First year project
Project 99: Who's Julia?

Group: 99a
Simon Lehmann Knudsen, simkn15
Sonni Hedelund Jensen, sonje15
Asbjørn Mansa Jensen, asjen15
DM501

May 25, 2016

Contents

1 Introduction

Introduction Julia is a new open source object orientated programming language which shares many similarities with python. The language is made for high performance and scientific computations while still supporting general purpose programming.

Julia supports the following:

Multiple dispatch and dynamic typing. Julia has a dynamic type system like python. A property of dynamic typing is that the type checks which are performed at runtime, mostly - opposed to static typing which makes the type checks at compile time. In Julia it is possible to specify types but for the most part it is not necessary because of the dynamic type system. Julia also makes decisions about what size to assign to the variables and but this does not change in run time and the user cannot change it after initialization. The types that are supported by Julia can be seen in the table below.

· Signed / unsigned integers of; 8, 16, 32, 64 and 128 bits. · Floating points of 16, 32 and 64 bits. · Boolean – 8 bits . Char – 32 bits

Julia will by default assign the middle value if no more is needed. Julia will be default assign 32 bits to an integer or floating if no more memory is needed at initialization. Julia has abstract types – an abstract type is just some from of generalization of a certain type. For example 16, 32 and 64 bit floats are an AbstractFloat. This is especially useful when the programmer doesn't how much memory a certain variable may need. To little memory will result in a bug and to much memory is a waste. FÅ DET TIL AT VÆRE SAMMENHÆNGENDE HER Julia supports multiple dispatch which is used to determine which function to call by the type of one or more of the parsed argument(s). This is needed when for example two functions with the same name are declared:

function a(arg1::Int8) println("Int") end

function a(arg1::Float16) println("Float"); end

If a variable is declared with a type of Int8 and parsed to the function a(...), then it will print "Int", and if the variable is declared as Float16 it will print "Float". It is also possible to use abstract types here, so arg1::Float16 and arg1::Int8 could be changed to arg1::AbstractFloat and arg1::AbstractInt – this will make the function 'a' accept any kinds of floats and ints.

User defined types: Another feature of Julia is user defined types, also known as composite types which gives the programmer the option to defined new types. A composite type in Julia may look something like the following:

type person age::Int16 name end

If a variable in a composite type is not specified with any type, the default is ::Any which accepts any kinds of types. To initialize a composite type you can treat it like an object. Person = person(21, "Carl") And the values can be changed with Person.age = 25 The developer of Julia states that user defined types are as fast and compact as built in types.

Garbage collector Julia uses a garbage collector to automatically free the memory when needed. There is no guarantee when the garbage collector will run, but is it possible to force garbage collection with a function call: Gc() One thing to keep in mind is that once a name is defined in Julia, it will always be present till termination. The garbage collector doesn't free the memory of unreachable object but rather reallocates memory for objects when memory size has changed. So the only way the garbage collector can free memory is when an object has been reduced in size, for example: A = rand(float32, 10000, 10000) A = 0; Gc() This code will generate a 10000x10000 matrix filled with random 32 bit floating points which take a bit of memory. A will then be set to 0 but will keep the memory size of the 10000x10000 matrix until garbage collection is done either manually as in this example or somewhere in the future when it is done automatically.

Built-in package manager Julia comes with a built in package manager which keeps track of which packages that needs to be included in the users program to run, so is it not necessary to state which packages that need to be included. For example when the user calls the sort function, no package include statement or math.sort() is needed – this is done automatically by the package manager.

Lightweight green threading Julia are using lightweight green threading. A thread is lightweight when it shares address space with other thread opposed to heavyweight threads which has its own address space. When threads shares the same address space communication between then is faster and much simpler, for communication between heavyweight thread pipes or sockets are needed. Green threads are threads that are scheduled not by the operating system but rather by the runtime library or virtual machine. This means that green threads does not have to relay on the operating system and can be controlled much better – this means that the threads are in user space and not kernel space. As of Julia 0.4.5 multithreading is not available but in the unstable version 0.5.0 an experimental support of multithreading is available.

Meta-programming and Macros Julia allows the use of meta-programming. Meta-programming is a way to write programs in programs and let them use program code as data. In Julia meta-programming can be done by defining macros. For example, the @time macro was used extensively in the project. The @time is in the standard Julia library. The @time takes a function as an argument and sets a timer in the top of the code from the argument and stops the timer in the bottom and prints the time passed and memory allocated. Meta-programming and macros are known from Lisp, a programming language, and have been used in early AI research.

