

First year project
Project 99: Who's Julia?

Group: 99a
Simon Lehmann Knudsen, simkn15
Sonni Hedelund Jensen, sonje15
Asbjørn Mansa Jensen, asjen15
FF501

June 1, 2016

1 Summary

Julia is a new programming language. Through tests and benchmarking it is found that Julia do some optimizations behind the scene but these optimizations does not seem as developed as Javas optimizations. In most of the tests in this report and the Project Euler solutions that was not included in the report showed that Julia is relatively fast even in this early stage of development. This is what the developers state and what seems like a lot of inspiration from Python, they managed to create an easy to learn and easy to write programming language.

Contents

1	Summary	i
2	Introduction	1
3	Julia	1
3.1	Syntax	3
3.2	Features	7
3.2.1	Dynamic typing and byte-related memory management	8
3.2.2	Multiple dispatch	9
3.2.3	User defined types	9
3.2.4	Garbage Collector	10
3.2.5	Built-in package manager	11
3.2.6	Lightweight green threading	11
3.2.7	Meta-programming and Macros	11
3.2.8	Implementing code from other languages	12
3.2.9	Designed for parallelism and distributed computation .	12
3.2.10	Supports Unicode	12
3.2.11	Devvectorized code is fast	13
4	Project Euler	13
5	Theory	14
5.1	Benchmarking	14
5.2	Profiling	14
5.3	Time measurement	14
6	Materials and methods	15
6.1	Benchmarking	15
7	Problems	16
7.1	Projecteuler 11	16
7.1.1	Solution	17
7.1.2	Results	19
7.2	Projecteuler 116	19
7.2.1	Solution	20
7.2.2	Results	21

7.3	Dual Pivot Quicksort	23
7.3.1	Results	26
7.4	Statistics	28
8	Personal experience with Julia	29
8.1	Sonni	29
8.2	Simon	31
8.3	Asbjørn	32
9	Discussion	32
10	Conclusion	32
11	Appendix	34
11.1	Process analysis	34
11.1.1	Projectmanagement, organizing workflow and organizing process of learning	34
11.1.2	Process of learning	34
11.1.3	Cooperation, difficulties in group and with supervisor	35
11.1.4	Conclusion	35
11.2	Copy of poster in A4	35

2 Introduction

Java, C++ and Python are on the top ten of the most used programming languages. There are hundreds of languages, but not minding the hard competition, new languages are still created with the thought of doing better. An example of this is the relatively new programming language Julia, which has been developed with the idea to combine the best features of other languages.

The purpose of this report is to find out how well Julia perform compared to some of the standard languages as of 2016. One of those languages are Python which Julia has a lot of inspiration from - syntax wise and the dynamic type system. The second language that Julia will be compared with is Java, these two does share similarities, but most of the similarities are behind the scene stuff, like garbage collection and compiler optimizations. The developers of Julia claim that Julia is as fast as C and therefore C++ is the last language that Julia will get compared with.

The report starts introducing Julia, Julia's background and features. Julia's introduction is followed by a brief introduction to Project Euler. Next, an explanation of the theory behind benchmarking, profiling and time measurement for the understanding of the rest of the report. The materials and methods used for benchmarking. The problems/algorithms used for benchmarking. The results, personal experiences learning Julia, discussion and conclusion.

3 Julia

In this section we are going to introduce why and when Julia was created. We will show some comparisons in the syntax with Python, C++ and Java; and explain some of the features listed below.

Features in Julia:

- Dynamic typing and byte-related memory management
- Multiple dispatch
- User defined types
- Garbage Collector
- Built-in package manager
- Lightweight green threading
- Meta-programming and Macros

- Julia is an object-oriented programming language, which has been under development since 2009. First released in 2012 and the newest stable version of Julia is version 0.4.5. The language has been created because the developers wanted a language with all the features they like from different languages. The developers wanted the language to be open source, which means everybody can read and modify the language. One of the ideas was to make the language as simple, readable and easy to learn as possible. The language is made for high performance and scientific computations while still supporting general purpose programming. Programming in Julia can be done in the terminal like Python with interactive mode:

(a) Julia

(b) Python

Writing longer programs in the terminal might not be the preferred method. Files containing Julia code can be interpreted by the Julia's interpreter with the following command from terminal:

2 of 35

The text editor Atom supports the Julia language. Atom has a package, **uber-juno**, which sets up Atom to act like an IDE, integrated development environment for Julia. However, the programmer can of course use whatever text editor he or she likes.

3.1 Syntax

Make examples of syntax between all the languages here.

1	Julia:	Python:	Java:	c++:
2	<code>a = 10</code>	<code>a = 10</code>	<code>int a = 10;</code>	<code>int a = 10;</code>

Figure 2: Declaring and assigning variables

```

1  for i = 1 : 10
2      println(i)
3  end

```

(a) Julia

```

1  for i in range(1, 11):
2      print i

```

(b) Python

```

1  for (int i = 1; i <= 10; i++){
2      System.out.println(i);
3  }

```

(c) Java

```

1  for (int i = 1; i <= 10; i++){
2      std::cout << i << "\n";
3  }

```

(d) C++

Figure 3: For-loops: Incrementing

```
1   for i = 10 : -1 : 1
2       println(i)
3   end
```

(a) Julia

```
1   for i in range(10, 0, -1):
2       print i
```

(b) Python

```
1   for (int i = 10; i > 0; i--){
2       System.out.println(i);
3   }
```

(c) Java

```
1   for (int i = 10; i > 0; i--){
2       std::cout << i << "\n";
3   }
```

(d) C++

Figure 4: For-loops: Decrementing


```
1      if i == a
2          println(a * b)
3      end
4      else if i == b
5          println(a / b)
6      else
7          println(a + b)
8      end
```

(a) Julia

```
1      if i == a:
2          print a * b
3      elif i == b:
4          print a / b
5      else:
6          print a + b
```

(b) Python

```
1      if(i == a){
2          System.out.println(a * b);
3      }
4      else if(i == b){
5          System.out.println(a / b);
6      }
7      else{
8          System.out.println(a + b);
9      }
```

(c) Java

```
1      if(i == a){
2          std::cout << a * b << "\n";
3      }
4      else if(i == b){
5          std::cout << a / b << "\n";
6      }
7      else{
8          std::cout << a + b << "\n";
9      }
```

(d) C++

Figure 5: If statements

```
1      int i = 0
2      while i < 10
3          i += 1
4      end
```

(a) Julia

```
1      int i = 0
2      while (i < 10):
3          i += 1
```

(b) Python

```
1      int i = 0;
2      while (i < 10){
3          i++;
4      }
```

(c) Java

```
1      int i = 0;
2      while (a < 10){
3          i++;
4      }
```

(d) C++

Figure 6: while-loop

```

1      function hello(name)
2          println("Hello $name")
3      end

```

(a) Julia

```

1      def hello(name):
2          print "Hello", name

```

(b) Python

```

1      public static void hello(String name){
2          System.out.println("Hello " + name);
3      }

```

(c) Java

```

1      void hello(std::string name)
2      {
3          std::cout << "Hello " << name << std
4              ::endl;
5      }

```

(d) C++

Figure 7: Declaring functions

```

1      array = fill(0, 100)

```

(a) Julia

```

1      array = [0 for col in range(100)]

```

(b) Python

```

1      int[] array = new int[100];
2      Arrays.fill(array, 0);

```

(c) Java

```

1      std::array<int,100> array;
2      array.fill(0);

```

(d) C++

Figure 8: Filling array

3.2 Features

In this section we are writing about some of Julia's features.

