Lab 1: Introduction to RUST

Learning Objectives:

- Introduction to Rust programming
- Getting up and running with Rust
- Running your first Rust program
- Add dependencies to your project
- Experiment with arithmetic libraries in Rust.

What is Rust?

- Rust is a multi-paradigm system programming, focused on safety, speed, and concurrency.
- Rust supports a mixture of imperative procedural, concurrent actor, object-oriented, and pure functional styles.
- It also supports generic programming and metaprogramming, in both static and dynamic styles.
- The syntax is similar to C++.
- The compiler is free and open-source software.

Getting Started with Rust

- You can try Rust online in the Rust Playground.
- To install rust, visit the rust language official site: https://www.rust-lang.org/tools/install
- Alternatively, you can install Rust using the Rustup tool, which is a Rust installer and version management tool.
    - On Linux and macOS systems, this is done as:

```
$ curl https://sh.rustup.rs -sSf | sh
```

    - On windows, install rustup-init.exe.
- When you install *Rustup*, you will also get the latest stable version of the Rust build tool and package manager, also known as ***Cargo***.
- **Editors:** Rust support is available in many editors such as Visual Studio Code (NOT Visual Studio), Sublime, IntelliJ IDEA, and Eclipse.

What is Cargo?

- Cargo is a tool that allows Rust packages to declare their various dependencies and ensure that you'll always get a repeatable build.
- To accomplish this goal, Cargo does four things:
    - Introduces two metadata files with various bits of package information.
    - Fetches and builds your package's dependencies.
    - Invokes *rustc* or another build tool with the correct parameters to build your package.
    - Introduces conventions to make working with Rust packages easier.

Deliverable 1: Writing your First Package with Cargo

- To create a new package, you can use *cargo new*:

```
$ cargo new hello_world
```

- Cargo has generated a new directory hello_world with the following files:

```
hello_world
├── Cargo.toml
└── src
    └── main.rs
1 directory, 2 files
```

  - *Cargo.toml:* is the manifest file which keeps metadata for the project along with the dependencies.
  - *src/main.rs* is where you write your code.

- Here's what you'll find in *src/main.rs*:

```
fn main() {
    println!("Hello, world!");
}
```

- To compile and run your project, we'll use Cargo:

```
$ cargo run
```

- Here are some of the main Cargo commands you will be using in this course to:
  - compile/build and run a project with `cargo run`
  - test a project with `cargo test`
  - benchmark your code with `cargo bench` (need Criterion crate)
  - free up significant storage space by running `cargo clean` (This command deletes the target folder and uninstalls the crates for that project. This command should be run before any code submission.)
- Here are also some other useful cargo commands:
  - build/compile (without running) a project with `cargo build`
  - create documentation with `cargo doc`
  - publish a project as a library (crate) with `cargo publish` (crate)
- Other cargo commands can be found here: https://doc.rust-lang.org/cargo/commands/index.html

- **DEMO this deliverable to the lab instructor.**

Deliverable 2: Adding dependencies

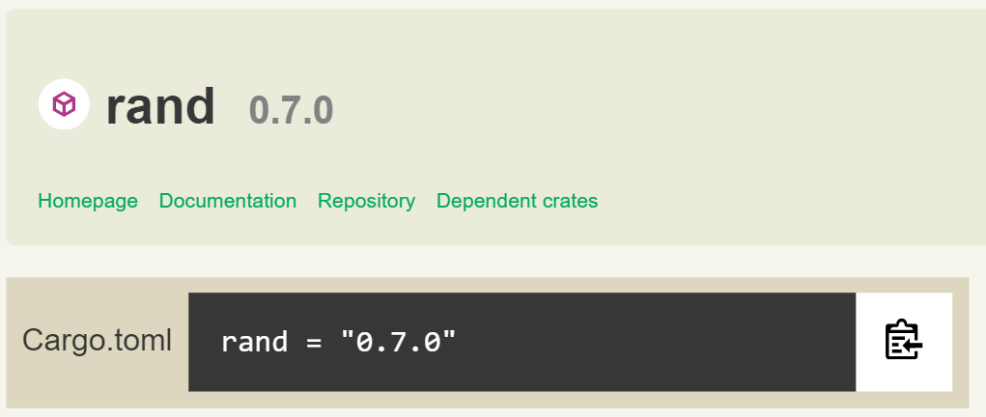
- Now, let us add a dependency to our project.

Rust stores all sorts of libraries on crates.io. *Crates* are what you use to refer to packages.

- In this part of the lab, we will use a crate called Rand, which is a Rust library for random number generation (**Hint**: this library may come in handy with any randomness-related algorithms)
- To add the crate to our project, we need to modify the *Cargo.toml* file as follows:

```
[dependencies]
rand = "0.7.0"
```

- You can find this information on from the crate page



- Now we run to let Cargo install the dependency:

```
cargo build
```

- Note that running this command creates a new file, *Cargo.lock*, which is the log for the versions of the dependencies we are using.
- To actually use Rand in your code, you need to open main.rs and modify it as follows:

```
use rand::prelude::*;
fn main() {
    let mut rng = thread_rng();
    if rng.gen() {
        println!("True was randomly generated.");
    } else {
        println!("False was randomly generated.");
    }
    let x: f64 = rng.gen();
    let y = rng.gen_range(-10.0, 10.0);
```

```
    println!("x is: {}", x);
    println!("y is: {}", y);
    println!("Random Number between 0 and 9: {}", rng.gen_range(0, 10));
}
```

- The first line imports everything from the prelude which is the lazy way to use rand as it only imports the most common items.
  The Rand library automatically initializes a thread-local generator which can be accessed via the *thread_rng* and random functions. For more information, see this link.
- **DEMO this deliverable to the lab instructor.**

Deliverable 3: The Mathematics of Cryptology

Nearly all modern systems use cryptography as an important tool. At the heart of cryptography are prime numbers, a prerequisite in a public-key system is the efficient generation of public-key parameters.