Implementing code from other languages Julia is a new language which will mean there are not many libraries written in Julia. Julia has an import feature for both Python and C libraries to avoid this disadvantage. Import the library PyCall with the "using" operation to use python and import Python libraries via a macro @pyimport. To use C libraries or coding, simply run the function ccall(). The programming language C is a well known and used language. Many systems are based on C, which makes C an advantage for hardware programming. Another feature is to execute shell commands. Execute shell commands using run(“). An example would be running run(‘echo Hello World‘) which would return the output "Hello World".

Conclusion It can be difficult to make any conclusions by the benchmarking but it may give an idea of how Julia performs, compared to Java, Python and C++. One thing to keep in mind is that as of writing this report, Julia is in version 0.4.5 - it has yet to reach version 1.0.0 while the other languages are much older than Julia. This could mean that Julia might get even faster in the future.

Personal experience with Julia Some of the design choices are really frustrating when adapting to Julia. An example would be the index starting at 1 - this could be debatable because of the statement that Julia is used in a lot in scientific computations, but as a programmer who is used to start at index 0, this will cause a lot of bugs. The developers of Julia made a weird design choice to the for-loop, when you want to loop downwards. In most programming language it will look something like

for i = n; i > 0; i–

But in Julia they switch the increment / decrement part with the exit condition

for i = n : -1 : 0

So in both examples the loops start at n, decrements down to 0 and stops. This kind of syntax change can be the cause of a lot of bugs, when programming in Julia. This is a simple thing that is easy to overcome, but a really big problem with Julia is the documentation. It feels far from complete. The documentation doesn't explain much but instead gives some examples. The problem is that it sometimes misses essential things like what types you'll have to parse for a certain function. It might tell you that you have to parse these arguments but forget to tell you what they are. An example from the documentation is:

"rand([rng] [, S] [, dims...])

Pick a random element or array of random elements from the set of values specified by S; S can be

an indexable collection (for example 1:n or ['x','y','z']), or a type: the set of values to pick from is then equivalent to typemin(S):typemax(S) for integers (this is not applicable to BigInt), and to [0,1) for floating point numbers; S defaults to Float64."

They explain what S is, but forgets about rng and dims.

Simple things like how to access data in an array at a specific index. This should be one of the first things in the Array section, but only gets shown after about 2 pages of text. If you have experience in other programming languages, you might know that you'll just have to type a[index] but this isn't so obvious for new programmers. It is clear that something has to be done to the documentation.

A downside of Julia is also their consistency. As seen with the for loops, but what is a bit worse is it is clear that the developer of Julia has tried to every remove burden of the programmer with the dynamic typing, package manager system etc. But with memory allocation it's really bad. First of all it does not make sense to make a garbage collector if it doesn't free unreachable objects. Also it does not make sense that you can't change the variable size and Julia won't do that automatically when more memory is need.

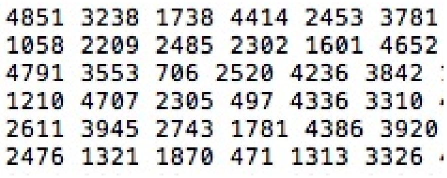
1.1 Julia

1.2 Features of Julia

1.3 Syntax differences

2 Projecteuler 11

In the 20x20 grid below, four numbers along a diagonal line have been marked in red. The product of these numbers is 26 x 63 x 78 x 14 = 1788696. What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20x20 grid?



```
4851 3238 1738 4414 2453 3781
1058 2209 2485 2302 1601 4652
4791 3553 706 2520 4236 3842
1210 4707 2305 497 4336 3310
2611 3945 2743 1781 4386 3920
2476 1321 1870 471 1313 3326
```

The program runs from command-line/terminal in order to be able to test on different inputs and amount of adjacent numbers multiplied. To run the Julia version from the terminal: **Julia euler11.jl 100 4**. This would make the program run a matrix of size 100 and calculating product of 4 adjacent numbers. There was made a matrix generator, found in appendix, which makes a file with data to test. The file would be named **mat100.txt** for data to a $100 \cdot 100$ matrix. The numbers in the file ranging from 100-9999, 3-4 digit numbers. The original problem states two digit numbers in the data, but this would make a lot of duplicated numbers when testing with larger inputs, e.g. a $5000 \cdot 5000$ matrix. Some programming languages optimizes code during run time if it can predict what a given result will be. Having many duplicated numbers in the dataset could have an influence of the benchmarking between languages. To avoid this situation, to some extent, the digits have been increased. The algorithm first reads through a file with the input and makes a matrix. To calculate products in all directions needed in the problem, the algorithm goes through the matrix a total of 3 times. A nested for loop is needed to go though a matrix, lines 1-2 at figure 1. The outer loop iterates through the rows, and the inner loop iterates through the columns. Figure 1 calculates the horizontal and vertical directions, firgure 2 diagonally from left to right(downwards) and figure 3 going diagonally from right to left(downwards). The algorithm starts in the most upper left cell, and iterates through all cells in the matrix. Lets have the first cell as (1,1), first number is row(**i**) and second number is column(**j**). Looking at figure 1, **matLength** is the size of the matrix, size = 100 would mean a matrix of size $100 \cdot 100$. **numProd** is the number of adjacent numbers multiplied together. For matrix with size 100, and multiplying four adjacent numbers, the algorithm would do the following in figure 1:

