<td>reverse-complement</td>
<td>reads DNA sequences and writes their reverse-complement</td>
<td>strings, lists</td>
<td>25,000,000</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>generates a Mandelbrot set and writes a portable bitmap</td>
<td>integers, threads</td>
<td>16,000</td>
</tr>
<tr>
<td>k-nucleotide</td>
<td>repeatedly updates hash-tables and k-nucleotide strings</td>
<td>strings, lists</td>
<td>25,000,000</td>
</tr>
<tr>
<td>regex-dna</td>
<td>matches DNA 8-mers and substitutes nucleotides for IUB code</td>
<td>records, strings, threads</td>
<td>5,000,000</td>
</tr>
<tr>
<td>pidigits</td>
<td>calculates the digits of Pi with streaming arbitrary-precision arithmetic</td>
<td>arbitrary precision integers</td>
<td>10,000</td>
</tr>
<tr>
<td>chameneos-redux</td>
<td>repeatedly performs symmetrical thread rendezvous requests</td>
<td>threads, records</td>
<td>6,000,000</td>
</tr>
<tr>
<td>threading</td>
<td>repeatedly switches from thread to thread passing one token</td>
<td>threads</td>
<td>50,000,000</td>
</tr>
<tr>
<td>binary-trees</td>
<td>allocates and deallocates many binary trees</td>
<td>integers, lists, threads</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 2: Descriptions of the programs that comprise the Computer Language Benchmarks Game.

4 Benchmark Game Results

Preliminary results for the implemented benchmarks follow, along with the performance of similar algorithms implemented in Python and C. Timings for the Unicon programs were calculated as real/wall clock time using Unicon’s gettimeofday() and CPU time using the Unicon keyword &time. Unicon’s &time keyword has a 1ms resolution, while the gettimeofday() calculates timing with a resolution of 1µs. For these tables, gettimeofday() results have been rounded to the nearest millisecond. Python and C timings were calculated using Linux’s time(1) utility where the CPU time reported below is the sum of the reported user and system time.
The times presented here represent the average runtime of over three executions on a 32 core machine running 64-bit Fedora 16 with an AMD Opteron 6272 2.1 GHz processor and 32 GB of RAM. Unicon was compiled with -O2 optimization and concurrency enabled. The C and Python results are presumably for carefully tuned, near optimal solutions, while the Unicon times are based on somewhat untuned translations of the C and Python solutions. For instance, due to the efficiency of string scanning in Icon and Unicon, there has been very little work done to optimize regular expression functionality. At this point, the benchmarks show Unicon to be 1.6x-334x slower than C, and 1x faster to 30x slower than Python. Unicon’s optimizing compiler, invoked via unicon -C and commonly referred to as Uniconc, performs remarkably well. When it is enabled in Unicon source distributions (via make Uniconc after the VM build is performed via make Unicon), Uniconc demonstrates execution times 1.6x faster to 60x slower than C and 11x faster to 6.2x slower than Python.

Additional benchmarks, tuning, and updated numbers will increase the usefulness of the information, and identify opportunities for language implementation improvements. For example, one of the Python benchmarks depends on installation of a C library (GMP) that appears to outperform Unicon’s arbitrary precision integer implementation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C</th>
<th>Python</th>
<th>Unicon</th>
<th>Uniconc</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-body</td>
<td>10.318s</td>
<td>136x</td>
<td>345.2x</td>
<td>57.9x</td>
</tr>
<tr>
<td>fannkuch-redux</td>
<td>65.245s</td>
<td>61.5x</td>
<td>199.7x</td>
<td>20.3x</td>
</tr>
<tr>
<td>meteor-contest</td>
<td>0.188s</td>
<td>100.6x</td>
<td>98.9x</td>
<td>9.1x</td>
</tr>
<tr>
<td>fasta</td>
<td>8.195s</td>
<td>45.1x</td>
<td>70.4x</td>
<td>10.2x</td>
</tr>
<tr>
<td>spectral-norm</td>
<td>37.929s</td>
<td>37.6x</td>
<td>58.4x</td>
<td>7.2x</td>
</tr>
<tr>
<td>reverse-complement</td>
<td>1.583s</td>
<td>6.5x</td>
<td>10.4x</td>
<td>4.1x</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>49.677s</td>
<td>1.3x</td>
<td>8.1x</td>
<td>8.1x</td>
</tr>
<tr>
<td>k-nucleotide</td>
<td>9.413s</td>
<td>64.8x</td>
<td>180.9x</td>
<td>55.4x</td>
</tr>
<tr>
<td>regex-dna</td>
<td>21.228s</td>
<td>1.3x</td>
<td>1.7x</td>
<td>0.6x</td>
</tr>
<tr>
<td>pidigits</td>
<td>2.491s</td>
<td>1x</td>
<td>8.6x</td>
<td>3.1x</td>
</tr>
<tr>
<td>chameneos-redux</td>
<td>2.083s</td>
<td>211.2x</td>
<td>334x</td>
<td>N/A</td>
</tr>
<tr>
<td>(thread specific)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread-ring</td>
<td>7m:32.017s</td>
<td>1.4x</td>
<td>1.6x</td>
<td>N/A</td>
</tr>
<tr>
<td>(thread specific)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary-trees</td>
<td>22.408s</td>
<td>2.1x</td>
<td>62.2x</td>
<td>11.6x</td>
</tr>
</tbody>
</table>

Figure 3: Execution times for the Computer Language Shootout Game benchmarks expressed as ratios of C program execution times.

5 Future Work

In order to answer the question of how Unicon performs on various operating systems and architectures, the Unicon project is soliciting the community of Unicon users to record and report results on diverse platforms. A similar approach was taken for Icon in the 1980’s with a table of user-reported benchmark results published in the Icon Newsletter #31[2]. Obtaining end user participation depends on an end product that is both easy for users to run, and easy for them to report their results. The Unicon 12 Benchmark Suite aims to meet
this end and includes multiple scripts and logging tools. Scripts included in the package are designed to allow users to log results for multiple run-times estimated at 15 minutes, 1 hour, and 20 hours to run to completion. Details about these tools can be found in Appendix A or the README file in test/bench.

