Project Report

Kratika Singhal - 69535971
Shrishail Zalake - 07466343

Problem definition:
An interesting problem in arithmetic with deep implications to *elliptic curve theory* is the problem of finding *perfect squares that are sums of consecutive squares*. A classic example is the Pythagorean identity:

$$3^2 + 4^2 = 5^2 \quad -- (1)$$

that reveals that the sum of squares of *3, 4* is itself a square. A more interesting example is *Lucas' Square Pyramid*:

$$1^2 + 2^2 + \ldots + 24^2 = 702 \quad --(2)$$

In both of these examples, sums of squares of consecutive integers form the square of another integer.
The goal of this project is to use FSharp and the actor model to build a good solution to the above state problem that runs well on multi-core machines.

Requirements:
Input: The input provided (as command line to your program, e.g. my app) will be two numbers: *N* and *K*. The overall goal of your program is to find all *K* consecutive numbers starting at *1* and up to *N*, such that the sum of squares is itself a perfect square (square of an integer).
Output: Print, on independent lines, the first number in the sequence for each solution.
Actor modeling: This project uses the AKKA framework to implement actor model in FSharp.

Execute the Program:
We have used the following command to run the program from command line interface and pass inputs.
dotnet fsi –langversion:preview program.fsx 1000000 4

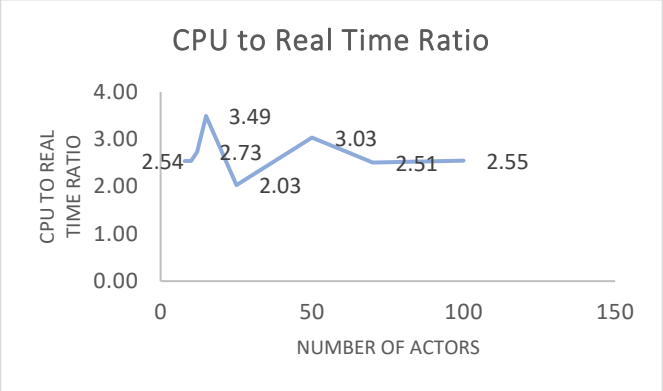
Work Unit:
The work unit for every actor in our design depends directly on the value of *N*. Every actor gets an interval of numbers for which they need to check for potential occurrence of Lucas Pyramid. The size of the interval is determined by formula

$$M = \frac{N}{(total\ actors)}$$

This divides our problem into *M* sub problems, each solved by one specific actor.
We compared the CPU and Real-time ratio for different number of actors "*actorNum*" and got best result for number of actors = 15. We were able to achieve best ratio between 5-6 for higher order inputs(10^8). For the input *N* = 1000000 and *K* = 4 we were able to achieve ratio of 3.49 which shows all 8 of cores are being utilized in parallel.

Table below shows the comparisons we did through hit and trial for different number of actors.

Input	Number of Actors	CPU Time (in ms)	Real Time (ms)	CPU to Real Time Ratio
N = 1000000, K = 4	8	609	240	2.54
	10	671	264	2.54
	12	671	246	2.73
	15	984	282	3.49
	25	546	269	2.03
	50	828	273	3.03
	70	890	355	2.51
	100	828	325	2.55



Important Observation: The CPU to real time ratio increases as the input size increases.
The below shown timings are for inputs 100000000 24. **The ratio is 5.4**

```
19823373
82457176
Real: 00:00:23.665, CPU: 00:02:09.890, GC gen0: 24509, gen1: 4, gen2: 0
```

Result: result of running your program for **dotnet fsi --langversion:preview proj1.fsx 1000000 4** is none.

```
C:\Users\skrat\TestProject>dotnet fsi --langversion:preview proj1.fsx 1000000 4
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
Real: 00:00:00.275, CPU: 00:00:00.937, GC gen0: 92, gen1: 2, gen2: 0
```

Running time: running time for N=1000000 K=4, as shown above the timing are:
Real time: 275ms
CPU time: 937ms
CPU to real time ratio: 3.40

Largest problem solved:
We were able to solve the largest problem of size 10^8.
Input: N = 100000000 and K = 50
CPU Time: 00:05:01.343
Real-Time: 00:00:44.391
CPU/Real: 6.7

Issues Identified:
While testing the code we observed that the inbuilt square root function (sqrt) in FSharp was giving making some approximations and hence the result was not accurate.

For instance, for N = 100000000 and K = 2 we are getting the number 65918161 in the output.
If we validate:
65918161 * 65918161 = 4345203949621921
65918162 * 65918162 = 4345204081458244
4345203949621921 + 4345204081458244 = 8690408031080165 which is not a perfect square.
However, the inbuilt sqrt function gives the square root as 932223580.0
And if we validate further 932223580.0*932223580.0 != 8690408031080165
To resolve this issue, we have implemented the following code block which makes the result accurate.

```
    sqSum <- sqSum +  (uint64 j*uint64 j)
sqRt <- sqSum |>double |>sqrt |>uint64
if sqSum = sqRt * sqRt then
    printer<!i
```