3.2.1 Dynamic typing and byte-related memory management

Dynamic typing is a type management approach where type checks are mostly performed at runtime, as opposed to static typing, where type checks are made at compile time. Dynamic typing allows the programmer to skip the type declaration and let the dynamic type system handle it. In Julia it is possible to specify types, but mostly it is not necessary. Julia makes decisions about how much memory to allocate for variables but the assigned memory size will not change. This means that if a value, that exceeds the allocated memory, is assigned to the variable after initialization, a bug will occur. Another point to notice is that the memory size can be decreased and this will be addressed later under the section Garbage Collector. However, there is a range of memory sizes to chose from when initializing a variable. Julia will by default assign 32 bits to an integer or a floating point if no more memory is needed at initialization, but it is possible to manually chose one of the following:

- Signed/unsigned integer - 8, 16, 32, 64 and 128 bits – Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Int128 and UInt128
- Floating point - 16, 32 and 64 bits – Float16, Float32 and Float64
- Boolean – 8 bits - Bool
- Character – 32 bits - Char

To initialize a variable with a given memory size:

```
1 x = Int8(10)
```

But if the variable is later assigned a new value, Julia will automatically allocate 32 bits for the variable or more if needed:

```
1 x = Int8(10)
2 typeof(x) #Int8
3 x = 4
4 typeof(x) #Int32
```

If no certain memory size is needed but only a specific type then abstract types will come in handy. An abstract type is a form of generalization of a concrete type. For example 16, 32 and 64 bit floating points are of type AbstractFloat. This is especially useful when the programmer do not know how much memory a certain variable may need. Too little memory will result

in a bug and too much memory is a waste.

* Example *

3.2.2 Multiple dispatch

Multiple dispatch is used to determine which function to call by the type of one or more of the parsed argument(s). This is useful when two functions, with the same name, are declared:

```
1 function a(arg1::Int8)
2     println("Int")
3 end
4
5 function a(arg1::Float16)
6     println("Float")
7 end
```

If a variable is declared as an **Int8** and used as argument in function **a()**, then Julia will print "Int". If the variable is declared as a **Float16** and used as argument in function **a()**, Julia will print "Float". It is possible to use abstract types, so **arg1::Float16** and **arg1::Int8** could be changed to **arg1::AbstractFloat** and **arg1::AbstractInt**, which will make the function 'a' accept any type of floating point and integer.

3.2.3 User defined types

User defined types, known as composite types, gives the programmer the option to define new types in Julia. A composite type in Julia could look as the following:

```
1 type person
2     age::Int16
3     name #a variable, which gets the assigned default type ::Any
4 end
```

If a variable in a composite type is not specified with any type, the default is **::Any**, which accept any type. To initialize a composite type, think of

it like an object. **Person** = **person**(21, "Carl"), and the values can be changed with "**Person.age** = 25". In the example above, the type **person** can now be used like any other of the built in types in Julia.

```
1 function a(arg1::person)
2     println(arg1.age)
3 end
4 Person = person(21, "Carl")
5 a(Person) #Will print 21
```

The developers of Julia claim that user defined types are as fast and compact as built in types.

3.2.4 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 it is possible to schedule garbage collection with a function call, **gc()**, and will be executed at some point. It is not possible to control exactly when to run the garbage collector, but only put it in queue. One thing to keep in mind is that once a variable is defined in Julia, it will always be present until program termination. The garbage collector does not free the memory of unreachable object, but rather reallocates memory for objects when memory size has changed. Therefore, the only way the garbage collector can free the memory is when an object has been reduced in size, for example:

```
1 A = rand(float32, 10000, 10000)
2 A = 0
3 gc()
```

This code will generate a 10.000×10.000 matrix filled with random 32 bit floating points, which consumes approximately 400MB of memory. When the value **A** is set to 0 the memory allocated will stay the same until the garbage collection is executed, as in the example above, or on program termination. After the execution of the garbage collector **A** will use memory equal to an integer of 32 bits, **Int32**.

3.2.5 Built-in package manager

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

3.2.6 Lightweight green threading

A thread is lightweight when it shares address space with other threads opposed to heavyweight threads, which has its own address space. If threads share the same address space, the communication between them is faster and much simpler. The communication between heavyweight threads have to go through pipes or sockets.

Green threads are threads that are scheduled not by the operating system but rather by the runtime library or the virtual machine. Green threads do not have to rely on the operating system and can be controlled much better – which means the threads are in user space and not kernel space. (See section 5.3 for explanation of time measurement) 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, therefore threading will be addressed no further.

3.2.7 Meta-programming and Macros

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` macro is a part of the Julia standard library. The `@time` macro takes a function as an argument and starts a timer before the code execution, stops the timer after execution 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.

****EXAMPLES HERE****

3.2.8 Implementing code from other languages

Since Julia is a newer language there are not many written libraries yet. Julia has an import feature for both Python and C libraries to decrease 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".

```

1 #Run Pkg.add("PyCall") for the first time, to install the package.
2 using PyCall
3
4 @pyimport pylab #Import library pylab
5
6 x = linspace(0, 4, 100); y = x + 1;
7 pylab.plot(x, y)
8 pylab.show() #Shows the graph x + 1
9
10 println(pyeval("4*3+1")) #prints 13

```

Figure 9: Import of Python libraries.

3.2.9 Designed for parallelism and distributed computation

The developers of Julia claim that Julia is designed for parallelism and distributed computation. Parallel computing is dividing a problem into smaller problems and solving them at the same time. Distributed computing is solving a problem using computers that are working together over a network. Julia have been trying to make this easier to achieve with different macros.

****WHAT MACROS??****

3.2.10 Supports Unicode

The Unicode support globalize the language. Unicode is an encoding system for characters. Unicode supports a large range of characters and is designed

to make it easier to read and write non-latin characters.