There remains the work of developing further benchmarks for the suite in order for it to address Unicon’s feature set comprehensively. Those benchmarks should include programs covering object construction, field access, method invocation, 2D and 3D graphics, TCP and UDP network communication, and DBM and ODBC databases.

6 Conclusions

The Unicon 12 benchmark suite serves a variety of purposes. It was motivated by a need to compare Unicon’s speed on different operating systems and CPU types. In addition it allows comparison of different Unicon implementations.

Future revisions to the Unicon 12 suite might not include all the original Icon benchmarks, since they are somewhat redundant. Similarly, not all the language benchmark game benchmarks might be necessary, since some of them are redundant with each other or with Icon benchmarks.

In terms of language coverage, there remains the work of developing further benchmarks for the suite in order for it to address Unicon’s feature set comprehensively. Those benchmarks should include programs covering object construction, field access, method invocation, 2D and 3D graphics, TCP and UDP network communication, and DBM and ODBC databases.

References


Appendix A: README

==========
Unicon 12 Benchmark Suite

Requirements for Windows: At least sh.exe and make.exe on the path. You may need the MinGW package.
Linux systems should be fine.

Some of these benchmarks require that Unicon has concurrency enabled.

To check if Unicon has concurrency enabled, type "unicon -features" on the command line and check for concurrent threads listed as a feature.

If you’ve built Unicon from sources, concurrency may be enabled by un-commenting the line "#define Concurrent 1" in the file define.h located in /src/h/ in your Unicon directory.

In this zip, four automated Unicon versions of the Computer Language Benchmark Game’s programs may be run using the Unicon scripts "run-test," "run-short," "run-med" and "run-shootout." The intricacies of these scripts are detailed below.

Instructions for building and running the benchmarks included in this zip file:

1. After unzipping the folder, run the make command to build independent executables, .u files for the automated scripts, and the .dat files required for k-nucleotide, regex-dna and reverse-complement.

2. Once built, the script executables run-test and run-shootout may be run in two ways:

[A] One option is to run a script with no arguments (e.g., ./run-test or ./run-benchmark), in order to run each benchmark in succession with output suppressed.

[B] The second option is to run the script with the name of the benchmark you’d like to run (listed below) as the script’s argument (e.g., ./run-test binary-trees or ./run-shootout meteor-contest, in order to run that specific benchmark with output suppressed.

3. Aside from the scripts, ideally the make command will build independent executables that may be run on their own in order to see output or to set your own input values. These executables do not have timing mechanisms built into them.

4. The included makefile has rules for each benchmark and script so you should be able to make each independently if desired.

Generating Input Files
A file fasta.c was included in order to quickly generate the input .dat files. If there is no C compiler present you may opt to generate all input files with a Unicon program when prompted after running make or you may generate them incrementally. Using Unicon to generate input files can take some time so if you’d like to generate input incrementally, you may specify which version or versions of the benchmarks to generate input files for using the generate program included in this package.

Examples would be:
./generate test
or
./generate benchmark
or
./generate test benchmark med
to generate a few at a time

Running the benchmarks

The data at below represents input values required for a short Unicon benchmark run and a full "benchmark shootout game website approved run." Also included are the input values for testing and for an alternate version of the benchmarks adjusted in order to achieve, hopefully, meaningful results without requiring an entire 20+ hour run.

Approximate time takes to run each version of this benchmark suite:

run-test: 20 seconds
run-benchmark: 5 minutes
run-shootout: 20 hours
run-med: 1 hour

Testing Purposes: run-test

concord concord.test
deal 500
ipxref ipxref.test
queens 9
rsg rsg.test
binary-trees: 10
chameneos-redux: 600
fannkuch: 7
fasta: 1000
k-nucleotide: fasta output 25,000
mandelbrot: 200
meteor-contest: 1
n-body: 1000
pidigits: 500
regex-dna: fasta output 10,000
reverse-complement: fasta output 150000
spectral-norm: 100
thread-ring: 1000

Standard Unicon 12 Benchmark Run: run-benchmark
-------------------------------------------------------
concord concord.dat
deal 50000
ipxref ipxref.dat
queens 12
rsg rsg.dat
binary-trees: 14
chameneos-redux: 65,000
fannkuch: 9
fasta: 250000
k-nucleotide: fasta output 150,000
mandelbrot: 750
meteor-contest: 600
n-body: 10,0000
pidigits: 7,000
regex-dna: fasta output 700,000
reverse-complement: fasta output 15,000,000
spectral-norm: 300
thread-ring: 700,000

Full Benchmark Shootout Game Run: run-shootout
-------------------------------------------------------
binary-trees: 20
chameneos-redux: 6,000,000
fannkuch: 12
fasta: 25,000,000
k-nucleotide: fasta output 25,000,000
mandelbrot: 16,000
meteor-contest: 2098
n-body: 50,000,000
pidigits: 10,000
regex-dna: fasta output 5,000,000
reverse-complement: fasta output 25,000,000
spectral-norm: 5,500
thread-ring: 50,000,000

Alternate "n" Benchmark Shootout Game Run: run-med

-------------------------------------------------------
binary-trees: 17
chameneos-redux: 1,500,000
fannkuch: 10
fasta: 7,000,000
k-nucleotide: fasta output 3,000,000
mandelbrot: 3,500
meteor-contest: 2,098
n-body: 1,500,000
pidigits: 10,000
regex-dna: fasta output 5,000,000
reverse-complement: fasta output 25,000,000
spectral-norm: 1,300
thread-ring: 15,000,000

