Technical University of Munich
School of Engineering and Design
Prof. Dr. Martin Werner

Computational Foundations I (Winter Term 2021/22)
Tutorial 2

Tasks marked with a star like **Optional Task*** are optional. Tasks marked like **Hard Task**+ are given, but it is not expected that you solve them now. It is great if you learn to solve them during the lecture. Go back to them after a few weeks and see your own progress.

Learning Outcome: We learn basic control structures and apply it to find prime numbers, you get an example of how small changes to algorithms can have huge impact on the performance (faster prime test), you train control structures by implementing written addition in the ten-digit system in MATLAB and you get introduced to time-discrete simulation

Task 3: Control Structures and Loops

In this task, you practice how to use control structures such as if commands and loops statements in order to control conditional or repeating elements of programs.

a. **Rough Square Root:** In this task, we will compute the square root of an integer number as follows (this is not the smartest way, but we can learn something): We set a variable to 1 and start a loop in which we double this number each iteration. In this way, the number `i` will quickly grow. Each time in the loop, we also compute the square of the number by multiplying the number with itself. As soon as the square is larger than the given input parameter, we start a second loop in which we go "backwards" decrementing (which is making a variable one less) the loop variable one step at a time until we (again) fall below the input parameter. As soon as the square of this variable becomes smaller, we know that the square root is inbetween this number and its predecessor, which is exactly one higher.

   Implement this strategy in MATLAB.

b. **Testing a Number for Being Prime:** A number is a prime number if it is not divisible by any other number than the number itself and one. As a number that divides another must at least be smaller, a simple algorithm just checks if all numbers smaller than the given number are dividers. Implement this using a for loop and test for divisibility by checking that the remainder (modulo function) is zero. Terminate the algorithm (`break` statement) as soon as you find a dividing number as you can then directly conclude the value is not prime.

```
1;
fprintf("Test: 3 divides 4 ==> %d (should be 0)\n",divides(3,4));
fprintf("Test: 1 divides 5 ==> %d (should be 0)\n",divides(1,5));
fprintf("Test: 2 divides 8 ==> %d (should be 1)\n",divides(2,8));
fprintf("======================================\n");
fprintf("Test: isPrime(9)  ==> %d (should be 0, 9=3*3)\n",isPrime(9));
fprintf("Test: isPrime(13) ==> %d (should be 1, 13 is prime)\n",isPrime
    (13));

function ret=isPrime(a)
```

```
% write this function
   ret = 1;
end

function ret = divides(a,b)
   ret = (mod(b,a) == 0); % modulo is the remainder of integer division
end
```

c. **Faster Prime Testing:** A simple observation is that at any time in the previous algorithm for a number to test p, we already found out that no smaller number than the loop counter is dividing. Hence, as soon as we exceed

$$c = \sqrt{p}$$

we can stop the process. Because if there was a dividing number larger than this threshold c, then it would have to be paired with a number smaller than c not to exceed p, but they all have been tested. This is a powerful speedup, as we reduce the number of loop body executions from $p - 1$ to the smallest integer larger than $\sqrt{p}$. Now look at the numbers:

p	Steps of First Algorithm	Steps of Second Algorithm
13	12	4
97	96	10
19,134,702,400,093,278, 081,449,423,917	19,134,702,400,093,278, 081,449,423,916	138,328,241,513,052

To feel the real effect of such speedups and, therefore, the power of a topic called time complexity, run this for the tenth Fibonacci prime (see `https://oeis.org/A005478`). This takes about 10 seconds on my computer for the slow version and 0.0016 seconds for the fast version. Note that both algorithms are doing exactly the same amount of work for non-prime numbers, namely finding the first dividing number smaller than $\sqrt{p}$. That is, the second algorithm is faster only for prime numbers.

Implement this optimized version.

Task 4: Problem Solving - Written Addition from School

A big part of programming is related to problem solving and algorithmic thinking: how do I decompose a problem into simple steps that can be executed. Using the Modulo (remainder of an integer divison, see `https://www.mathworks.com/help/matlab/ref/mod.html` function implement the algortihm of written addition from school.

a. Given two row vectors `x =[1,2,3,4]` and `y = [5,6,7,8]`, write a program that adds them as numbers 1234 + 5678, which should be 6912.

b. If you haven't done already, write the addition program as a function taking two arbitrary column vectors representing the numbers

Task 5: Time Discrete Simulation of Movement According to Basic Physics*

Simulations are one of the many kinds of programs we can apply in engineering. Simple physics simulations run as so-called **time-discrete simulations** in a loop that advances a shared variable t holding the time by a a discrete timestep Δt in every iteration.
That is, the basic pattern of time-discrete simulation is as follows:

```
    t_start = 1  % some sort of start time
    t_end = 100  % when to end
    for time = t_start:t_end
    % proceed for a timestep with simulation logic
    % proceed for a timestep with physics
    end
```

We want to practice this principle.

a. **Throwing a Ball:** When throwing a ball, you use your body to assign a certain velocity (e.g., speed acting into a known direction) to it. How does this given speed combined with gravity complete to a time-discrete simulation returing the distance of your pitch? Complete the following simulation.

```
p = [0,1] % initial position
angle = 45 % start angle
speed = 10 % inital speed in m/s
v = speed*[cos(deg2rad(angle)),sin(deg2rad(angle))]  % velocity
g = [0,-9.81] % gravity
dt = 0.2 % delta time
dg = g*dt % delta gravity
L = [p] % Lines

for i = 1:100
                    % <-- update speed
                    % <-- update location
                    % <-- break when hitting floor


                    % <-- add  point to trajectory
    disp(p(2));
end

disp(sprintf("we went for %f m in %i iterations", L(end,1), i))

% visualize
plot(L(:,1),L(:,2))
```

b. **Optimal Angle:** Adjust the program in such a way, that you find the best integer angle, that achieves the largest distance and have the program display the best angle as well as plot the best trajectory. Tipp: As we constrained to integer angles, you can use a keep-the-best pattern similar to the following:

```
best_angle = -1; % We dont know our best angle
best_length = -inf; % We dont know the best length
for this_angle = 1:360
    this_length = simulate_throw(this_angle)
    if this_length > best_length
        fprintf("Updating best_length from %.2f to %.2f at angle %d\n",
           best_length, this_length, this_angle)
        best_length = this_length
        best_angle = this_angle
    end
end
fprintf("The best result of %.2f m was achieved with an angle of %d\n",
    best_length, best_angle)
```

In this pattern, `-inf` denotes minus infinity and every finite number is larger. We then run the loop and if there is an improvement, we update the variables `best_angle` and `best_length`.