3.2.11 Devectorized code is fast

Vectorization of code is, for example, when a for loop is rewritten to sequential operations. Figure 10 shows an example of a non-vectorized for loop and the vectorized version. Vectorized code is usually faster. The developers of Julia claim that there is no reason to vectorize as the performance stays the same.

```
1      a = 1, 2, 3, 4
2      b = 2, 3, 4, 5
3      c is empty
4      for i = 1 to 4
5      begin
6          c[i] = a[i] + b[i]
7      end
```

(a) Devectorized code

```
1      a = [1, 2, 3, 4]
2      b = [2, 3, 4, 5]
3      c is empty
4      c[1] = a[1] + b[1]
5      c[2] = a[2] + b[2]
6      c[3] = a[3] + b[3]
7      c[4] = a[4] + b[4]
```

(b) Vectorized code

Figure 10: Vectorization

4 Project Euler

Project Euler is a website (<https://projecteuler.net/>) with a huge database of mathematical and programming challenges. In this report Project Euler has been used to find challenges for testing the different programming languages. From here on problems from Project Euler will be referred to as euler 'a number'. For example, problem 11 from Project Euler will be referred to as euler 11.

5 Theory

In this section we explain some of the theory behind Benchmarking, profiling and time measurement.

5.1 Benchmarking

******In computer science, benchmarking is experimental measuring of the amount of performance crammed by a particular program******. In this report an execution of a set of programs, implemented in Julia, Python, C++ and Java will be time measured. In order to get an indication of the real CPU time, multiple tests are done and the median is calculated. The median run time is considered since there can be fluctuations in run times for every test.

5.2 Profiling

Profiling is a way to analyze code. For example, the analysis could be measuring the memory usage, time and function calls. The analysis shows, which part of the code took the longest time or used up all memory. Profiling is often used for optimization.

5.3 Time measurement

When looking at the run time the most common terms are wall time (real time) and CPU time. Wall time is the time it took the program to terminate, from start to finish. CPU time is how much time the program used on the CPU. The time difference between wall time and CPU time comes from the fact that the program being benchmarked or profiled might get interrupted by the operating system, because other tasks needs to be handled. These tasks could be other programs, or something that the OS needs to get done. The processing time of those tasks will be included in the wall time but excluded in the CPU time. To get the run times in Unix based system, the time command is available from the terminal:

```
1 time <COMMAND>
```

Where <COMMAND> can be any terminal command. The time command will return three different times.

```
real    0m0.627s
user    0m0.462s
sys     0m0.147s
```

Figure 11: Sample output of the time command

- Real: This is the wall time mentioned above.
- User: User time is how much CPU time is spent in user mode.
- Sys: System time is how much CPU time is spent in kernel mode.

The difference between user- and kernel mode is the following:

In most memory protected systems there are some locked operations for security reasons. E.g. allocation of memory or accessing hardware requires kernel mode and cannot be done in user mode. Operations as **malloc** and reading/writing to files will be processed in the kernel and is counted as system time. This does not necessarily mean that all time is spent in kernel mode operating with file data, since the only time that is spent in kernel mode is writing the data from the file to memory. The rest of the operation is counted towards user time. Therefore to get the amount of time spent by the CPU, **User time + system time** is a good estimation. Another way to measure time spent on the CPU is to use the built in libraries. For example the **ThreadMXBean** in Java. This is especially useful for profiling.

6 Materials and methods

In this section we have written how we benchmarked and with what materials/software.

6.1 Benchmarking

Materials for benchmarking:

Lenovo W530: Intel Core i5-3320M 2.60GHz x 4, 8GB RAM, Ubuntu 16.04 64-bit

MacBook (MacOne): Intel Core i5 1,4 GHz, 8 GB RAM, OS X El Capitan 10.11.5

MacBook (MacTwo): Intel Core i5 1,8 GHz, 4 GB RAM, OS X El Capitan 10.11.5

Julia 0.4.5

Python 2.7.11+

C++11

Java 1.8.0_92

Benchmarking will be done by comparing the run time of the same algorithm in different languages. In this project the run time is measured as the sum of **system** and **user** time, because the **system** and **user** time are not affected by heavy loads on the machine. All time measurement are measured using the shell **'time'** command that returns the three types of time taken for the algorithm to run. The problems will be scaled to have smaller and larger inputs. However, in some situations a large input can cause memory problems, which will be addressed in the specific problem. The memory problems is unfortunately making an upper bound for how large data is possible to pass for certain problems. A tool was created for this project that gathers the data and calculates the median of the run time.

7 Problems

In this section we explain some of the problems we have been using for benchmarking.

7.1 Projecteuler 11

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

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

```

The program runs from command-line/terminal with command line arguments in order to test on different inputs. To run the Julia version from the terminal:

Julia euler11.jl 100 4.

This would make the program run with a matrix of size 100 and calculating product of 4 adjacent numbers. A matrix generator was created to produce the matrix data, which can be found in appendix. The matrix generator requires one argument, which is the size of the matrix. So if 500 is passed to the matrix generator, a file named **500mat.txt** will be created containing 500×500 digits ranging from 100-9999.

7.1.1 Solution

The algorithm first reads through the input file and creates 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 through a matrix, lines 1-2 at figure 12. The outer loop iterates through the rows, and the inner loop iterates through the columns. Figure 12 calculates the horizontal and vertical directions, figure 13 diagonally from left to right(downwards) and figure 14 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 12, **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 12:

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 = Int64(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 = Int64(1)
13         for k = 0 : numProd - 1
14             prod *= mat[j + k, i]
15         end
16         if prod > maxProd
17             maxProd = prod

```

Figure 12: Horizontal and vertical

Figure 13 shows the loop that iterates over the matrix and calculating the product of the diagonal going downwards from left to right. Figure 14 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.

```
1  #diagonal left->right
2  for i = 1 : matLength - numProd
3      for j = 1 : matLength - numProd
4          prod = Int64(1)
5          for k = 0 : numProd - 1
6              prod *= mat[i + k, j + k]
7          end
8          if prod > maxProd
9              maxProd = prod