Appendix B: Source

The following is source code for the Unicon programs translated from algorithms written for the Computer Language Benchmark Game.[1]
% n-body.icn
%
% The Computer Language Benchmarks Game
% http://benchmarksgame.alioth.debian.org/
% Translated from Kevin Carson, Tupteq, Fredrick Johansson,
% Daniel Nanz and Maciej Fijalkowski’s Python program

link printf

global PI, SOLAR_MASS, DAYS_PER_YEAR, BODIES, SYSTEM, PAIRS

procedure combinations(L)
  result := []
  every put(result, [L[x := 1 to *L], L[x+1 to *L]])
  return result
end

record xyz(x, y, z)

procedure advance(dt, n)
  every i := 1 to n do {
    every p := !PAIRS do {
      p1 := p[1]
      p11 := p1[1]
      v1 := p1[2]
      p2 := p[2]
      p21 := p2[1]
      v2 := p2[2]
      dx := p11.x - p21.x
      dy := p11.y - p21.y
      dz := p11.z - p21.z
      mag := dt * ((dx^2 + dy^2 + dz^2)^-1.5)
      b1m := p1[3] * mag
      b2m := p2[3] * mag
      v1.x -= dx * b2m
      v1.y -= dy * b2m
      v1.z -= dz * b2m
      v2.x += dx * b1m
      v2.y += dy * b1m
      v2.z += dz * b1m
    }
    every s := !SYSTEM do {
      r := s[1]
      v := s[2]
      r[1] += dt * v.x
      r[2] += dt * v.y
      r[3] += dt * v.z
    }
  }
end

procedure report_energy()
  local e := 0.0, p1
  every p := !PAIRS do {
    p1 := p[1]
    p11 := p1[1]
    p2 := p[2]
    p21 := p2[1]
    dx := p11.x - p21.x
    dy := p11.y - p21.y
    dz := p11.z - p21.z
    e -= p1[3] * p2[3] / ((dx^2 + dy^2 + dz^2)^0.5)
  }
  every b := !SYSTEM do {
    v := b[2]
  }

}
e += b[3] * (v.x * v.x + v.y * v.y + v.z * v.z) / 2.
}

fprintf(output, "%.9r\n", e)
end

procedure offset_momentum(ref)
local px := 0, py := 0, pz := 0.0
every s := !SYSTEM do {
v := s[2]
m := s[3]
px -:= v.x * m
py -:= v.y * m
pz -:= v.z * m
}
v := ref[2]
m := ref[3]
v.x := px / m
v.y := py / m
v.z := pz / m
end

procedure run_nbody(argv)
local ref := "sun"
PI := 3.14159265358979323
SOLAR_MASS := 4 * PI * PI
DAYS_PER_YEAR := 365.24

BODIES := table(
  "sun",
  [xyz(0.0, 0.0, 0.0), xyz(0.0, 0.0, 0.0), SOLAR_MASS],
  "jupiter",
  [xyz(1.84143144246472090e+00, -1.16032004402742839e+00, -1.0362204471123109e-01),
   xyz(1.66007664274036940e-03 * DAYS_PER_YEAR, 7.699011849740425e-03 * DAYS_PER_YEAR, -6.90460016972032e-05 * DAYS_PER_YEAR),
   9.54791938424326609e-04 * SOLAR_MASS],
  "saturn",
  [xyz(4.84143695621393310e+00, 4.1247985641230479e+00, -4.0352341711432138e-01),
   xyz(-2.76742510726862411e-03 * DAYS_PER_YEAR, 4.9952801234917238e-03 * DAYS_PER_YEAR, 2.3041729757376392e-05 * DAYS_PER_YEAR),
   2.85885980666130812e-04 * SOLAR_MASS],
  "uranus",
  [xyz(-1.28943695621393310e+01, -1.5111514016986312e+00, -2.3307578892655734e-01),
   xyz(2.66460137564761618e-03 * DAYS_PER_YEAR, 2.37847173959480950e-03 * DAYS_PER_YEAR, -2.96589568540237556e-05 * DAYS_PER_YEAR),
   4.36624404335156298e-05 * SOLAR_MASS],
  "neptune",
  [xyz(1.53796971148509165e+01, -2.5919316099879641e+01, 1.7925877295037118e-01),
   xyz(2.606777249038932e-03 * DAYS_PER_YEAR, 1.62824170038242295e-03 * DAYS_PER_YEAR, -9.51592254519715870e-05 * DAYS_PER_YEAR),
   5.15138902046611451e-05 * SOLAR_MASS]
)

SYSTEM := []
every put(SYSTEM, !BODIES)
PAIRS := combinations(SYSTEM)
offset_momentum(BODIES[ref])
report_energy()
advance(0.01, argv[1])
report_energy()
end

ifdef MAIN
procedure main(argv)
  output := &output
  run_nbody(argv)
end
endif
procedure fannkuch(n)
maxFlipsCount := checksum := 0
permSign := 1
perm1 := list(n)
every perm1[i := 1 to n] := i-1
count := copy(perm1)
repeat {
k := perm1[1]
if k := 0 then {
    perm := copy(perm1)
    flipsCount := 1
    kk := perm[k + 1]
    while kk := 0 do {
        # reverse elements 1..k+1
        top := k+2
        every i := 1 to (k+1)/2 do
            perm[i] := perm[top-i]
        flipsCount += 1
        k := kk
        kk := perm[kk+1]
    }
    if maxFlipsCount < flipsCount then {
        maxFlipsCount := flipsCount
    }
    if permSign = 1 then {
        checksum += flipsCount
    }
    else {
        checksum -= flipsCount
    }
}
# generate another permutation via incremental change
flag := 1
if permSign = 1 then {
    permSign := 0
}
else {
    permSign := 1
    every r := 3 to n - 1 do {
        if count[r] := 0 then {
            flag := 0
            break
        }
        count[r] := r-1
        perm0 := pop(perm1)
        insert(perm1, r + 1, perm0)
    }
    if flag = 1 then {
        r := n
        if count[r] := 0 then {
            write(output, checksum)
            return maxFlipsCount
        }
        count[r] := 1
    }
}
procedure run_fannkuch(av)
    n := integer(av[1])
    write(output, "Pfannkuchen(" , n , ") = ", fannkuch(n))
end

 ifdef MAIN
procedure main(av)
    output := &output
    run_fannkuch(av)
end
$endif
# meteor-contest.icn

