Lab 3: Benchmarking and Databases

Part 1: Benchmarking in Rust

Consider the following x86 machine code:

Details of x86 machine code can be found at:

http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

and also, at:

https://www.youtube.com/watch?v=75gBFiFtAb8

```
.LCPI0_0:
        .long   10
        .long   0
        .long   12
        .long   32
example::main:
        sub     rsp, 16
        movaps  xmm0, xmmword ptr [rip + .LCPI0_0]
        movups  xmmword ptr [rsp], xmm0
        xor     eax, eax
        cmp     dword ptr [rsp + 4], 32
        setg    al
        mov     dword ptr [rsp + 8*rax + 4], 10
        add     rsp, 16
        ret
```

Question 1: Examine the machine code and explain what it does? *(Hint: What can you do to a list with two for loops and a single conditional?)*

Question 2: Translate the machine code to Rust.

Question 3: Use the compiler explorer (https://godbolt.org/) to generate the machine code for your answer in Question 2.

Question 4: Does the optimization flag -O make a difference?

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

Criterion (https://github.com/bheisler/criterion.rs) is a statistics-driven micro benchmarking crate in Rust. It helps you write fast code by detecting and measuring performance improvements or regressions, even small ones, quickly and accurately. The crate helps us optimize with confidence, knowing how each change affects the performance of our code.

Question 5: Follow the instructions in the quickstart section (https://github.com/bheisler/criterion.rs#quickstart) to add the Criterion crate to your project (**Question 2**).

Question 6: Bench your function from Question 2 using the following code:

```
let mut rng = rand::thread_rng();
let mut l: Vec<i64> = (0..10000).map(|_| {rng.gen_range(1, 10000)}).collect();
c.bench_function("your function: ", |b| b.iter(|| your_function::<i64>(black_box(&mut l))));
```

Feel free to replace “your_code” with the name of the function you wrote in Question 2 and examine the output of the benchmark.

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

Part 2: Code Optimization

Based on the results obtained in Question 6, Do you see any room for optimization*? (Hint: consider using iterators (https://doc.rust-lang.org/std/iter/trait.Iterator.html)*

Question 7: Optimize your code from Question 2.

Question 8: Bench your function from Question 7 using the following code:

```
let mut rng = rand::thread_rng();
let mut l: Vec<i64> = (0..10000).map(|_| {rng.gen_range(1, 10000)}).collect();
c.bench_function("your function: ", |b| b.iter(|| your_function::<i64>(black_box(&mut l))));
```

Question 9: Compare between the two performance results obtained from Criterion crate from Question 6 and Question 8.

Question 10: What are “zero-cost abstractions” in Rust? Explain how this term applies to Question 9.

- **DEMO (and Explain) this deliverable to the lab instructor.**

Part 3: Managing Databases

Modern applications are rapidly evolving to become data-heavy systems. For example, Training machines with data is starting to become a mainstream way to solve complex problems. As a result, the data should be easy to retrieve and must have guarantees regarding consistency and durability. Databases are the go-to solution for providing a robust foundation for building applications that are data-backed and that support the expectations of their users. A database is a collection of tables. Tables are units of data organization. Data that's been organized into tables is only applicable to relational databases. Other databases, such as NoSQL and graph-based databases, use a more flexible document model to store and organize data. Tables are usually representations of entities from the real world. These entities can have various attributes that take the place of columns in these tables.

In this part, we will use SQLite to design a convenient database system to provide data storage for a simple Bank application written in Rust.

1- SQLite Integration

First, we need need to install SQLite in order to be able to create our database.

- For windows:
  1- Install SQLite for their website (https://www.sqlite.org/download.html).
  2- Download the sqlite-shell-win32-*.zip and sqlite-dll-win32-*.zip zipped files.
  3- Create a folder on your machine (e.g., C:\sqlite) and unzip these two zipped files in this folder.
  4- Add the path for this folder (e.g., C:\sqlite) to your PATH envrionment variable.
  5- To test the installation, go to the terminal and write sqlite3 comand, the following message should be displayed.

```
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
```

- For Linux:

  SQLite should be installed by default on almost all the flavours of Linux OS. You can test the installation by issuing the *sqlite3* command in the terminal.

- For Mac OS:
  1- Although the lates version of Mac should have SQLite installted by default, you can insall it from their website (https://www.sqlite.org/download.html).
  2- Download sqlite-autoconf-*.tar.gz from source code section.
  3- Run the following command −

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

  4- To test the intallation, run the following command on your machnie:

```
$sqlite3
```

Second, once you have SQLite installed, you can create a new Rust project, then create a new folder called data. Your project's folders should have the following folders/files:

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

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

Navigate to the data folder and run the following command to create your database

```
sqlite3 users.db
```

Your database will have two tables:

1- One for username and password

2- Another one to store a list of transactions.