```

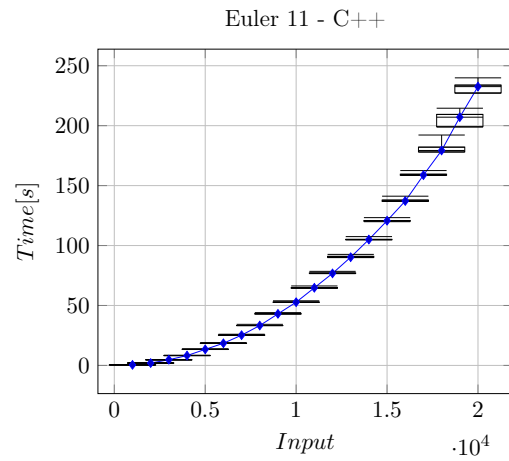
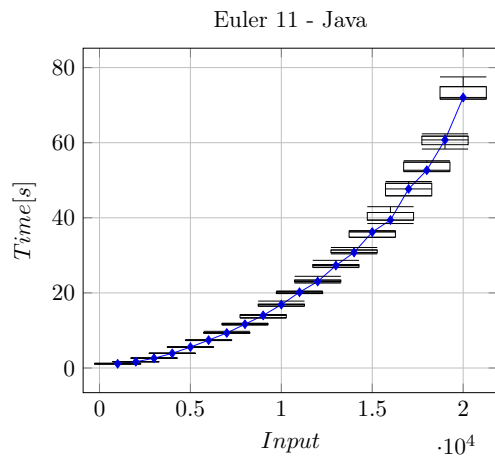
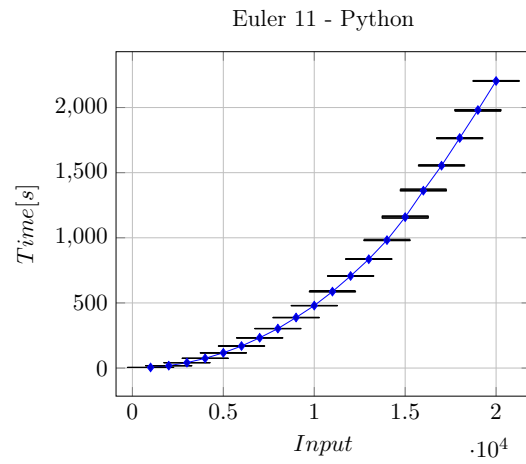
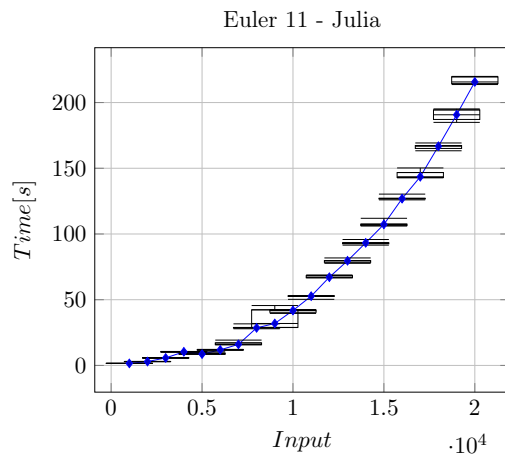
Figure 13: Diagonal downwards left to right

```
1  #diagonal right->left
2  for i = 1 : matLength - numProd
3      for j = matLength : -1 : numProd
4          prod = Int64(1)
5          for k = 0 : numProd - 1
6              prod *= mat[i + k, j - k]
7          end
8          if prod > maxProd
9              maxProd = prod

```

Figure 14: Diagonal upwards left to right

7.1.2 Results



Euler 11 - Python excluded

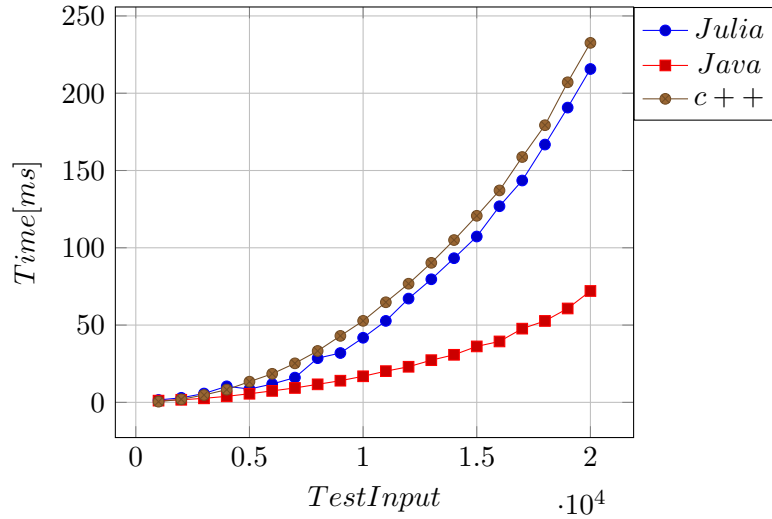


Figure 17: Graph

Euler 11

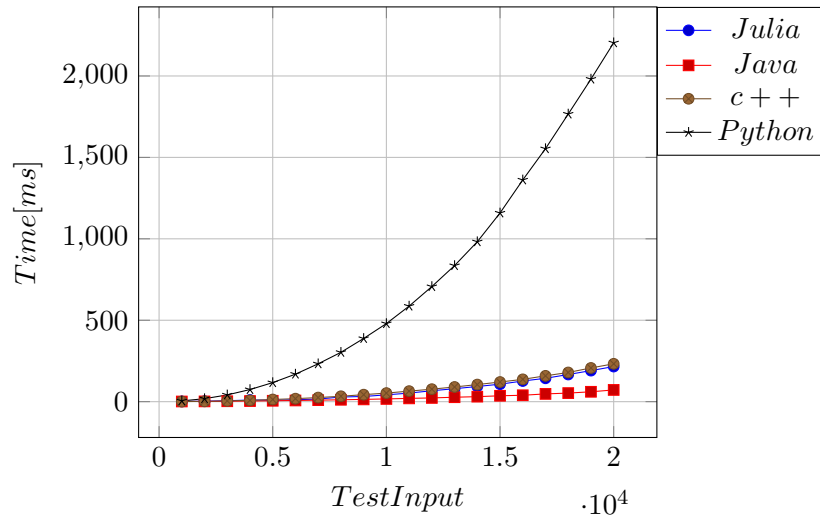


Figure 18: Graph

7.2 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 15 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 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?

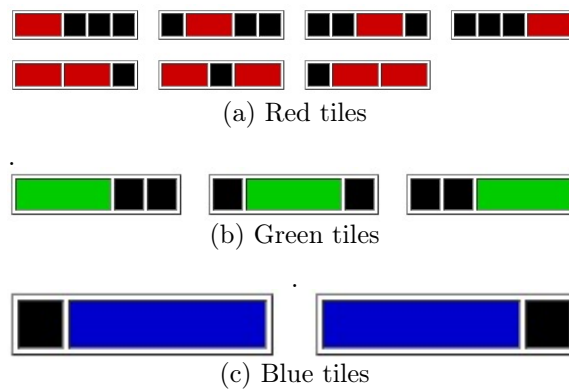


Figure 19: Projecteuler: 116

7.2.1 Solution

The solution of this problem is done recursively, although an optimal solution would be done iterative. The work being done in every recursive call is very simple opposed to quicksort. When the work in every recursive call is simple and similar, some sort of compiler optimization will most likely happen if supported by the language. This is a good way to test Julia's tail recursion optimization against Java's, and comparing with Python and C++.