# The Computer Language Benchmarks Game
# http://shootout.alioth.debian.org/
# Translated from Olof Kraigher’s
# Python program

global width, height, masksAtCell
global solutions, masks, directions
global rotate, flip, moves, pieces

record xy(x, y)

procedure findFreeCell(board)
  bitposn := 1
every y := 0 to height - 1 do {
every x := 0 to width - 1 do {
  if iand(board, bitposn) = 0 then
    return xy(x, y)
  bitposn := ishift(bitposn, 1)
}
}
end

procedure floodFill(board, coords)
  local bitposn
  x := coords.x
  y := coords.y
  bitposn := ishift(1, x + width * y)
  if (not valid(x,y)) | (iand(board, bitposn) ˜= 0) then
    return board
  board := ior(board, bitposn)
every board := ior(board, floodFill(board, (!moves)(x, y)))
  return board
end

procedure noIslands(mask)
  zeroes := zerocount(mask)
  if zeroes < 5 then fail
  while mask ˜= 16r3FFFFFFFFFFFFF do {
    mask := floodFill(mask, findFreeCell(mask))
    new_zeroes := zerocount(mask)
    if (zeroes - new_zeroes) < 5 then fail
    zeroes := new_zeroes
  }
  return
end

procedure getBitmask(x,y,piece)
  mask := ishift(1, (x + width*y))
every cell := !piece do {
  results := moves[cell](x,y)
  x := results.x
  y := results.y
  if (0 <= x < width) & (0 <= y < height) then # valid
    mask := ior(mask, ishift(1, (x + width*y)))
  else {
    fail
  }
}
  return mask
end

procedure allBitmasks(piece, color)
  bitmasks := []
every !2 do {
  every rotations := 1
  to (6 - 3*(if color = 4 then 1 else 0)) do {
    every y := 0 to height - 1 do {
      every x := 0 to width - 1 do {
        if noIslands(mask := getBitmask(x, y, piece))
        then {
          put(bitmasks, mask)
        }
      }
    }
  }
  every piece[cell := 1 to *piece] := rotate[piece[cell]]
}
  every piece[cell := 1 to *piece] := flip[piece[cell]]
}
return bitmasks

end

procedure generateBitmasks()
  local color := 0
  every piece := !pieces do {
    m := sort(allBitmasks(piece, color))
    cellMask := ishift(1, (width*height-1))
    cellCounter := width*height - 1
    j := *m
    while j > 0 do {
      if iand(m[j], cellMask) = cellMask then {
        put(masksAtCell[cellCounter + 1, color+1], m[j])
        j -:= 1
      } else {
        cellMask := ishift(cellMask, -1)
        cellCounter -:= 1
      }
    }
    color +:= 1
  }
end

procedure solveCell(cell, board, n)
  if *solutions >= n then {
    return
  }
  if board = 16r3FFFFFFFFFFFFF then {
    s := stringOfMasks(masks)
    put(solutions, s, s) # inverse(s))
    return
  }
  if iand(board, ishift(1, cell)) ˜= 0 then {
    solveCell(cell-1, board, n)
    return
  }
  if cell < 0 then {
    return
  }
  every color := 1 to 10 do {
    if masks[color] = 0 then {
      every mask := !masksAtCell[cell + 1, color] do {
        if iand(mask, board)=0 then {
          # legal
          masks[color] := mask
          solveCell(cell-1, ior(board, mask), n)
          masks[color] := 0
        }
      }
    }
  }
end
procedure solve(n)
generateBitmasks()
solveCell(width*height-1, 0, n)
end

procedure stringOfMasks(masks)
s := ""
mask := 1
every !height do {
every !width do {
every color := 0 to 9 do {
    if iand(masks[color+1], mask) <> 0 then {
    s ||:= color
    break
    }
    else if color = 9 then
    s ||:= "."
    }
    mask := ishift(mask, 1)
}
}
return s
end

procedure inverse(s)
ns := s
write(output, image(s))
every x := 0 to width - 1 do
  every y := 0 to height - 1 do {
    ns[(x + y*width) + 1] := s[(width-x-1 + (width - y - 1)*width) + 1]
  }
return s
end

procedure printSolution(solution)
every y := 0 to height - 1 do {
every x := 0 to width - 1 do
  writes(output, solution[(x + y*width) + 1], " ")
  if (y%2) = 0 then {
    write(output, )
    writes(output, " ")
  } else
    write(output, )
}
end

procedure valid(x, y)
return (0 <= x < width) & (0 <= y < height)
end

procedure legal(mask, board)
return iand(mask, board) = 0
end

procedure zerocount(mask)
static zeros_in_4bits
local sum := -2
initial zeros_in_4bits :=
[
4, 3, 3, 2, 3,
2, 2, 1, 3, 2,
2, 1, 2, 1, 1,0
]
every x := 0 to 48 by 4 do {
    sum += zeros_in_4bits[1 + ishift(ishift(15, x), mask), -x]
}

return sum
end

procedure move_E(x, y)
    return xy(x+1, y)
end

procedure move_W(x, y)
    return xy(x-1, y)
end

procedure move_NE(x, y)
    return xy(x+(y%2), y-1)
end

procedure move_NW(x, y)
    return xy(x+(y%2)-1, y-1)
end

procedure move_SE(x, y)
    return xy(x+(y%2), y+1)
end

procedure move_SW(x, y)
    return xy(x+(y%2)-1, y+1)
end

procedure run_meteorcontest(argv)
    if *argv < 1 then stop("usage: meteor-contest num")
    width := 5
    height := 10

directions :=
    table(
        "E", 0,
        "NE", 1,
        "NW", 2,
        "W", 3,
        "SW", 4,
        "SE", 5
    )

rotate :=
    table(
        "E", "NE",
        "NE", "NW",
        "NW", "W",
        "W", "SW",
        "SW", "SE",
        "SE", "E"
    )

flip :=
    table(
        "E", "W",
        "NE", "NW",
        "NW", "NE",
        "W", "E",
        "SW", "SE",
        "SE", "SW"
    )

moves :=
    table(
        "E", move_E,
        "W", move_W,
        "NE", move_NE,
        "NW", move_NW,
        "SE", move_SE,
        "SW", move_SW
    )

pieces := [

]
solutions := []
masks := list(10, 0)
masksAtCell := list(width * height)
every !masksAtCell := [ [], [], [], [], [], [], [], [], [] ]
solve(argv[1])
write(output, *solutions, " solutions found\n")
printSolution(min(solutions))
write(output, )
printSolution(max(solutions))
write(output, )
end

 ifdef MAIN
procedure main(argv)
 output := &output
 run_meteorcontest(argv)
end
 endif
procedure genRandom()
        static ia, ic, im, imf, seed
        initial {
            ia := 3877
            ic := 29573
            im := 139968
            seed := 42
            imf := real(im)
        }
        seed := (seed * ia + ic) % im
        return seed / imf
end

