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

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.

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

Figure 2: Diagonal downwards left to right

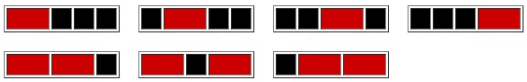
```
1  #diagonal right->left
2  for i = 1 : matLength - numProd
3    for j = matLength : -1 : numProd
4      prod = 1
5      for k = 0 : numProd - 1
6        prod *= mat[i + k, j - k]
7      end
8      if prod > maxProd
9        maxProd = prod
```

Figure 3: Diagonal upwards left to right

3 Projecteuler 116

A row of five black square tiles is to have a number of its tiles replaced with coloured oblong tiles chosen from red (length two), green (length three), or blue (length four).

If red tiles are chosen there are exactly seven ways this can be done.



If green tiles are chosen there are three ways.

And if blue tiles are chosen there are two ways.

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?

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.

8 Appendix (source code)

References

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