The above code is the solution written in Julia

```

1 function solve(m::Int, n::Int) #m=color block size n = black box
    size
2   if m > n
3     return 0
4   end
5   solutions = 0
6
7   for i = m : n
8     solutions += 1

```

```
9     solutions += solve(m, n-i)
10 end
11
12 return solutions
13 end
14
15 function s(size::Int)
16     result = 0;
17     for i = 0 : 2
18         result += solve(2+i, size)
19     end
20     return result
21 end
22
23 size = parse{Int, ARGS[1]}
24 s(size)
```

The input that is changing when doing the benchmark is the size of the black box that's get filled with the three different sized blocks. The input will start at size 45 and increase by one every test. This will be done 10 times for each language. The result is a the following:

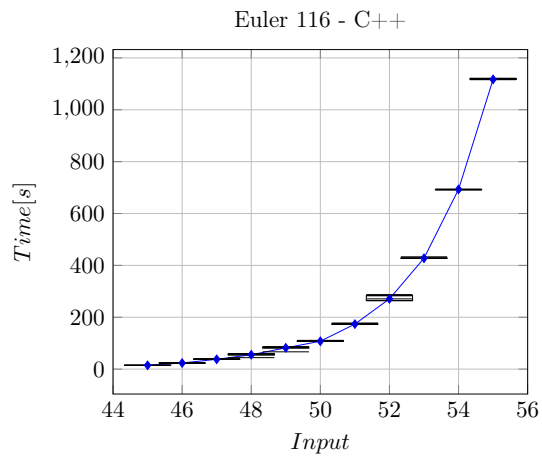
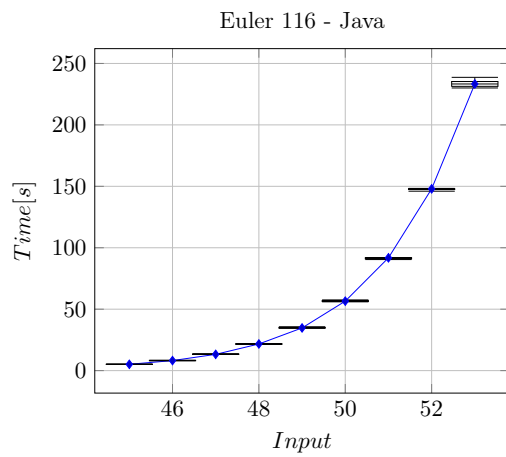
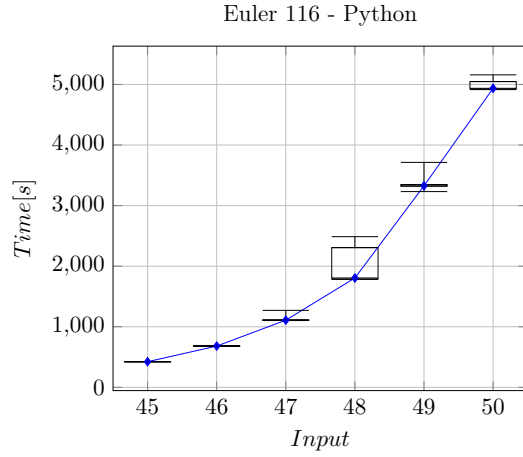
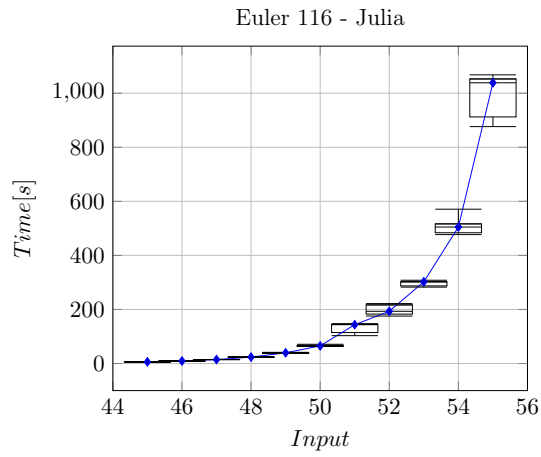
7.2.2 Results

It is clear that the graphs are exponential increasing. Another thing to notice is that the input is only increasing by one but is still making a huge difference in the run time. One of the reasons is that the problem is solved with recursion, for every extra one bit of space added to the black box a lot more recursive calls will have to be made.

The difference in time between the four languages are as expected. Python does not support any form of tail recursion optimization and the result is really slow. Java and Julia on the other hand does a good job at optimizing the recursive calls and is actually faster than C++ - keep in mind that no compiler flags were used in C++, so the default optimization level is used.

Julia and Java is close but Java is a bit faster, which is expected because of the fact that Java has been in development much longer than Julia and has

a lot more optimization than Julia.