procedure makeCumulative (genelist)
        local P := [], C := [], prob := 0.0
        every i := !genelist do {
            prob +:= i.p
            put(P, prob)
            put(C, i.c)
        }
        return [P, C]
end

procedure repeatFasta(src, n)
        width := 60
        r := *src
        s := src || src || src[1 : n % r + 1]
        every j := 0 to integer(n / width)-1 do {
            i := j * width % r
            write(output, s[i + 1 +: width])
        }
        if n % width ≠ 0 then
            write(output, s[-(n % width) : 0])
end

procedure randomFasta(genelist, n)
        local width := 60
        local results := makeCumulative(genelist)
        pros := results[1]
        chars := results[2]
        every 0 to integer(n / width)-1 do {
            x := ""
            every i to width do
                x ||:= chars[bisect(pros, genRandom())]
            write(output, x)
        }
        if n % width ≠ 0 then {
            y := ""
            every i to n%width do
                y ||:= chars[bisect(pros, genRandom())]
            write(output, y)
        }
end

procedure bisect(L, x)
        i := 1
        while L[i] < x do i += 1
        return i
end
record aminoacids(p, c)

procedure run_fasta(argv)
  alu := "GGCCGCGCAGGTGGCTCAAGAATCTCCGGCACTTTG_
         GAGGCCGAGGCGGGCGGATCACCTGAGGTCAGGAGTTCGAGA_
         CCAGCTGGCCAACATGCTGGGAACCCCGCTCTCTCTCTAAAAAT_
         ACAAAAAATGACGGCGGCGGTGCGGGGCGGACTCCGATCTCC_
         GTACTCGGGAGGGCCAGGAGCAGGAGGAGAATCGCTTGAACCCGGG_
         AGGCCGAGGAGTTGCGAGGAGGCCGAGTCCGCCGACTGACTCC_
         AGCCGGCCGACAGAGCGAGACTCCGTCCTCAAAAA"

  iub := [ aminoacids(0.27, "a"),
           aminoacids(0.12, "c"),
           aminoacids(0.12, "g"),
           aminoacids(0.27, "t"),
           aminoacids(0.02, "B"),
           aminoacids(0.02, "D"),
           aminoacids(0.02, "H"),
           aminoacids(0.02, "K"),
           aminoacids(0.02, "M"),
           aminoacids(0.02, "N"),
           aminoacids(0.02, "R"),
           aminoacids(0.02, "S"),
           aminoacids(0.02, "V"),
           aminoacids(0.02, "W"),
           aminoacids(0.02, "Y")
     ]

  homosapiens := [ aminoacids(0.3029549426680, "a"),
                   aminoacids(0.1979883004921, "c"),
                   aminoacids(0.197547306391, "g"),
                   aminoacids(0.3015094502008, "t")
     ]

  n := integer(argv[1])
  write(output, ">ONE Homo sapiens alu")
  repeatFasta(alu, n*2)
  write(output, ">TWO IUB ambiguity codes")
  randomFasta(iub, n*3)
  write(output, ">THREE Homo sapiens frequency")
  randomFasta(homosapiens, n*5)
end

$ifdef MAIN
procedure main(argv)
  output := &output
  run_fasta(argv)
end
$endif
procedure eval_A (i, j)
    return 1.0 / ((ishift((i + j) * (i + j + 1), -1) + i + 1))
end

procedure eval_A_times_u (u, resulted_list)
    u_len := *u
    every i := 0 to u_len - 1 do {
        partial_sum := 0
        every j := 0 to u_len - 1 do {
            partial_sum += eval_A(i, j) * u[j + 1]
        }
        resulted_list[i + 1] := partial_sum
    }
end

procedure eval_At_times_u (u, resulted_list)
    u_len := *u
    every i := 0 to u_len - 1 do {
        partial_sum := 0
        every j := 0 to u_len - 1 do {
            partial_sum += eval_A(j, i) * u[j + 1]
        }
        resulted_list[i + 1] := partial_sum
    }
end

procedure eval_AtA_times_u (u, out, tmp)
    eval_A_times_u (u, tmp)
    eval_At_times_u (tmp, out)
end

procedure run_spectralnorm(av)
    n := integer(av[1])
    u := list(n, 1.0)
    v := list(n, 1.0)
    tmp := list(n, 1.0)
    every 1 to 10 do {
        eval_AtA_times_u (u, v, tmp)
        eval_AtA_times_u (v, u, tmp)
    }
    vBv := vv := 0
    every i := 1 to n do {
        vi := v[i]
        vBv += u[i] * vi
        vv += vi * vi
    }
    write(output, sqrt(vBv/vv))
end
$ifdef MAIN
procedure main(av)
    output := &output
    run_spectralnorm(av)
$endif
end
$endif
procedure mapseq(str)
    head := ""
    ns := ""
    H := []
    M := []
    chars := &letters ++ '> '
    str ? {
        while tab(upto(chars)) do {
            if c := move(1) == '>' then {
                if *head > 0 then {
                    put(M, ns)
                    ns := ""
                }
                head := tab(many(chars))
                head := c || head
                put(H, head)
            } else {
                ns ||:= tab(many(chars))
            }
        }
        put(M, ns)
    }
    every i := 1 to *M do {
        M[i] := reverse(map(M[i], "ACBDGHKMNSRUTVYacbdghkmnsrutwvy", "TGVHCDMKNSYAAABRTGVHCDMKNSYAAWBR"))
    }
    return [H, M]
end

procedure run_reversecomplement(argv)
    fin := open(argv[1])
    str := reads(fin, stat(argv[1]).size)
    L := mapseq(str)
    i := 1
    while i < 4 do {
        write(output, L[1,i])
        L[2, i] ? {
            while write(output, move(60))
                s := tab(0)
                if *s > 0 then
                    write(output, s)
            i +=: 1
        }
    }
end

#undef MAIN
procedure main(argv)
    output := &output
    run_reversecomplement(argv)
end
#define MAIN
# mandelbrot.icn