- To create the user table that stores usernames and passwords for bank clients, we can run the following command:

```
sqlite> create table users(u_name text PRIMARY KEY, p_word text);
```

To display a list of tables in your database, you can use:

```
sqlite> .tables
```

This should return:

```
users
```

- the transactions table should contain the following fields:
    - u_from: the user from which the transaction is sent.
    - u_to: the user to which the transaction is going.
    - t_date: the transaction date
    - t_amount: the transaction amount
  - The primary key is using multiple names combining u_from and t_date.
- To create the table:

```
sqlite> create table transactions(u_from text, u_to text, t_date integer, t_amount
text, PRIMARY KEY(u_from,t_date), FOREIGN KEY (u_from) REFERENCES users(u_name),
FOREIGN KEY (u_to) REFERENCES users(u_name));
```

  - The table has two foreign keys that reference the u_name from the users table.

Now running the .tables command:

```
sqlite> .tables
```

should return:

```
transactions  users
```

Third, to insert data into the users database, we can use the insert statemetn as follows:

```
sqlite> insert into users (u_name, p_word) values ("Matt", "matt_pw"), ("Dave",
"dave_pw");

sqlite> insert into transactions (u_from, u_to, t_date, t_amount) values
("Dave","Matt",datetime("now"),50);
```

We can use the phrase datetime("now"), which is a SQL function to return the current time.

To view the data, we can use the select statement:

```
sqlite> select * from users;

Matt|matt_pw

Dave|dave_pw

sqlite> select * from transactions;

Dave|Matt|2019-10-15 01:54:09|50
```

To delete from your tables, you can use:

```
sqlite> delete from transactions;

sqlite> delete from users;
```

2- Data Management using Rust

First, we should not save passwords as text. To make secure passwords, we will use the bcrypt library:

Cargo.toml

```
bcrypt = "0.6.1"
```

From that bcrypt, we will need the methods: DEFAULT_COST, hash, verify; and the error type BcryptError.

Also, to manage our SQLite database from our code, we need SQLite crate:

Cargo.toml

```
sqlite = "0.25.0"
```

Second, we will need to create a struct *UserBase* to handle access to the database. So, we will add the following to our *main.rs* file

```
pub struct UserBase{
        fname:String,
}
```

We are going to combine two different error types: bcrypt errors and the SQLite errors. We can use another OR structure by implementing the erros as enum:

```
use bcrypt::{DEFAULT_COST, hash, verify, BcryptError};
use sqlite::Error as SqErr;
#[derive(Debug )]
pub enum UBaseErr{
        DbErr(SqErr),
        HashError(BcryptError)
}
```

To handle both types of error as UBaseErr, we can use the From trait allows for a type to define how to create itself from another type, hence providing a very simple mechanism for converting between several types.

```
impl From<SqErr> for UBaseErr{
        fn from(s:SqErr)->Self{
                UBaseErr::DbErr(s)
        }
}
impl From<BcryptError> for UBaseErr{
        fn from(b:BcryptError)->Self{
                UBaseErr::HashError(b)
        }
}
```

Third, let's implement some functions for the UserBase struct. We will start with the *add_user* method for adding a user to the database:

```
impl UserBase{
  pub fn add_user(&self, u_name:&str, p_word:&str)->Result<(),UBaseErr>{
    let conn=sqlite::open(&self.fname)?;
    let hpass=bcrypt::hash(p_word,DEFAULT_COST)?;
    let mut st= conn.prepare("insert into    users(u_name, p_word) values (?,?);")?;
```

```
    st.bind(1,u_name)?;
    st.bind(2,&hpass as &str)?;
    st.next()?;
    Ok(())
        }
}
```

Now, can you write the pay function that inserts into the transaction table:

```
pub fn pay(&self, u_from:&str, u_to:&str, amount:i64)->Result<(),UBaseErr>{
        let conn=sqlite::open(&self.fname)?;
        let mut st= conn.prepare("insert into    transactions (u_from, u_to, t_date,
t_amount)       values(?,?,datetime(\"now\"),?);")?;
        st.bind(1,u_from)?;
        st.bind(2,u_to)?;
        st.bind(3,amount)?;
        st.next()?;
        Ok(())
        }
```

3- Writing Tests

Question 11: Write some test cases to test all the functions implemented for UserBase Struct. Your tests should ensure that your code runs properly, and the functions' arguments are of correct types.

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