For instance, cryptosystems require *prime numbers* p and q for an admissible RSA modulus ***n = pq***.

Many cryptosystems depend on a trapdoor one-way function, which is a mathematical function that is easy to compute in one way but requires secret information to compute efficiently its inverse. We can construct trapdoors one-way functions from hard number-theoretic problems like the integer factorization. The formal definition of factoring is:

Given a composite integer N, the factoring problem is to find integers p, q such that pq=N.

So far, the largest number factored is 232 digits long (768-bit) by using hundreds of computers and two years. Using a home computer, it would have taken 2000 years. Finding the prime factors for numbers the size of what we use for cryptography today (2048 bits) is approximately four billion times harder than 768-bits.

Before starting to generate prime numbers, we need to select an arbitrary-precision arithmetic library. This is required; because the numbers that we are going to manipulate might be pretty big (a 1024-bit number is an integer of 309 digits). We need to use a special data structure to support big integers. Usually, the Integer class manages 32 bits, meaning it can hold values up to 2^{32} (4,294,967,295) which for many applications might be enough; however, to achieve security we need to be able to use big numbers. We do not just need to store the numbers, but to compute operations like multiplication and exponentiation between them. Rust has (many) common arbitrary-precision arithmetic libraries, e.g., apint, rug.

```
// Blue code is pseudo-code.
use rand::prelude::*; // Need this for rng.gen_range(…) function to work.
fn function(n: u32) -> Int  {
    let mut rng = rand::thread_rng();
    loop {
        let mut candidate::Int = Int::from(rng.gen_range(0, n));
        candidate.set_bit(0, true);
        let i = u64::from(&candidate);
        if is_prime(i)== true {
            return candidate;
        }
    }
}
```

Question 1. What is the above algorithm doing?

Question 2. Rewrite the above algorithm to use another multiple-precision arithmetic library (e.g., apint, rug). You can use a library (crate) for the is_prime function (i.e. you do not have to write the is_prime function from scratch). You can modify both the blue and black code, so long as the code remains functionally the same. Note: Rug crate may give an error when run for some students. It is recommended to use apint crate instead rather than trying to fix rug.

- To test whether a number is prime or not, there is the school method which is to iterate over all the smaller numbers and check if any of those numbers perfectly divide the prime candidate. We can immediately optimize our first attempt. First of all, we do not need to iterate through all the numbers, but just up to the square root, (the random number is forced to be odd). A better approach is to use a probabilistic algorithm, which can be used to determine whether a number is prime with a given degree of confidence. Probabilistic algorithms are algorithms that employ randomness to reduce the complexity or expected running time. The best known probabilistic algorithm used in practice is the Miller-Rabin test. It determines whether a candidate number is prime or not. The confidence of the algorithm depends on the number of iterations. Three iterations lead to a probability of failing to once in 2^{80}, which is considered a secure implementation.

```
// Blue code is pseudo-code.
fn miller_rabin(candidate: &Int) -> bool {
    // Rewrite finds the values s and d from this equation: candidate-1=d*2^s
    let (s, d) = rewrite(candidate);
    for _ in 0..5 {
        let basis = thread_rng().gen_int_range(&Int::from(2), candidate-2);
        let mut x = mod_exp(&basis, &d, candidate);
        // 1_usize is just the number 1.
        if x == 1_usize || x == (candidate - 1_usize) {
            continue;
        } else {
            for i in Int::zero()..s - 1_usize {
                x = mod_exp(&x, &Int::from(2), candidate);
                if x == 1_usize {
                    return false;
                } else if x == candidate - 1_usize {
                    break;
                }
                if i == s - 2_usize {
                    return false;
                {
            }
        }
    }
    true
```

```
}
```

Question 3. Explain the algorithm the above code segment is implementing.

Question 4. Several cryptographic libraries use the Miller-Rabin test to output probable primes. Albrecht et al. were able to construct composite numbers that some of these libraries declared to be prime. Hence, we need something better. Identify the crate which most closely implements the recommendations from Prime and Prejudice. (Hint: The crate mentions the paper "Prime and Prejudice".)

Question 5. Finally, prime fun. Prime number 41, can be written as the sum of six consecutive prime numbers: 41 = 2 + 3 + 5 + 7 + 11 + 13

Where the number of terms is X = 6, sum of the primes is Y = 41, and the list of prime numbers is [2, 3, 5, 7, 11, 13].

This is the longest list of consecutive primes that sum up to a prime below one-hundred.

Write a Rust program which calculates the longest list of consecutive primes that sum up to a prime below one-thousand. Print out X, Y, and the list of prime numbers. The goal is to maximize the value of X, NOT Y. Note: You do not need to start the sum from the value 2. The starting value can be any prime number (e.g.: 5 + 7 + 11 = 23).

- **DEMO this deliverable to the lab instructor.**