The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/
Translated from Greg Buchholz’s C program

global w, h, wr, hr
$define iter 50
$define limit 2.0

procedure do_y(y)
    local bit_num, byte_acc, x, Zr, Zi, Cr, Ci, Tr, Ti, i, rv
    bit_num := byte_acc := 0
    rv := ""

    Ci := (2.0*y/hr - 1.0)
    every x := 0 to w-1 do {
        Zr := Zi := Tr := Ti := 0.0
        Cr := (2.0*x/wr - 1.5)
        every i := 0 to iter-1 do {
            if Tr+Ti > 4.0 then break
            Zi := 2.0 * Zr * Zi + Ci
            Zr := Tr + Ti + Cr
            Tr := Zr * Zr
            Ti := Zi * Zi
        }
        byte_acc := ishift(byte_acc, 1)
        if Tr+Ti <= 4.0 then {
            byte_acc := ior(byte_acc, 1)
        }
        bit_num +:= 1
        if bit_num = 8 then {
            rv ||:= char(byte_acc)
            byte_acc := bit_num := 0
        }
    }
    if bit_num `≠` 0 then {
        byte_acc := ishift(byte_acc, abs(8-w%8))
        rv ||:= char(byte_acc)
        byte_acc := bit_num := 0
    }
    return rv
end

procedure run_mandelbrot(argv)
    local y, i, pool
    pool := Pool(64)
    wL := list()
    rL := list()
    w := h := integer(argv[1])
    wr := hr := real(w)
    write(output, "P4\n", w, " ", h)
    every i := 0 to h-1 do
        put(wL, i)
    rL := pool.imap(do_y, wL)
    pool.kill()
    every i := !rL do
        writes(output, i)
end

$ifdef MAIN
procedure main(argv)

25
output := $output
run_mandelbrot(argv)
end
$endif
/* k-nucleotide.icn */
/* The Computer Language Shootout */
/* http://shootout.alioth.debian.org/ */
/* Translated from Ian Osgood, */
/* Sokolov Yura and bearophile’s */
/* Python program */

link printf, sort

procedure gen_freq(sequ, frame)
    ns := *sequ + 1 - frame
    frequences := table(0)
    every ii := 1 to ns do {
        nucleo := sequ[ii:ii+frame]
        if member(frequences, nucleo) then
            frequences[nucleo] += 1
        else
            frequences[nucleo] := 1
    }
    return [ns, frequences]
end

procedure rev(L)
    i := *L
    temp := []
    while i > 0 do{
        put(temp, L[i])
        i -= 1
    }
    return temp
end

procedure sort_sequ(sequ, length)
    local results, l, reverse
    results := gen_freq(sequ, length)
    n := results[1]
    frequences := results[2]
    l := sort(frequences, 2)
    l := sortff(l, 2, 1)
    l := rev(l)
    every pair := !l do {
        st := pair[1]; fr := pair[2];
        fprintf(output, "%s %.3r\n", st, 100.0*fr/n)
    }
    write(output, )
end

procedure find_sequ(sequ, s)
    local results := []
    results := gen_freq(sequ, *s)
    n := results[1]
    t := results[2]
    fprintf(output, "%d\t%s\n", t[s], s)
end

procedure run_knucleotide(argv)
    fin := open(argv[1])
    se := ["GGT", "GGTA", "GGTATT",
           "GGTATTTTAATT", "GGTATTTTAATTTATAGT"
    ]
    while line := read(fin) do {
        if line[1:4] == ">TH" then
            break
    }

27
sequ := ""
while line := read(fin) do
    sequ ||:= line
sequence := map(sequ, &lcase, &ucase)
every nl := 1 to 2 do
    sort_sequ(sequence, nl)
every s := !se do
    find_sequ(sequence, s)
end

$ifdef MAIN
procedure main(argv)
    output := &output
    run_knucleotide(argv)
end
$endif
procedure do_trim()
  outs := ""
  sequ ? {
    while outs ||:= tab(findp()) & mark := __endpoint do {
      tab(mark)
    }
    outs ||:= sequ[pos:0]
  }
  return copy(outs)
end

procedure do_variants(e)
  i := e[2](sequ)
  return e[1] || " " || i
end

procedure do_subst()
  every e := !subst do {
    outs := ""
    sequ ? {
      while outs ||:= tab(find(e.f)) do {
        outs ||:= e.r
        move(1)
      }
      outs ||:= sequ[pos:0]
    }
    sequ := copy(outs)
  }
  return sequ
end

procedure findp(p, subj:&subject, i1:&pos, i2:0)
  alphaspace := &letters ++ ' '
  every i := upto('>
  ', subj, i1, i2) do {
    case subj[i] of {
      ">": {
        i3 := many('>', subj, i, i2)
        i3 := many(alphaspace, subj, i3, i2)
        if not i3 := many('\\n', subj, i3, i2) then {
          next
        }
        __endpoint := i3
      }
      "\\n": {
        i3 := many('\\n', subj, i, i2)
        __endpoint := i3
      }
    }
    suspend i
  }
end
generate all the positions in s at which this pattern matches, in order

\[\text{aggtaaa|tttaccct}\]

**procedure var_1(s)**

local \(i := 0\)

\(s ? \{\) every find("aggtaaa"|"tttaccct") do \(i := i + 1\) \(\}\) return \(i\) end

\[\text{"[cgt]gggtaaa|tttaccc[acg]"},\]

**procedure var_2(s)**

local \(i := 0\)

\(s ? \{
\) every tab(upto('cgt')+1) & "gggtaaa" do \(i := i + 1\)
\&pos := 1
\) every tab(find("tttaccct")+7) & tab(any('acg')) do \(i := i + 1\)
return \(i\) \(\}\) end

\[\text{"a[act]ggtaaa|tttacc[agt]t"},\]

**procedure var_3(s)**

local \(i := 0\)

\(s ? \{
\) every tab(find("a")+1) & tab(any('act')) & "ggtaaa" do \(i := i + 1\)
\&pos := 1
\) every tab(find("tttacc")+6) & tab(any('agt')) & "t" do \(i := i + 1\)
return \(i\) \(\}\) end

\[\text{"ag[act]gtaaa|tttac[agt]ct"},\]

**procedure var_4(s)**

local \(i := 0\)