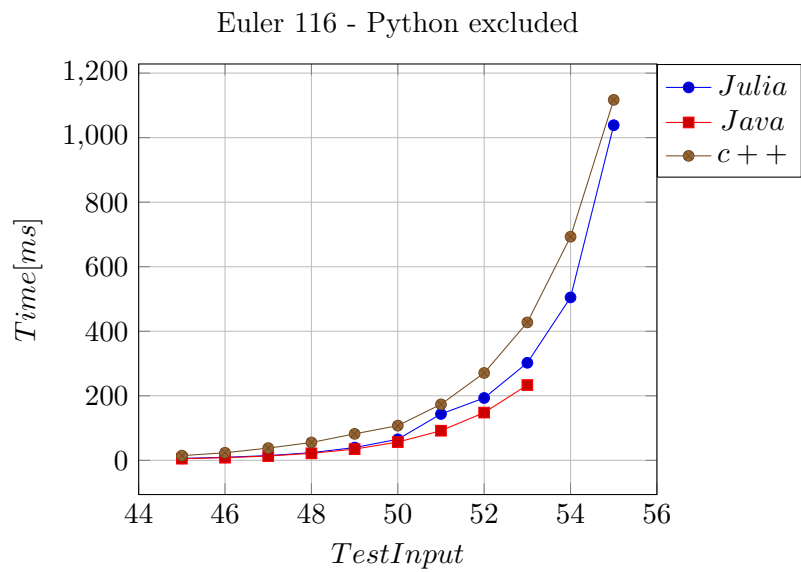


Figure 22: Graph

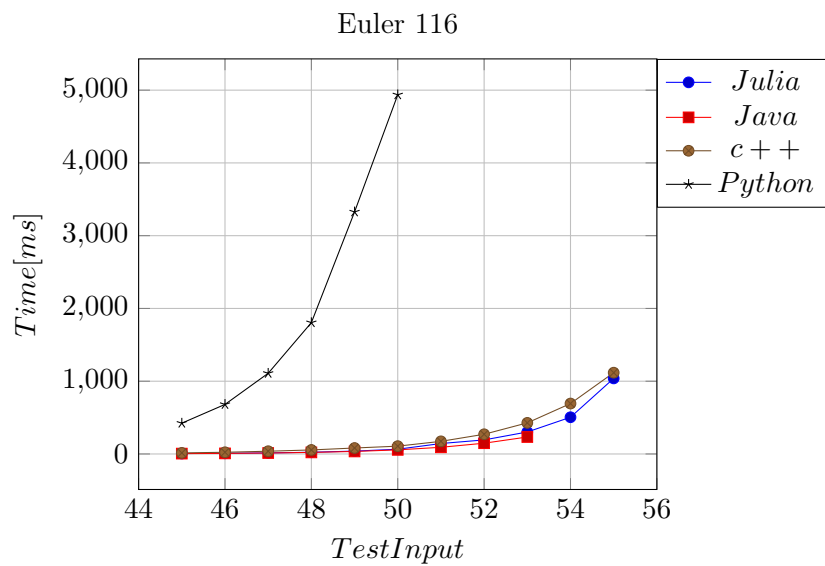


Figure 23: Graph

7.3 Dual Pivot Quicksort

Quicksort applies the divide-and-conquer paradigm to sort numbers. Dual pivot quicksort is very similar to the standard quicksort, the only difference is that instead of one pivot there are two pivots and three recursive calls. The first recursive call is the numbers that are less than the first pivot (which also

must be the smallest of the two pivots). The second recursive call is with the numbers that are greater than the first pivot but less than the second pivot. The last recursive call is with the numbers greater than the second pivot. Dual pivot quicksort is so fast that it was not possible to get large enough data to get the run time close to 15 minutes. With a data set of 20.000.000 numbers the loading time was a few minutes while the sorting was only a few seconds. Much larger data set will cause out of memory error so it was necessary to change the approach. instead of sorting one big array, the program will first sort the elements from 0-125, then sort 0-250 and keep increasing the end index until the array size has been reached. This approach insures that the loading time of the data is so low that it does not matter in the full picture, but this way of doing quicksort cannot test how well quicksort performs with different inputs and is the reason why only uniform random inputs are used.

The data set start at 100.000 numbers and increase by 30.000 every test. The solution is below:

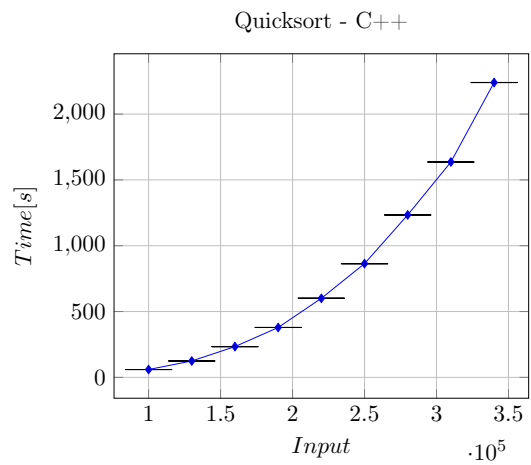
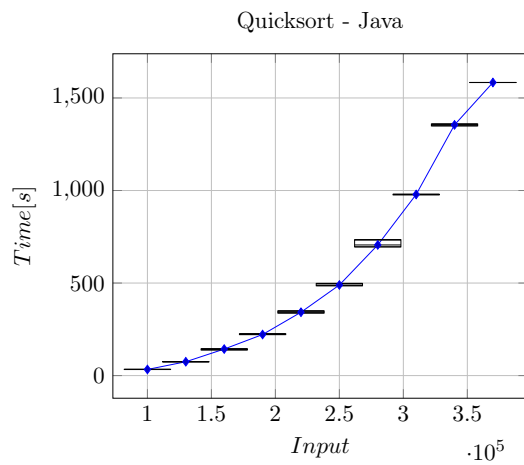
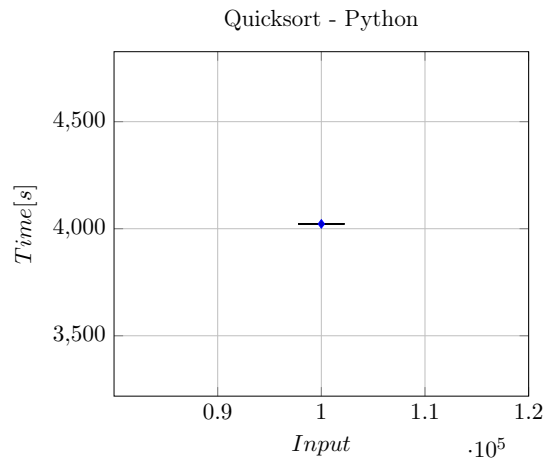
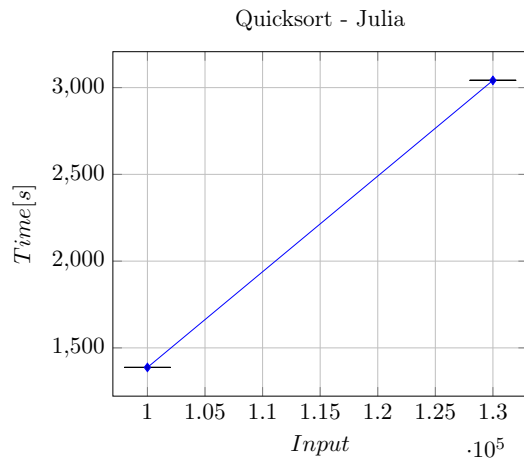
```
1 function quicksortDualPivot(a, s, e)
2     if e > s
3
4         if (a[s] > a[e])
5             swap(a, s, e)
6         end
7
8         leftPiv = a[s]
9         rightPiv = a[e]
10
11        leftPoi = s + 1
12        rightPoi = e - 1
13
14        curEle = leftPoi
15
16        while curEle <= rightPoi
17            if a[curEle] < leftPiv
18                swap(a, curEle, leftPoi)
19                leftPoi += 1
20            elseif a[curEle] >= rightPiv
21                while a[rightPoi] > rightPiv && curEle < rightPoi
```

```
22         rightPoi -= 1
23     end
24
25     swap(a, curEle, rightPoi)
26     rightPoi -= 1
27     if (a[curEle] < leftPiv)
28         swap(a, curEle, leftPoi)
29         leftPoi += 1
30     end
31 end
32     curEle += 1
33 end
34     leftPoi -= 1
35     rightPoi += 1
36
37     swap(a, s, leftPoi)
38     swap(a, e, rightPoi)
39
40     quicksortDualPivot(a, s, leftPoi - 1)
41     quicksortDualPivot(a, leftPoi + 1, rightPoi - 1)
42     quicksortDualPivot(a, rightPoi + 1, e)
43 end
44 end
45
46 function swap(a, i, j)
47     tmp = a[i]
48     a[i] = a[j]
49     a[j] = tmp
50 end
51
52 #Load input
53 f = open(ARGS[1])
54 s = readall(f)
55 ss = split(s)
56 data = []
57 for i = 1 : length(ss)
58     push!(data, parse{Int32, ss[i]})
59 end
60 close(f)
61
62
```

```
63 | jumpLength = 125
64 | i = 0
65 | shouldBreak = false;
66 |
67 | while true
68 |     i += jumpLength
69 |     if i > length(data)
70 |         i = length(data)
71 |         shouldBreak = true
72 |     end
73 |
74 |     quicksortDualPivot(data, 1, i)
75 |
76 |     if shouldBreak == true
77 |         break
78 |     end
79 | end
```