> Starting at cell (1,1) to the end which is cell (100, 100 - 4).
>
> Line 5-7: Loops over the adjacent cells and multiplies the numbers.
>
> Line 8-10: Sets current product to max product, if current is larger than previously max product.

```
 1  for i = 1 : matLength
 2    for j = 1 : matLength - numProd
 3      #right/left
 4      prod = 1
 5      for k = 0 : numProd - 1
 6        prod *= mat[i, j + k]
 7      end
 8      if prod > maxProd
 9        maxProd = prod
10      end
11      #up/down
12      prod = 1
13      for k = 0 : numProd - 1
14        prod *= mat[j + k, i]
15      end
16      if prod > maxProd
17        maxProd = prod
```

Figure 1: Horizontal and vertical

Figure 2 shows the loop that iterates over the matrix and calculating the product of the diagonal going downwards from left to right. Figure 3 shows the diagonal product going upwards from left to right. The loop starts in the first row, and starting column is the last, not the first like the previously loops.

```
#diagonal left->right
for i = 1 : matLength - numProd
  for j = 1 : matLength - numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j + k]
    end
    if prod > maxProd
      maxProd = prod
```

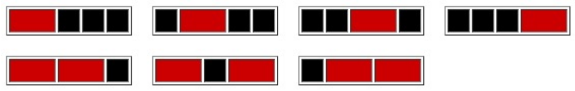
Figure 2: Diagonal downwards left to right

```
#diagonal right->left
for i = 1 : matLength - numProd
  for j = matLength : -1 : numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j - k]
    end
    if prod > maxProd
      maxProd = prod
```

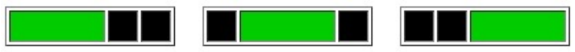
Figure 3: Diagonal upwards left to right

3 Projecteuler 116

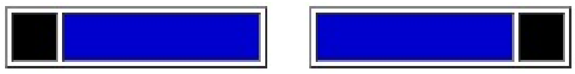
Description: A row of five black square tiles is to have a number of its tiles replaced with coloured oblong tiles from red(length two), green(length three), or blue(length four). If red tiles are chosen there are exactly seven ways. If green tiles are chosen there are three ways. And if blue tiles are chosen there are two ways. Figure 4 is a visualization of how the tiles can be lain. Assuming that colours cannot be mixed there are $7+3+2 = 12$ ways of replacing the black tiles in a row measuring five units in length. How many



(a) Red tiles



(b) Green tiles



(c) Blue tiles

Figure 4: Projecteuler: 116

different ways can the black tiles in a row measuring fifty units in length be replaced if colours cannot be mixed and at least one coloured tile must be used?

4 Quicksort

Quicksort applies the divide-and-conquer paradigm to sort numbers. Divide-and-conquer has three steps. **Divide** the problem into a number of subproblems that are smaller instances of the same problem. **Conquer** the subproblems by solving them recursively. If the subproblems sizes are small enough, however, just solve the subproblems in a straightforward manner. **Combine** the solutions to the subproblems into the solution for the original problem [1, chapter 4, page 65]. Description of quicksort: [1, chapter 7, page 170-171]

> **Divide**: Partition (rearrange) the array A[p..r] into two (possible empty) subarrays A[p..q-1] and A[q+1..r] such that each element of A[p–q1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
>
> **Conquer**: Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.
>
> **Combine**: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p..r] is now sorted.

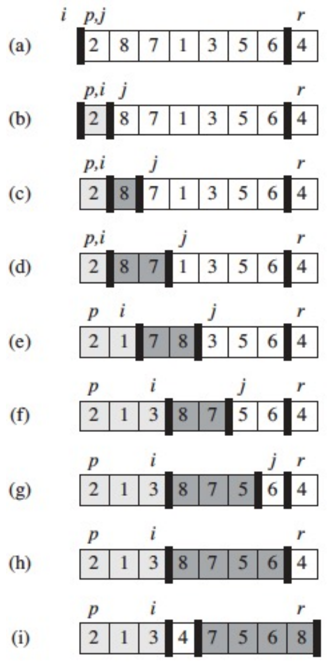


Figure 5: Quicksort

Figure 5 shows a visualization of quicksort.

5 Statistics

6 Learning / Personal experience

7 Conclusion

8 Appendix (source code)

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein *Introduction to Algorithms*, Cambridge, Massachusetts, third edition, 2009.