\(s ? \{
\) every tab(find("ag")+2) & tab(any('act')) & "gtaaa" do \(i := i + 1\)
\&pos := 1
\) every tab(find("tttac")+5) & tab(any('agt')) & "ct" do \(i := i + 1\)
return \(i\) \(\}\) end

\[\text{"agg[act]taaa|tttac[agt]ct"},\]

**procedure var_5(s)**

local \(i := 0\)

\(s ? \{
\) every tab(find("agg")+3) & tab(any('act')) & "taaa" do \(i := i + 1\)
\&pos := 1
\) every tab(find("ttta")+4) & tab(any('agt')) & "cct" do \(i := i + 1\)
return \(i\) \(\}\) end

\[\text{"aggg[acg]aaa|ttt[agt]ccct"},\]

**procedure var_6(s)**

local \(i := 0\)

\(s ? \{
\) every tab(find("aggg")+4) & tab(any('acg')) & "aaa" do \(i := i + 1\)
\&pos := 1
\) every tab(find("ttt")+3) & tab(any('agt')) & "ccct" do \(i := i + 1\)
return \(i\) \(\}\) end
procedure var_7(s)
    local i := 0
    s ? {
        every tab(find("agggt")+5) & tab(\(cgt\)) & ="aa" do i += 1
        &pos := 1
        every tab(find("tt")+2) & tab(\(acg\)) & ="accct" do i += 1
        return i
    }
end

procedure var_8(s)
    local i := 0
    s ? {
        every tab(find("aggta")+6) & tab(\(cgt\)) & ="a" do i += 1
        &pos := 1
        every tab(find("t")+1) & tab(\(acg\)) & ="taccct" do i += 1
        return i
    }
end

procedure var_9(s)
    local i := 0
    s ? {
        every tab(find("agggtaa")+7) & tab(\(cgt\)) do i += 1
        &pos := 1
        every tab(upto(\(acg\))+1) & ="ttaccct" do i += 1
        return i
    }
end

procedure run_regexdna(av)
    fin := open(av[1]) | stop("usage: regex-dna filename")
    siz := stat(av[1]).size | stop("can't stat ", av[1])
    sequ := reads(fin, siz) | stop("can't read ", av[1])
    pool := Pool(9)
    p := ">+[A-Za-z ]*\n+|\n+"
    llen := +sequ

    outs := ""
    sequ ? {
        while outs ||:= tab(findp()) & mark := __endpoint do {

            tab(mark)
        }
        outs ||:= sequ[&pos:0]
    }
    sequ := copy(outs)
    clen := *sequ

    variants := ["agggtaaa|tttaccct", var_1],
                ["agggtaaat[acg]taccct", var_2],
                ["a[act]ggttaaattttt[acg]t", var_3],
                ["a[act]ggttaaattttt[acg]tct", var_4],
                ["agg[act]ttaaatttttt[acg]tct", var_5],
                ["agg[acg]aaaatttttttt[acg]tct", var_6],
                ["aggg[acg]aaaatttttttt[acg]tct", var_7],
                ["agggtaaat[acg]taccc", var_8],
                ["agggtaaat[acg]taccc", var_9]

    subst := { [sp("B", "(c|g|t)")],
                [sp("D", "(a|g|t)")],
                [sp("H", "(a|c|t)")],
                [sp("K", "(g|t)"),
                [sp("M", "(a|c)"),
                [sp("N", "(a|c|g|t)"),

31
sp("R", "(a|g)"),
sp("S", "(c|g)"),
sp("V", "(a|c|g)")
sp("W", "(a|t)")
sp("Y", "(c|t)"
)

wL := []
every e := !variants do put(wL, e)

rL := pool.imap(do_variants, wL)
pool.kill()
every write(output, !rL)

every e := !subst do {
outs := ""
sequ ? {
while outs ||:= tab(find(e.f)) do {
outs ||:= e.r
move(1)
}
outs ||:= sequ[&pos:0]
}
sequ := copy(outs)
}

write(output, \n", ilen)
write(output, "", clen)
write(output, *sequ)
end

$ifdef MAIN
procedure main(av)
output := &output
run_regexdna(av)
end
$endif
The Computer Language Benchmarks Game
http://benchmarksgame.alioth.debian.org/
Translated from Mario Pernici’s Python program

link printf

procedure run_pidigits(av)
N := integer(av[1])
i := k := ns := a := t := u := 0
k1 := n := d := 1
repeat {
k := 1
   t := ishift(n,1)
   n := k
   a := t
   k1 := 2
   a := k1
   d := k1
   if a >= n then {
      cse := n * 3 + a
      t := cse / d
      u := cse % d + n
      if d > u then {
         ns := ns * 10 + t
         i := 1
         if i % 10 = 0 then {
            fprintf(output, "%010d	:%d
", ns, i)
            ns := 0
         } else if i >= N then {
            fprintf(output, "%-10dxx	:%d
", ns, i)
            break
         }
         a := (a - d * t) * 10
         n := 10
      }
   }
}
end

$ifdef MAIN
procedure main(av)
   output := &output
   run_pidigits(av)
end
$endif
global creature_colors, compl_dict, in_lock, out_lock, venue

procedure complement(c1, c2)
  if c1 == c2 then return c1
  if c1 == "blue" then {
    if c2 == "red" then return "yellow"
    return "red"
  }
  if c1 == "red" then {
    if c2 == "blue" then return "yellow"
    return "blue"
  }
  if c2 == "blue" then return "red"
  return "blue"
end

procedure check_complement()
  every c1 := !creature_colors do
    every c2 := !creature_colors do
      write(output, c1, " + ", c2, " -> ", compl_dict[c1||c2])
    write(output, )
  end
end

procedure spellout(n)
  numbers := ["zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine"]
  s := " "
  every c := !string(n) do
    s ||:= numbers[c+1] || " "
  return s
end

procedure report(input_zoo, met, self_met)
  s := " 
  every j := !input_zoo do
    s ||:= j || " 
  write(output, s)
  every i := 1 to *met do
    write(output, met[i], spellout(self_met[i]))
    i := 0
  every i +:= !met
    write(output, spellout(i), "\n")
end