7.3.1 Results

The fastest of the four languages is definitely Java, which is kind of surprising as the expectation was that the compiler optimization would break with this kind of work in every recursive call - to check if Java was over optimizing a few elements from the sorted list was printed at the end of the program, to insure Java did not just terminate because it found that the array was not going to be used further after the sorting. Although Javas optimization is still present, the optimization of Julia's breaks and has a hard time to complete the first few inputs. C++ seems pretty slow compared to Java but keep in mind that this is without much optimization - C++ is known for the fast speed performance when working with arrays and this is why it does manage to get almost all inputs opposed to Python, which has no recursion optimization and is really slow when working with arrays and lists.



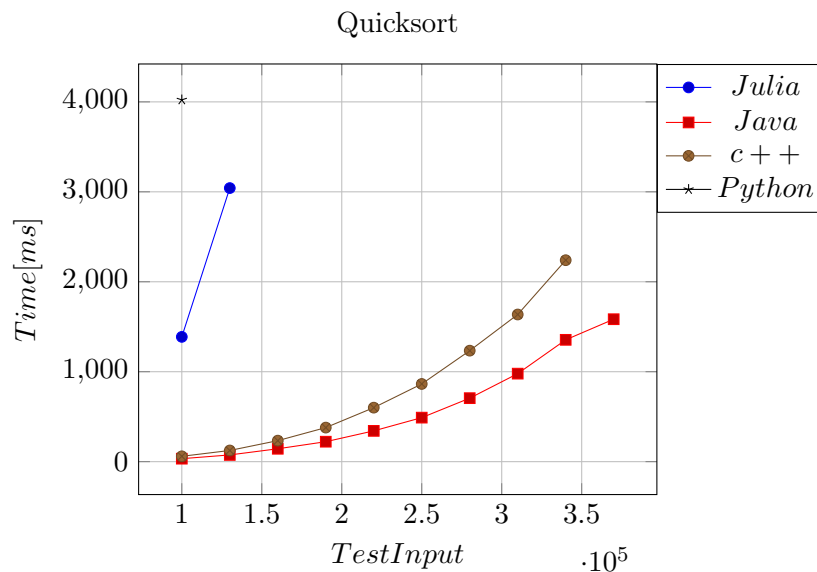


Figure 26: Graph

7.4 Statistics

Up until now there has been tests on euler11, euler116 and Quicksort. With only three types of testing, the results will not cover every aspect of comparing different usages in a languages, for example comparing arrays, for-loops, recursion etc.. Statistics was added to gain a bit broader vision on how Julia performs in different situations, since having three programs, in each language, does not cover everything. In order to make statistics, a set of problems was selected from Projecteuler. While having four languages to compare, a small set of problems will quickly turn in to a larger set.

**** Update after testing ****

Currently there is twenty problems from Projecteuler, and that becomes 80 separate programs. Testing the base case of Projecteuler, the problem as it is stated, mostly results in a run time of less than a second. No one cares if the run time of a program is 300ms, or 500ms, for this particular reason it was needed to scale the problems. A given problem was tested with ten different inputs. The first input is as stated in the original problem, and then increased nine times, giving ten results for each problem. There was set a maximum run time of five minutes for each test. Having 80 programs, the time needed for testing will get high very fast. All programs were tested three times with each input, and the median is used for the graphs shown

later. Starting with 80 programs, which is tested with ten different inputs and each input tested three times, resulting in 2400 programs. Keeping the amount of programs in mind, an average run time of one minute, would make the testing phase to a total of 40 hours for one computer. Therefore, the testing was divided to run on three computers. The specifications of the computers can be seen in the appendix. MacOne tested eulers: 1, 4, 8, 9, 12, 13, 18 and 22. Ubuntu tested eulers: 14, 16, 19, 20 and 21. MacTwo tested eulers: 30, 31, 36, 50, 52, 76 and 112. One big disadvantage, and time consumer, is that the inputs had only been properly tested on the Julia versions, before making the final tests. In some cases, the other languages, opposed to Julia, had trouble going through all inputs within the time limit. All programs were implemented in Julia as the first language, and copied into Python, Java and C++. Meaning the problems are not optimized in a certain language, possibly giving an advantage Julia over the other languages. If a test exceeds the time limit of five minutes, it will not be included in the graphs shown.

Some of the problems had to be altered in order to being able to handle large inputs. Problem 8, 13, 18 and 22 all reads their inputs from a .txt file, before doing any computations. The time for reading data from a file and setting up the data, is included in the run time of the tests.

8 Personal experience with Julia

In this section we each wrote about our personal experience with Julia.

8.1 Sonni

Some of the design choices, in Julia, can be really frustrating for someone who is adapting to Julia. The indices starts at 1 - this could be debatable because of the statement that Julia is used in a lot of scientific computations and many programming languages made for scientific computations use 1-based indexing. But as a programmer who is used to indices starting at 0, this can cause a lot of unnecessary bugs. The developers of Julia also made unusual design choice with the for-loop syntax. A incrementing for loop looks like:

1	Julia:
---	--------

```

2  for i = 1 : n
3      Some code ...
4  end
5
6  General languages:
7  for i = 1; i < n; i++
8      Some code ...

```

This is straightforward many other languages, but take a look at the decrementing for-loop:

```

1  Julia:
2  for i = n : -1 : 0
3      Some code ...
4  end
5
6  General languages:
7  for i = n; i > 0; i--
8      Some code

```

In Julia the exit condition and incrementation / decrementation are swapped and there is no consistency. The decrementing for-loops both start at n , and stops at 0. This kind of syntax can again cause bugs, and is not that easy to get eyes on.

The documentation of Julia is far from being great, which is a huge bump on the road to learn Julia. It feels far from complete. The documentation do not explain much but instead gives some examples. Essential information are often lacking, e.g. what types is needed to parse for a given function. It might point out that these arguments are needed, but lacking to describe those arguments. An example from the documentation is the function **rand()**, which is used to generate random numbers:

```

1  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`.”

The argument **S** is explained, but does not mention anything about **rng** or **dims**.

Another example is how to access data within an array at a specific index. This should be one of the first things in the Array section, but is first described after about two pages of text. A person with experience in programming might know that to access an element in an array with an index is this simple line of code:

```
1 a[index]
```

But this isn't so obvious for new programmers. It is clear that something has to be done with the documentation.

Another downside to Julia is its consistency. The developers has tried to remove some of the burden from the programmer with dynamic typing, package manager system etc., but with memory allocation and garbage collection it does not make much sense. The garbage collector does not free memory from unreachable object. Sizes of variables cannot be changed after declaration, and is not automatically increased if needed. If the programmer assign for example 8 bit memory to a variable and later change the value of that variable, then julia automatically assigns 32 bit or more if needed to that variable even if the 8 bits were enough.

8.2 Simon

I personally see my self the one in the group with the least programming experience, and might needed more time learning Julia. I was not learning as fast as the others. I only know a few languages, and as many knows, the more languages you know, the easier it gets learning a new language. The documentation of Julia was not satisfying enough to me. I learn faster if I can visually see how a function is used, rather than reading a wall of text. The visualizations on the current documentation was not fulfilling my needs. The documentation also misses to explain crucial information to a

function, like what a given argument is presenting. For that reason I often found myself looking through the documentation of Python, to read about a function, since the syntax is so similar. Nevertheless, for programming I keep finding Julia more satisfying as I learn more, and for future programming I will definitely use Julia over Python.

8.3 Asbjørn

I found the syntax a bit confusing at first. I am not used to Python and I missed code structure and forced typing, in Julia and Python it looks like the code is hovering. After some time I got used to it and I found it very easy to learn. It makes it a little hard to find answers with the documentation and community being a bit thinner than with other languages. I think the 1-indexing is a bit complicated because I had never thought any language would use this, but I found that a lot of languages actually use this, they just are not that popular.

9 Discussion

Pros and Cons with good arguments <http://www.evanmiller.org/why-im-betting-on-julia.html>

High-Performance JIT Compiler chart at <http://julialang.org/>

<http://www.nowozin.net/sebastian/blog/the-julia-language-for-scientific-computing.html>

<http://radar.oreilly.com/2013/10/julias-role-in-data-science.html>

10 Conclusion

Is Julia a worthy competitor? Is it worth switching to Julia? Would we recommend Julia to anyone?

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 and by the benchmarking it could look like that Julia is already performing well. The idea of the developers is clear, they are trying to make

a programming language that is as easy to learn and write as Python but with a compiler that optimizes code like Java and both of these goals have more or less been reached by the developers. Julia was differently easy to learn and by the benchmarking it does look like that the compiler is doing okay with the optimization when it is possible. It does also feel like that the Julia developers are playing with the idea of giving the user some control over what is happening behind the scene. 'Playing' because this does not feel even close to done. Examples are that they are giving the user an option to allocate memory for variables manually but as soon as a new value is assigned to that variable then Julia will change the memory size. Also it is not possible to free a declared variable's memory entirely and the only operation the user can do with the garbage collector is force garbage collection and disable the garbage collector. Somewhat the same story goes for the option to implement code from other languages this seems kind of buggy and not complete.

11 Appendix

11.1 Process analysis

This section is about our management of time and resources.

11.1.1 Projectmanagement, organizing workflow and organizing process of learning

The first thing we agreed doing was getting to know Julia for the research. We used a lot of time solving problems in Julia and trying to figure out how to get a view of what was to be done and how to organize it. We mostly communicated through facebook and created a repository on github to share project files. We gathered our solutions to Project Euler and made comparisons. Gathering information about benchmarking we mostly ran into walls of text on how to do it properly, and did not know where to start. Every time we thought we were understanding benchmarking, our supervisor taught us something new. We created a board on Trello. We made a to-do list of cards so we knew what had to be done and started handing out tasks. We communicated more with our supervisor via email as we started writing the report. The group was really good at getting the tasks, we agreed on doing, done. We should have been using more tools for organizing and we did not set any deadlines for anything. Scrum is a tool, which would have helped us a lot, but it is not that easy to learn. We should have met and worked together, several times a week, discussing our problems and goals.

11.1.2 Process of learning

We learned by both direct and indirect learning process. We learned Julia indirectly by solving Project Euler. We pretty much read about every topic, which is direct learning. We cannot think of another way to learn programming than reading, trying it out and reading to get a result. Project Euler was a good support for learning, since learning something new, is much easier, when there's a purpose or a goal. Programming is not as fun if there's no end goal in mind.

11.1.3 Cooperation, difficulties in group and with supervisor

We actually had a good idea about who was good at what and splitting the tasks between us with this in mind. The communication could have been better. It seemed that we worked on different times of the day, which made communication difficult. Our supervisor did his best and we think he did a good job.

11.1.4 Conclusion

Trello is a very good tool for organizing and we should have been using it from the beginning. Looking beyond the fact that some data was lost using Git, the tool is perfect for team programming. We have payed the price not making any deadlines. Deadlines seems useless when the focus is just on the project, but when the workload gets more intense, it can be very good to have smaller deadlines or milestones during a project. Meetings should have occured more often, and syncing the current progress. Overall collaboration was very well, taken into account what could have been improved.

11.2 Copy of poster in A4

References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein *Introduction to Algorithms*, Cambridge, Massachusetts, third edition, 2009.
- [2] http://www.tiobe.com/tiobe_index
- [3] <http://julialang.org/blog/2012/02/why-we-created-julia>
- [4] <http://docs.julialang.org/en/latest/manual/parallel-computing/>
- [5] <http://unicode.org/standard/WhatIsUnicode.html>
- [6] <http://www.cs.cornell.edu/courses/cs1112/2013fa/Exams/exam2/vectorizedCode.pdf>