procedure creature(my_id, my_lock, in_lock, out_lock)
  repeat {
    lock_acquire(my_lock)
    lock_acquire(in_lock)
    critical venue: venue[1] := my_id
    lock_release(out_lock)
  }
end

procedure lock_object()
  repeat{
    @>>
    <<=@
  }
end
procedure lock_release(T)
  @>>T
end

procedure lock_acquire(T)
  <<@T
end

procedure let_them_meet(meetings_left, input_zoo)
  local c_no, met, self_met, colors, my_locks
  c_no := *input_zoo
  met := list(c_no, 0)
  self_met := list(c_no, 0)
  colors := copy(input_zoo)
  my_locks := []
  in_lock := thread lock_object()
  lock_acquire(in_lock)
  out_lock := thread lock_object()
  lock_acquire(out_lock)
  every 1 to *input_zoo do {
    put(my_locks, thread lock_object())
  }
  every ci := 1 to c_no do {
    thread creature(ci, my_locks[ci], in_lock, out_lock)
  }
  delay(0)
  lock_release(in_lock)
  lock_acquire(out_lock)

  critical venue: id1 := venue[1]
  while meetings_left > 0 do {
    lock_release(in_lock)
    lock_acquire(out_lock)
    critical venue: id2 := venue[1]
    if id1 <> id2 then {
      new_color := compl_dict[colors[id1]||colors[id2]]
      colors[id1] := new_color
      colors[id2] := new_color
      met[id1] += 1
      met[id2] += 1
    }
    else {
      self_met[id1] += 1
      met[id1] += 1
    }
    meetings_left -= 1
    if meetings_left > 0 then {
      lock_release(my_locks[id1])
      id1 := id2
    }
    else{
      report(input_zoo, met, self_met)
    }
  }
end

procedure chameneosiate(n)
  check_complement()
  let_them_meet(n, ["blue", "red", "yellow"])
  let_them_meet(n, ["blue", "red", "yellow", "red", "yellow",
                  "blue", "red", "yellow", "red", "blue"])
end

procedure run_chameneos(argv)
  venue := mutex([-1])
  creature_colors := ["blue", "red", "yellow"]
  compl_dict := table("blueblue", complement("blue", "blue"),
"bluered", complement("blue", "red"),
"blueyellow", complement("blue", "yellow"),
"redblue", complement("red", "blue"),
"reddred", complement("red", "red"),
"redyellow", complement("red", "yellow"),
"yellowblue", complement("yellow", "blue"),
"yellowred", complement("yellow", "red"),
"yellowyellow", complement("yellow", "yellow")

camenesosiate(integer(argv[1]))
end

ifdef MAIN
procedure main(argv)
 output := &output
  run_chameneos(argv)
end
endif
# thread-ring.icn

# The Computer Language Benchmarks Game
# http://shootout.alioth.debian.org/
# Translated from Antti Kervinen and
# Tupteq's Python program

global main_lock, first_lock, n
procedure threadfun(number, my_lock, next_lock)
repeat {
    critical my_lock: wait(my_lock)
    if n > 0 then {
        n -= 1
        signal(next_lock)
    } else {
        write(output, number)
        signal(main_lock)
    }
}
end

procedure run_threadring(argv)
n := argv[1]
main_lock := condvar()
next_lock := first_lock := condvar()
every number := 1 to 503 do {
    my_lock := next_lock
    next_lock := if number < 503 then condvar() else first_lock
    thread threadfun(number, my_lock, next_lock)
}
signal(first_lock)
critical main_lock: wait(main_lock)
end

ifdef MAIN
procedure main(argv)
output := &output
run_threadring(argv)
end
endif
record nodes(a, b, c)

procedure make_tree(i, d)
  if d > 0 then {
    i2 := i + i
    d := d - 1
    return nodes(i, make_tree(i2 - 1, d), make_tree(i2, d))
  }
  return nodes(i, &null, &null)
end

procedure check_tree(nodelst)
  local i := nodelst[1], l := nodelst[2], r := nodelst[3]
  if /l then
    return i
  else
    return i + check_tree(l) - check_tree(r)
  end
end

procedure make_check(i, d)
  return (check_tree(make_tree(i, d)))
end

record ca(a, b)
record cb(a, b)

procedure get_argchunks(i, d)
  local chunksize:= 5000, chunk := []
  every k := 1 to i do {
    put(chunk, cb(ca(k, d), ca(-k, d)))
    if *chunk = chunksize then {
      suspend chunk
      chunk := []
    }
    }
  if *chunk > 0 then
    suspend chunk
end

procedure build_trees(d)
  local i, cs
  i := 2 ^ (mmd - d)
  cs := 0
  every argchunk := !get_argchunks(i, d) do {
    cs := make_check(argchunk[1,1], argchunk[1,2])
    cs := make_check(argchunk[2,1], argchunk[2,2])
  }
  return (i*2 || " trees of depth " || d || "\t check: " || cs )
end

global min_depth, max_depth, stretch_depth, mmd

procedure run_binarytrees(argv)
  local wL := [], rL := []
  min_depth := 4
  max_depth := argv[1]
  stretch_depth := max_depth + 1
mmd := max_depth + min_depth

write(output, "stretch tree of depth ", stretch_depth,
"\t check: ", make_check(0, stretch_depth))

every d := min_depth to stretch_depth by 2 do
  put(wL, d)
  pool := Pool(*wL)
  rL := pool.imap(build_trees, wL)
  pool.kill()
  every write(output, !rL)
  long_lived_tree := make_tree(0, max_depth)
  write(output, "long lived tree of depth ", max_depth,
"\t check: ", check_tree(long_lived_tree))
end

$ifdef MAIN
procedure main(argv)
  output := &output
  run_binarytrees(argv)
end
$endif

#################################################
# Optional sizes of environment variable BLKSIZE
# to reduce garbage collections
#
# 1GB
# export "BLKSIZE=1073741824"
#
# 2GB
# export "BLKSIZE=2147483648"
#
# 3GB
# export "BLKSIZE=3221225472"
#
#################################################

#################################################
# To potentially giving each thread block
# region of their own
#
# # 40MB block region
# #pool := Pool(*wL, 41943040)
# # 400MB block region
# #pool := Pool(*wL, 419430400)
# #1GB block region
#################################################